Specialization of Perceptual Processes

by

Ian D. Horswill

B.CSci. University of Minnesota (1986)
S.M. Massachusetts Institute of Technology (1988)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

© Ian D. Horswill, 1993

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author . .....
Department of Electrical Engineering and Computer Science
May 14, 1993

Certified by .....
Rodney Brooks
Associate Professor of Computer Science
Thesis Supervisor

Certified by .....
Lynn Andrea Stein
Class of 1957 Assistant Professor of Computer Science
Thesis Supervisor

Accepted by .....
Campbell L. Searle
airman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
JUL 09 1993
ARCHIVES
LIBRARIES

Specialization of Perceptual Processes

by

Ian D. Horswill

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 1993, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this dissertation, I will discuss the use of vision to support concrete, everyday activity. I will argue that a variety of interesting tasks can be solved using simple and inexpensive vision systems. I will provide a number of working examples in the form of a state-of-the-art mobile robot, Polly, which uses vision to give primitive tours of the seventh floor of the MIT AI Laboratory. By current standards, the robot has a broad behavioral repertoire and is both simple and inexpensive (the complete robot was built for less than $20,0000 using commercial board-level components).

The approach I will use will be to treat the structure of the agent's activity—its task and environment—as positive resources for the vision system designer. By performing a careful analysis of task and environment, the designer can determine a broad space of mechanisms which can perform the desired activity. My principal thesis is that for a broad range of activities, the space of applicable mechanisms will be broad enough to include a number mechanisms which are simple and economical.

The simplest mechanisms that solve a given problem will typically be quite specialized to that problem. One thus worries that building simple vision systems will be require a great deal of *ad-hoc* engineering that cannot be transferred to other problems. My second thesis is that specialized systems can be analyzed and understood in a principled manner, one that allows general lessons to be extracted from specialized systems. I will present a general approach to analyzing specialization through the use of transformations that provably improve performance. By demonstrating a sequence of transformations that derive a specialized system from a more general one, we can summarize the specialization of the former in a compact form that makes explicit the additional assumptions that it makes about its environment. The summary can be used to predict the performance of the system in novel environments. Individual transformations can be recycled in the design of future systems.

Thesis Supervisor: Rodney Brooks
Title: Associate Professor of Computer Science

Thesis Supervisor: Lynn Andrea Stein
Title: Class of 1957 Assistant Professor of Computer Science

Contents

1 Introduction **12**
- 1.1 Example . . . 12
  - 1.1.1 A more efficient algorithm . . . 16
  - 1.1.2 Analysis of the algorithm . . . 16
- 1.2 Preview of results . . . 17
  - 1.2.1 Lightweight vision . . . 17
  - 1.2.2 Mobile robotics . . . 18
  - 1.2.3 Analysis of specialization . . . 19
- 1.3 Roadmap . . . 20

2 Introduction to the Polly system **21**
- 2.1 Task and environment . . . 21
- 2.2 Software architecture . . . 22
- 2.3 Detailed example . . . 25
  - 2.3.1 Patrolling . . . 25
  - 2.3.2 Giving tours . . . 27
- 2.4 Programming language issues . . . 28
- 2.5 Hardware design . . . 29
- 2.6 Other visually guided mobile robots . . . 32
  - 2.6.1 Systems with geometric maps . . . 32
  - 2.6.2 Non-geometric systems . . . 32
  - 2.6.3 Outdoor road following . . . 33

3 Lightweight vision **35**
- 3.1 Background . . . 35
  - 3.1.1 Problems with reconstruction . . . 36
  - 3.1.2 Active vision and task-based vision . . . 37
- 3.2 Building lightweight vision systems . . . 38
  - 3.2.1 Resources for simplifying vision . . . 38
  - 3.2.2 Distributed representation and mediation . . . 40
- 3.3 Other related work . . . 41
- 3.4 Summary . . . 42

4 Analysis of specialization **44**
- 4.1 Introduction . . . 44

4.1.1 Computational minimalism 45
4.2 Preview 46
4.3 Habitat Constraints 48
4.3.1 Computational properties of large scale space 48
4.3.2 Computational properties of kitchens 49
4.3.3 Computational properties of office environments 50
4.3.4 Background texture constraint 51
4.3.5 Ground plane constraint 53
4.4 Specialization as optimization 54
4.4.1 Transformations and optimizations 54
4.4.2 Derivations 55
4.4.3 Synthesis versus post hoc analysis 56
4.5 Analysis of first order, 1 degree of freedom control systems 56
4.6 Related work 58

5 Analysis of simple visual systems **61**
5.1 Semantics 62
5.1.1 Derivability and equivalence 63
5.2 Equivalence of perceptual systems 64
5.3 Unconditional equivalence transformations 65
5.4 Conditional equivalence transformations 67
5.4.1 Salience functions and figure/ground separation 68
5.4.2 Depth recovery 69

6 Analysis of discrete control problems **72**
6.1 Definition 74
6.1.1 Discrete control with hidden state 75
6.1.2 Externalization of internal state 75
6.2 Progress functions and hill-climbing 76
6.3 Reduction and isomorphism 78
6.4 Cartesian products and separability 80
6.5 Optimization of control policies 82

7 The core visual system **84**
7.1 Computation of depth 85
7.2 Detection of carpet boundaries 88
7.3 Vanishing-point detection 90
7.4 Person detection 94
7.4.1 Symmetry detection 94
7.4.2 The protrusion test 95
7.4.3 Reliability 95
7.5 Gesture interpretation 95
7.5.1 Foot waving 96
7.5.2 The first nod detector 96
7.5.3 The second nod detector 99

7.6 Summary 100

8 Low level navigation **101**
8.1 Speed controller 102
8.2 Corridor follower 102
8.2.1 Aligning with the corridor 103
8.2.2 Avoiding the walls 103
8.2.3 Integrating the control signals 104
8.3 Wall follower 105
8.4 General obstacle avoidance 105
8.5 Ballistic turn controller 105
8.6 Steering arbitration 106
8.7 The FEP bump reflex 106
8.8 Odometric sensing 107

9 High level navigation **108**
9.1 Formal problem definition 108
9.2 Simplifying navigation 110
9.2.1 Reduction to the search of a small graph 111
9.2.2 Removal of search 112
9.2.3 Summary 113
9.3 Navigation in Polly 113
9.3.1 The navigator 114
9.3.2 The unwedger 115
9.4 Place recognition 115
9.5 Patrolling 120
9.6 Sequencers and the plan language 120
9.7 Giving tours 123

10 Experiments with Polly **125**
10.1 Speed 126
10.1.1 Processing speed 126
10.1.2 Driving speed 127
10.2 Complete test runs 128
10.3 Other environments 128
10.3.1 Tech Square 129
10.3.2 The Brown CS department 131
10.4 Burn-in tests 133
10.5 Limitations, failure modes, and useful extensions 135
10.5.1 Low-level navigation 135
10.5.2 The navigator 137
10.5.3 The unwedger 138
10.5.4 Place recognition 138
10.5.5 Camera limitations 138
10.5.6 Multi-modal sensing 139

11 Summary and conclusions 140
11.1 Why Polly worked 140
11.2 Lightweight vision 141
11.3 Studying the world 142

A The frame database 144

B Log of the last burn-in run 150

List of Figures

1-1 Office image 13
1-2 Texture in the office scene. 15
1-3 Viewing a textureless cliff 15
1-4 The carpet blob. 16

2-1 Patrol pattern 23
2-2 Gross anatomy of Polly 23
2-3 High level states 24
2-4 Leaving the office 26
2-5 Polly's habitat 27
2-6 Basic components and layout of the robot 30
2-7 Computational components and datapaths within the robot hardware 30

4-1 Effect of perspective projection on fourier spectrum of a surface patch 53

5-1 A camera viewing a ground plane. 69
5-2 Shape requirements of the ground plane constraint 71

7-1 Major components of the core vision system 85
7-2 Computation of freespace from texture and height 86
7-3 Source code for `find-dangerous` 89
7-4 Source code for `carpet-boundary?` 91
7-5 The vanishing point computation 92
7-6 Source code for `vanishing-point` 93

8-1 The low level navigation system 101
8-2 The corridor following problem 102
8-3 Nearest points in view 104

9-1 Freespace of the 7th floor 109
9-2 Graph structure of environment 112
9-3 The high level navigation system. 113
9-4 Landmarks in Polly's environment and their qualitative coordinates. . 116
9-5 Example place frames. 116
9-6 Source code for `frame-matcher` 118
9-7 Source code for matching process. Code has been slightly simplified by removing declarations and code for computing image differences. . 119

9-8 Source code for `find-districts` 121
9-9 Sequencer for leaving office 122
9-10 Sequencer for offering tour 123
9-11 Sequencer for giving a tour 124

10-1 Detail of elevator lobby area 129
10-2 Transcript of first test run. 130
10-3 Transcript of first test run (contd.) 131
10-4 Transcript of second test run. 131
10-5 Transcript of third test run. 132
10-6 Transcript of third test run (contd.). 133
10-7 Transcript of the last test run. 134
10-8 Transcript of the last test run (contd.). 135
10-9 Typical robot paths through the playroom after workmen 136
10-10The shadow problem 137

List of Tables

7.1 Partial list of visual percepts 85
7.2 Habitat constraints used for depth-recovery. 88
7.3 Habitat constraints used by the vanishing point computation. 92
7.4 Constraints used in nod detection 99
7.5 Habitat constraints assumed by the core vision system 100

10.1 Processing and driving speeds of various visual navigation systems . . 126
10.2 Execution times for various levels of competence 127

Acknowledgements

I would like to thank my advisors, Rod Brooks and Lynn Stein, for their support and encouragement. My time as a graduate student would have suffered badly had they not been here for me at the right time. I would also like to thank the members of my committe—Tom Knight, Shimon Ullman, and Eric Grimson—for their time, patience, and support. I would particularly like to thank Eric for the time he put in talking to me and giving me suggestions when he was not (yet) even on the committee. This document also owes much to Patrick Winston for his comments on my talks.

Ones fellow graduate students are perhaps the most important part of ones graduate education. Phil Agre, David Chapman, and Orca Starbuck taught me much before they moved on. Phil was in many ways another advisor. The members of the Mobot lab, particularly Maja Mataric, Dave Miller (not a graduate student and perhaps not a mobot person), Anita Flynn, and Colin Angle, were very patient with me and did everything from reading paper drafts to showing me how to build a power supply. My various office mates, Eric Aboaf, Paul Resnick, David Michael, Jose Robles, and Tina Kapur gave me comments on papers, talks, ideas, and general nonsense, and put up with me when I was moody. The local vision people and honorary vision people, particularly Karen Sarachik, Mike Bolotski, Henry Minsky, Steve White, Sandy Wells, David Jacobs, and David Clemens, gave me useful advice, encouragement, and skepticism. Other members of the lab, Jeanne Speckman, Annika Pfluger, Laurel Simmons, Jonathan Meyer, and Ron Wiken made my life much easier. Patrick Sobalvarro made it much funnier. Sigrid Unseld was as good a friend as one can ask for.

The local hardware hackers, particularly Tom Knight, Mike Bolotski, Grinnell Moore, Mike Ciholas, Henry Minsky, Anita Flynn, and Colin Angle have earned my undying gratitude for putting up with my EE questions while I was building my robot.

A number of people have helped me through my thesis by giving me words of encouragement or enthusiasm while I was at conferences. I would particularly like to thank Bruce Donald, Avi Kak, Stan Rosenschein, Takashi Gomi, Ron Arkin, and Drew McDermott.

My housemates over the years, Hilda Marshall, Craig Counterman, John Romkey, Rob and Cath Austein, Diana Walker, Rick and Rose Stout, and David Omar White, have remained good friends, even when I have done a poor job of keeping in touch. They have always been there for me.

Finally, I would like to thank my parents for all their love and care, and my wife,

Louise, for being my wife.

Support for this research was provided in part by the University Research Initiative under Office of Naval Research contract N00014–86–K–0685, and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014–85–K–0124.

Chapter 1

Introduction

In this dissertation, I will discuss the use of vision to support concrete, everyday activity. I will argue that a variety of interesting tasks can be solved using simple and inexpensive vision systems. I will provide a number of working examples in the form of a state-of-the-art mobile robot, Polly, which uses vision to give primitive tours of the seventh floor of the MIT AI Laboratory. By current standards, the robot has a broad behavioral repertoire and is both simple and inexpensive (the complete robot was built for less than $20,000 using commercial board-level components).

The approach I will use will be to treat the structure of the agent's activity—its task and environment—as positive resources for the vision system designer. By performing a careful analysis of task and environment, the designer can determine a broad space of mechanisms which can perform the desired activity. My principal thesis is that for a broad range of activities, the space of applicable mechanisms will be broad enough to include a number mechanisms which are simple and economical.

The simplest mechanisms that solve a given problem will typically be quite specialized to that problem. One thus worries that building simple vision systems will be require a great deal of *ad-hoc* engineering that cannot be transferred to other problems. My second thesis is that specialized systems can be analyzed and understood in a principled manner, one that allows general lessons to be extracted from specialized systems. I will present a general approach to analyzing specialization through the use of transformations that provably improve performance. By demonstrating a sequence of transformations that derive a specialized system from a more general one, we can summarize the specialization of the former in a compact form that makes explicit the additional assumptions that it makes about its environment. The summary can be used to predict the performance of the system in novel environments. Individual transformations can be recycled in the design of future systems.

1.1 Example

Suppose we view vision as being the problem of answering questions about the environment using images. Figure 1-1 shows an image of my office taken from my robot. We might want to determine whether the robot should turn left or right so as to avoid

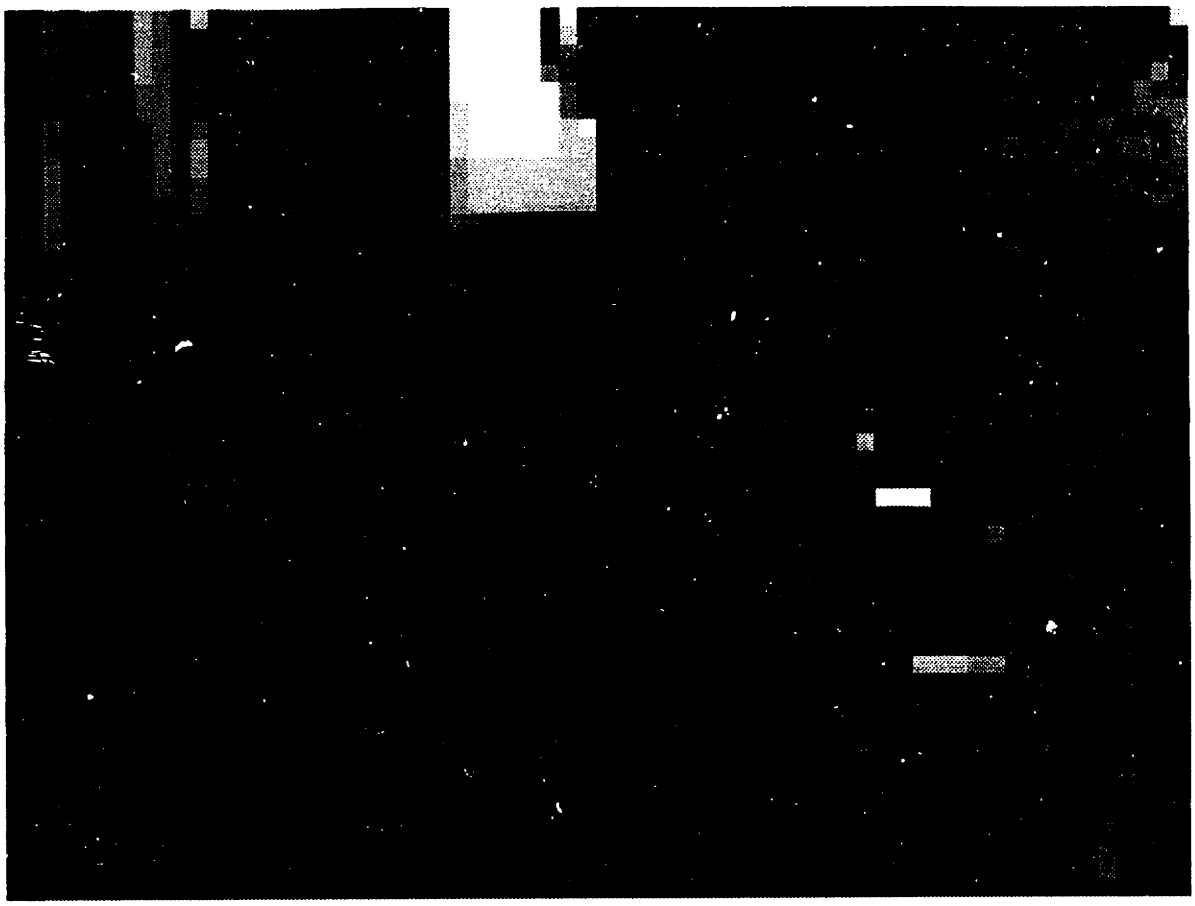


Figure 1-1: Image of my office taken from the robot's camera. The dots in the lower middle of the image are artifacts due the quantization in the rendering process. The structure in the lower right hand portion of the image is a 5-legged office chair. The structures in the top-left are (left to right) a doorway viewed from an oblique angle, a small trash can, and a file cabinet. The homogeneous region in the lower and middle left is the carpet.

the objects in the scene. This amounts to the problem of finding which regions of the floor are free and which have objects on top of them. The correct answer is that the robot should turn left, since there are obstacles nearby on the right but there is clear floor on the left. The fundamental difficulty here, as with most vision problems, is that the projection process of the camera loses information, depth information in particular, and so we cannot uniquely determine the structure of the scene without additional information either in the form of extra images or of extra information assumed about the problem.

A common way of solving the problem would be to build a complete depth map of the scene using multiple images. We can then project the features in the depth map into the floor plane to determine which parts of the floor do and do not have obstacles on top of them.

The simplest version of this is two use two cameras in a stereo configuration. Distinctive features (usually edges) can be found in the two images and matched to one another. Given the matching of the features, we can compute each feature's shift due to parallax. From the camera positions and the parallax data, we can compute the positions of the surface patches in the world from which the individual features were imaged (see Barnard and Fischler [14]).

The stereo approach is a perfectly reasonable approach, but it does have two undesirable features. First, it is computationally expensive, particularly in the matching phase. It also requires that features be localized accurately and reliably, which usually means the use of high resolution data, which requires more computational power. A more important problem however is that the floor in this environment is textureless and therefore featureless. Figure 1-2 shows a map of the image in which pixels with significant texture (actually, significant intensity gradients) are marked in white. The floor is uniformly black. The stereo process cannot make any depth measurements in the region of the image which is most important to the robot because these are no features to be matched there.

This problem is easily remedied. Since the floor is always flat, it is reasonable for the stereo system to interpolate a flat surface in the absence of texture from which to compute depth. However, it is important to remember that the stereo system is then working not because it is measuring the depth of the floor directly, but because it is making a smoothness assumption which happens to be true of the floor. This need not be true in general. The floor could slope gently or suddenly. There could even be a cliff. While a sudden discontinuity in depth, such as a cliff, would typically generate image features right at the discontinuity, there would still be no features on either side of the discontinuity that could be used to detect the cliff (see figure 1-3).

This brings out two important points. First *truly general systems are extremely rare*, and so claims of generality should be considered carefully. Often the mechanisms we build have hidden assumptions which can fail to be true. These can be particularly difficult to diagnose because we typically choose test data that fit the assumptions. I do not wish to argue that implicit assumptions about the world are bad. Quite the contrary: those assumptions can lead to great improvements in performance. Rather, we should make *informed decisions* about our use of specialization.

Figure 1-2: The pixels with significant texture. Pixels marked in white differ from their neighbors above and to the right by total of at least 15 grey levels out of a possible 510. Pixels marked in black differ by less than 15 grey levels. The image was first smoothed with a 3×3 filter to remove camera noise. Note that the floor is uniformly black.

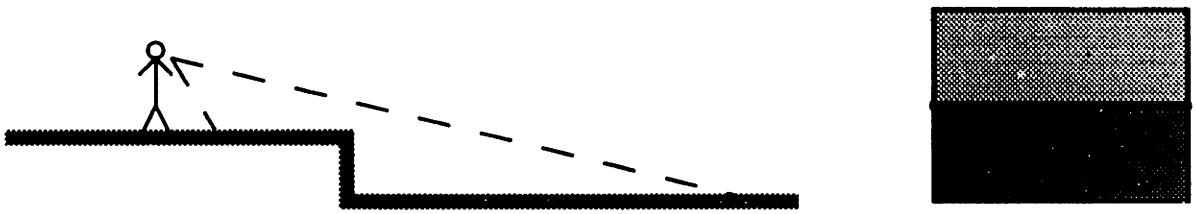


Figure 1-3: An observer views a cliff of a textureless surface (left). Although variations in lighting of the two sides of the cliff may produce a local variation in image brightness at the point of discontinuity (right), there is still no texture in the image above or below the discontinuity which would allow the observer to infer the depth, or even the presence, of the cliff.

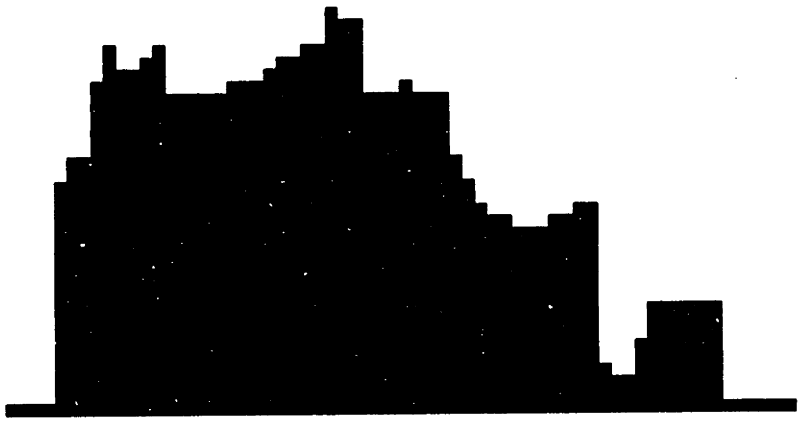


Figure 1-4: The set of textureless pixels corresponding to the carpet. Note that the blob is taller where there is more exposed carpet.

1.1.1 A more efficient algorithm

When the stereo system works on the scene in figure 1-1, it works because the floor is flat and the obstacles have texture. We can make a different system to solve the problem, one that is much more efficient, by using these facts directly.

Let us treat the lack of texture on the floor as a positive feature of the environment. Notice that the floor forms a single, connected black blob at the bottom of figure 1-2. This blob is shown alone in figure 1-4). I will call this the carpet blob. The carpet blob is easily computed by starting at the bottom of the screen and tracing up each image column until a textured pixel is found. The set of pixels skipped over will be the blob.

Notice that the heigh of the blob varies with the amount of exposed floor in the corresponding direction so the number of pixels skipped in a given column gives us a rough and ready measure of the amount of free space in that direction. This suggests the following algorithm for solving the original problem: first, find the textured pixels in the image, then extract the carpet blob, and then turn in the direction in which the carpet blob is taller. This algorithm is the basis of much of Polly's navigation capabilities. It can easily be executed in real time on a bottom of the line Macintosh.

1.1.2 Analysis of the algorithm

We can understand the relationship of the stereo-based and blob-based algorithms as follows. Both systems are trying to determine whether there is more free space in front of the robot on the left side or the right side of the image. The stereo system measures this directly by computing a depth map and projecting it into the floor plane. Since we are only concerned with determining which side is larger, however,

we do not need to know the exact distances in any particular units of measure. Any measure which increases monotonically with the distance will work, provided that we use the same measure on both sides. We can thus substitute *any system* that computes a monotonic function of the distance for the stereo system. More importantly, we do not even need to know what the monotonic function is. It could vary from moment to moment so long as it was used uniformly within a given image. It has been known at least since Euclid that image plane height is a monotonic function of distance. This means, roughly, that if all the obstacles rest on the floor, then we can substitute the image plane height of the obstacle for the stereo system, provided that we have some way of labeling each pixel as being either obstacle or carpet. A general carpet detector which could recognize any carpet (or, equivalently, any obstacle) might be be more difficult to build than the stereo system. However, the carpet in this environment has a very predictable appearance: it has no texture. This means that we can substitute a texture detector for the general carpet detector or obstacle detector.

We can summarize this analysis with the following general principles:

- We can substitute any monotonic measure of a quantity for a calibrated measure, provided that the measure will only be used for comparisons.
- We can substitute height in the image plane for some other distance calculation, provided that all objects rest on the floor and there is some way of classifying pixels as being floor or object.
- We can substitute a texture detector for a floor detector, provided that the floor is textureless.

These principles concisely describe the specialization of the blob-based algorithm. Each describes a general transformation from a possibly inefficient algorithm to a more efficient one, along with the conditions on the task and environment which make it valid. These transformations can be applied to the design of other systems or used to predict and modify the performance of the original system. For example, if we wanted to use the blob-based algorithm in an environment with a textured carpet, we would have to abandon the last transformation, but we would still be able to use the other two. If there was some property other than texture which allowed carpet pixels to be easily classified, then we could use that property as the basis of a new transformation.

1.2 Preview of results

This dissertation contributes to three areas: the design of efficient vision systems, the design of mobile robots, and the analysis of specialized systems.

1.2.1 Lightweight vision

Vision is traditionally thought of as being very expensive. A number of researchers have recently argued that vision can be significantly simplified by using the dynamics

of the agent's interaction with its environment or by specializing the vision system to a particular task. (See Bajczy [10], Horswill [65], Ballard [11], Ikeuchi [66], or Aloimonos [6]. See also section 3.1.2.) We can simplify vision further, however, by taking into account specific properties of the agent's environment. We can view both the task and the environment as resources for the designer. The task imposes concrete constraints on the information which the system must extract from the image and the performance with which it must extract it. Far from being bad, these constraints are a good thing, for they tell the designer not only what is required of the agent, but also what is *not required* of it.

Knowing the constraints is important because it allows the designer to make trade-offs intelligently. Performance has many parameters. These parameters typically can not be optimized simultaneously, so the designer must decide what to optimize and what not to optimize. Knowing the task constraints allows the designer to make these choices effectively. A similar situation holds for choosing what information to extract from the image. Resolution is a useful case in point. For a given amount of computing power, the rate at which agent will be able to process frames will be bounded above by the inverse of the number of pixels in the image, assuming the program in the agent at least reads every pixel. The number of pixels is quadratic in the linear resolution of the image however. This means that of you want to double the accuracy with which you can localize a feature in the image, you must drop the rate at which you process images by at least a factor of four. Most vision research has focused on relatively high resolution images, whereas all the algorithms discussed in this work operate on very low resolution images (64×48 or 16×12).

Understanding the environment allows the designer to make optimizations such as the substitution of low level image properties for high level object properties. The algorithm discussed above is a case in point: because of the special properties of the environment, we can substitute a low level image property, texture, for a high level symbolic property, namely being-an-obstacle. Understanding the environment can also tell the designer that certain pathological situations cannot develop or that certain performance requirements can be relaxed. All of these serve to broaden the space of mechanisms available to the designer.

For a given task and environment, the space of mechanisms which will solve that task in that environment is typically large. My principal claim is that for many real world (task, environment) pairs, the space is large enought to include a number of very efficient mechanisms. I call these very efficient mechanisms, "lightweight" vision systems.

1.2.2 Mobile robotics

The principal contribution of this dissertation to the field of mobile robotics is to demonstrate that it is possible to build a robust and economical vision-based robot using only on-board computation and standard commerical hardware. The Polly robot uses a single 16MIP digital signal processor, a Texas Instruments TMS320C30, for nearly all its computation. A modified version of the chip is available for $35 in quantity. Efforts are underway at MIT to build a system which can run most of the

Polly algorithms for less than $500 (see Gavin and Yamamoto [51]. The robot is able to process video frames as fast as its frame digitizer can convert them (15 frames per second). With each frame, the robot recomputes all its visual percepts, compares the scene to its complete database of landmarks, and updates its motor velocities.

Polly patrols the seventh floor of the MIT Artificial Intelligence Laboratory, searching for visitors who may want tours. When it finds a visitor, it offers them a tour. If they gesture properly, the robot leads the vistor around the lab, giving informative speeches as it recognizes landmarks. This task involves a very wide repertoire of behaviors by the current standards of the field. It can follow paths, recognize landmarks, detect the presence of people, interpret simple gestures, and choose paths to a specified landmark.

The robot is extremely well tested. The lower layers of its control system have been running for over a year and have seen several hundred hours of service. During that time, the environment has steadily changed its geometry and appearance: the student population rose and fell, furniture was rearranged, a small mountain range was built in part of the lab, carpets were shampooed, and office lighting was complete replaced. The robot tollerated all these changes, although it sometimes had to be modified slightly to cope with them (see chapter 10). The robot has also been tested in other, similar environments.

1.2.3 Analysis of specialization

There has been renewed interest within the AI community in the relationship between agents and their environments, particularly within the artificial life, biologically-based AI, and situated action communities (see section 4.6 for a survey). We can analyze this relationship formally by deriving an agent that is specialized to its environment from a hypothetical general agent through a series of transformations that are justified by particular properties of the agent's environment. In performing the derivation, the designer divides the agents specialization into a discrete set of reusable transformations, each of which is paired with an environment property that makes it valid. I call such properties "habitat constraints" because the set of such constraints define the agent's habitat. I will generally refer to the transformations as optimizations. These are optimizations in the sense of compiler optimizations: they are transformations which improve performance, but do not necessarily yield optimal performance.

The advantage of the derivation is that it makes explicit the agent's implicit assumptions about its environment. These assumptions can then be used in the design or analysis of other agents. Thus it provides a way of recycling the experience gained in designing specialized systems. Although it is not possible in general to fully automate the design of specialized systems, this kind of post-hoc analysis can eventually allow us to develop cookbook methods for a broad range of problems.

The technique is not limited to the analysis of vision systems. I will present examples of its application to vision, motor control, and discrete action selection.

1.3 Roadmap

The next three chapters present high level introductions to Polly, lightweight vision, and specialization, respectively. Chapter 2 describes Polly's task and environment, its high level software design, and discusses the language and hardware used to implement it. It also provides a detailed example of a run of the robot and a survey of other vision-based robot navigation systems. Chapter 3 discusses the history of vision research, particularly reconstructionism and the recent move toward task-based and active vision. It then discusses how task and environment can be used as resources for building lightweight vision systems. Chapter 4 draws out the general framework for analyzing specialization and introduces many of the properties of Polly's environment that will be used later to simplify its vision problems. Subsequent chapters are more detailed and technical.

Chapter 5 gives the formal basis for applying the transformational theory to a class of simple vision systems. Chapter 6 does the same with the problem of discrete action selection. The reader who is less interested in formalism may wish to skip these.

Chapters 7, 8, and 9 describe in detail the robot's vision, low level navigation, and high level navigation systems, respectively, and use the analytical framework to explain their performance.

Chapter 10 discusses a number of experiments made on Polly over the last year or so, gives a sense of the robot's reliability, and categorizes its failure modes.

Chapter 11 summarizes the key points of the thesis and gives additional conclusions.

Chapter 2

Introduction to the Polly system

Polly is a low-cost vision-based robot built to explore the use of domain constraints in the design of lightweight vision systems. Polly lives on the 7th floor of the MIT AI Laboratory and gives simple tours of the 7th floor. Polly is interesting for a number of reasons. First, it has a relatively wide range of capabilities. It can navigate, recognize places, detect people, and understand their gestures. Second, it is simple and inexpensive, costing only $20K to build from off-the-shelf, board-level components (a faster version could now be built for less that $10K). Third, it is very fast, running between 0.75 and 1 meters per second, with its perception and control systems running at 15Hz. Finally, it performs its tasks almost exclusively using vision. Vision-based autonomous robots are relatively rare, and cheap, fast, autonomous systems with wide behavioral repertoires have not been built previously.

In this chapter, I will describe the task, environment, and basic structure of the robot. Section 2.1 describes the robot's task and habitat. Section 2.2 describes the high level architecture of the robot. Section 2.3 gives a detailed run of the robot and describes the operation of the robot during the run. Section 2.4 briefly describes the programming language used. Section 2.5 describes the basic hardware components of the robot, their capabilities, and their interconnections. Finally, section 2.6 describes previous visually guided mobile robots. Subsequent chapters will discuss the software components in more detail.

2.1 Task and environment

Polly's environment is the network of corridors on the seventh floor of the AI Laboratory at MIT (seee figure 2-1). Its task is to patrol the lab, find visitors, and give them tours. Its patrol pattern is shown in figure 2-1. As it drives through the hallways, it searches for visitors who want tours. Since it can only look downward at the floor, it has to detect people by looking for their legs. The only way it can distinguish a visitor from a normal occupant is to rely on the fact that the normal occupants are probably sick of it and will leave it alone, whereas visitors will actively investigate

it.[1] For this reason, Polly only responds to people who stand directly in front of it. It ignores people who causally pass by it or who lean against the wall. When Polly does find a leg-like object directly in front of it, it introduces itself and offers a tour, saying that the person should wave their foot around if they want a tour. If the person indicates that s/he would like a tour by gesturing with their foot, Polly leads the person around the lab, making pithy comments and giving canned speeches when it recognized landmarks. For example, when it recognizes the T.V. lounge, it says "This is the T.V. lounge. We waste a lot of time here." When Polly returns to the place where it picked the visitor up, it thanks the visitor and says goodbye. It then looks for another visitor.

A typical scenario might be the following:

Event	Speech
Polly approaches visitor	Hello. I am Polly. Would you like a tour? If so, wave your foot around.
Visitor waves foot	Thank you. Please stand to one side.
Visitor moves	Thank you. Please follow me.
Polly drives	I can avoid obstacles, follow corridors, recognize places, and navigate from point to point.
Keeps driving	My vision system runs at 15 frames per second on a low cost computer
Robot passes vision lab	On the right here is the vision lab. By the way, I don't understand anything I'm saying.
Robot enters T.V. lounge	This is the T.V. lounge. We waste a lot of time here.
Passes office	This is Karen and Mike's office.
Passes office	This is the office of Anita Flynn.
Enters playroom	This is the playroom.
	This is the end of the tour. Thank you and have a nice day.
Robot drives off.	

The robot continually alternates between searching for visitors and giving tours until it is told to go home (by throwing a switch on the robot's control panel). When the go-home switch is thrown, the robot drives to my office and parks by my desk.

2.2 Software architecture

[1] Or so the theory went when I designed the system. In practice, it is exactly the other way around visitors, not wanting to cause any problems, get out of the robot's way; the denizens of the lab however are perfectly happy to harass it in an effort to test its limits.

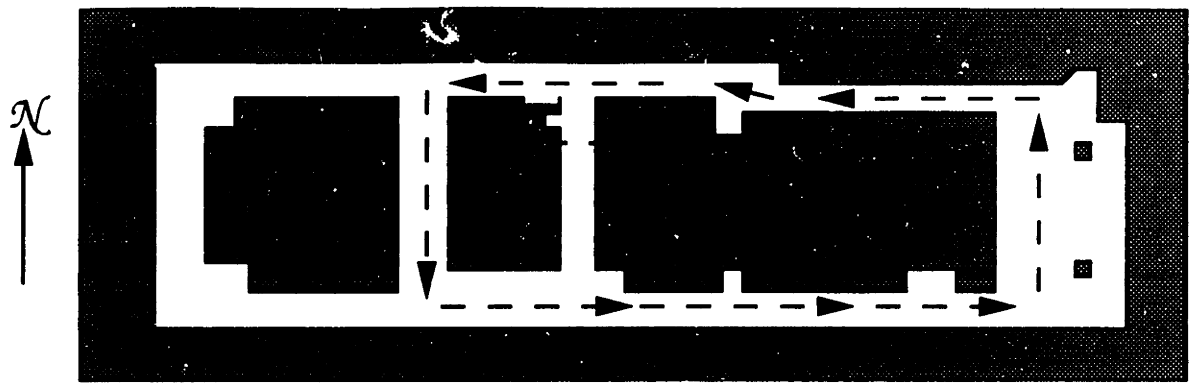


Figure 2-1: The layout of the 7th floor of the lab, and Polly's patrol pattern within it.

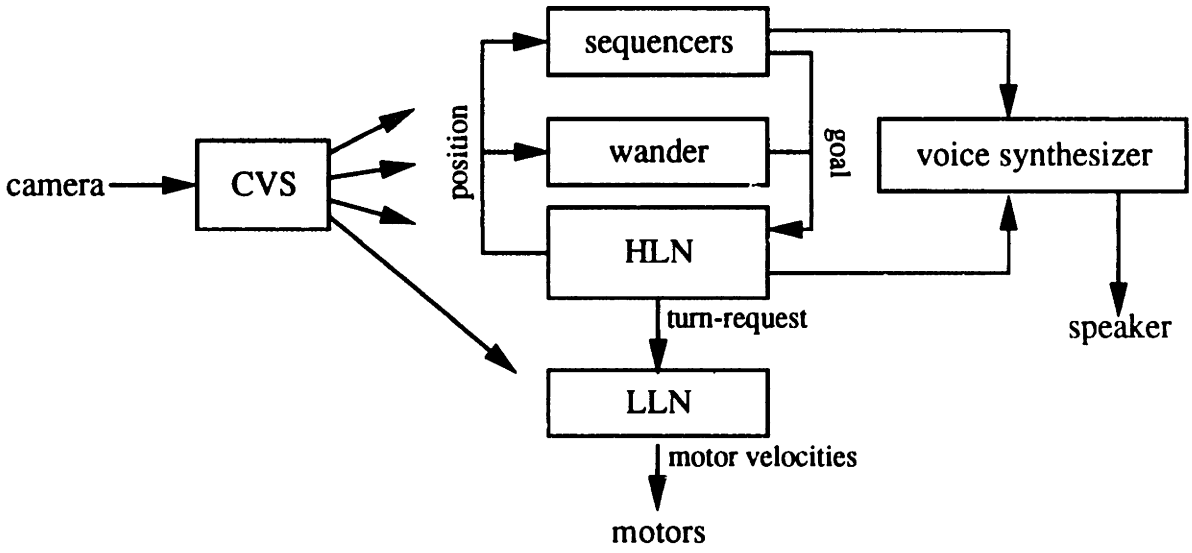


Figure 2-2: Polly's gross neuroanatomy.

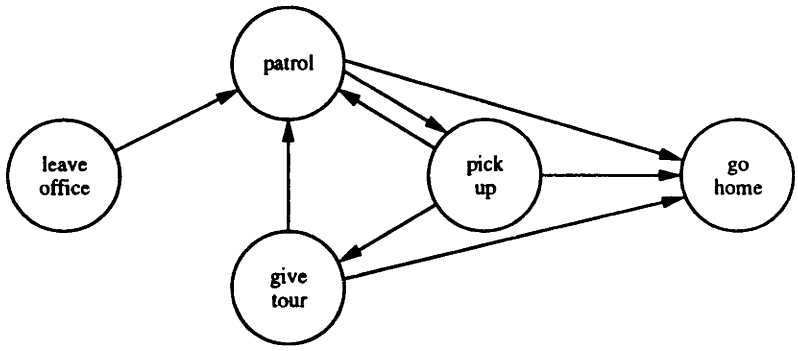


Figure 2-3: High level states and transitions

Polly is meant to be thought of as a group of communicating asynchronous processes connected by fixed links (see, for example, Brooks [24] or Rosenschein and Kaelbling [108]). These can be grouped into the high level structures shown in figure 2-2.

A group of related visual processes, the core vision system (CVS), transform images into high level percepts that encode specific relevant information such as whether the robot's path is blocked or whether there is a person in view. These percepts are effectively broadcast to all other parts of the robot.

Navigation is implemented by two groups of processes. The low-level navigation system (LLN), controls the motors and implements obstacle avoidance, path (wall or corridor) following, and switching from one path to another. It can also perform open-loop turns of specified angles. The high-level navigation system (HLN) matches landmarks to an internal map, and performs goal directed navigation. When a higher-level process gives the HLN a goal landmark, the HLN directs the LLN through a series of corridors leading to the goal. The goals are usually specified by the `wander` system which implements the patrol pattern by alternately instructing the high level navigation system to go to opposite corners of the patrol circuit (the playroom and the vision lab, see figure 2-5).

A number of Polly's tasks, such as offering tours, require the robot to perform fixed sequences of actions. Each such sequence is implemented by a special kind of process called a "sequencer" (see section 9.6).

Another set of processes control the voice synthesizer. The `tour-announce-place` process gives a canned speech whenever the robot recognizes a landmark and is in the process of giving a tour. The `tour-chatter` process generates periodic small talk during tours. Typical message would be "my vision system runs at 15 frames per second on a low cost computer," or "by the way, I don't understand anything I'm saying." The `messages` process generates random messages at random times when the robot is idle (not giving tours). This was done purely for the amusement of the author. The `cryer` process can be programmed to give a specific message at specified intervals. It is useful for turning the robot into a "town cryer." I have used it to announce talks. Finally the `chatter` process arbitrates requests from the other processes for access to the voice synthesizer. The design of these processes will not be discussed further.

The one piece of globally visible state in Polly is its high-level mode (see figure 2-3),

which is stored in the global variable `global-mode`. The values of `global-mode` are 0, meaning the the robot is starting up or going home, 1, meaning that it is on patrol, 2 meaning that the robot is giving a tour, and 3, meaning that it is in the process of offering a tour. The global mode is used by the wander process to determine whether it should drive in the patrol pattern, by the speech system to determine whether it should give tour speeches, and by the pick-up routine to determine whether it should attempt a pick-up. It is written by the pickup routine, the tour routines, and the routines for starting up and going home. This could have been implemented using local state information, such as having the different processes check to see which other processes were enabled, or by using a hormone-like mechanism (see Brooks [25]). In this case, my needs were simple enough that a global variable sufficed.

2.3 Detailed example

The robot always begins near my desk, facing south (see figure 2-4). It leaves the office by performing a fixed set of steps specified by a sequencer. First it drives south until it is blocked by an obstacle, at which point it should be at the file cabinet labeled (1) in figure 2-4. Then it turns left and drives until no longer sees a wall on the left, at which point it should be out of the office door (point 2 in the figure). It then moves forward a few feet, turns left again, and drives until it is blocked and facing east, at which point it should be facing the right-hand wall (point 3 in the figure). It then turns right 120 degrees so as to face the pillar and couch, and moves forward. The robot's obstacle avoidance mechanisms are sufficient to move it into the channel between the couch and the wall and off into the corridor (see point 4 in the figure). At this point, Polly's normal navigation systems are activated. The robot's position is initialized to "Ian's office" and the wanderer sets the goal landmark to be the vision lab (see figure 2-5).

2.3.1 Patrolling

The robot new begins patrolling. It is in the corridor between "Ian's office" and "Elevator lobby" in figure 2-5, travelling west. The HLN determines that it is east of its goal, the vision lab, and so allows the LLN to continue to move west. When the robot reaches the elevator lobby, the wall on the right-hand side suddenly vanishes from view, indicating that the robot has reached a right turn. The robot checks its map for a right turn and determines that it is at the elevator lobby. The entry in the map for the elevator lobby says that the robot should veer to the right to remain in the corridor, so the HLN instructs the LLN to make a small right turn. The LLN does so and continues across the open space of the elevator lobby until it reaches the other side. There, the LLN realigns with the corridor and continues west. As the robot drives past the leftmost corridor in figure 2-5 (the corridor between the vision lab and the kitchen), the left-hand wall vanishes from view and the vision system signals the presence of a left turn. The HLN finds the turn in its map and updates its position.

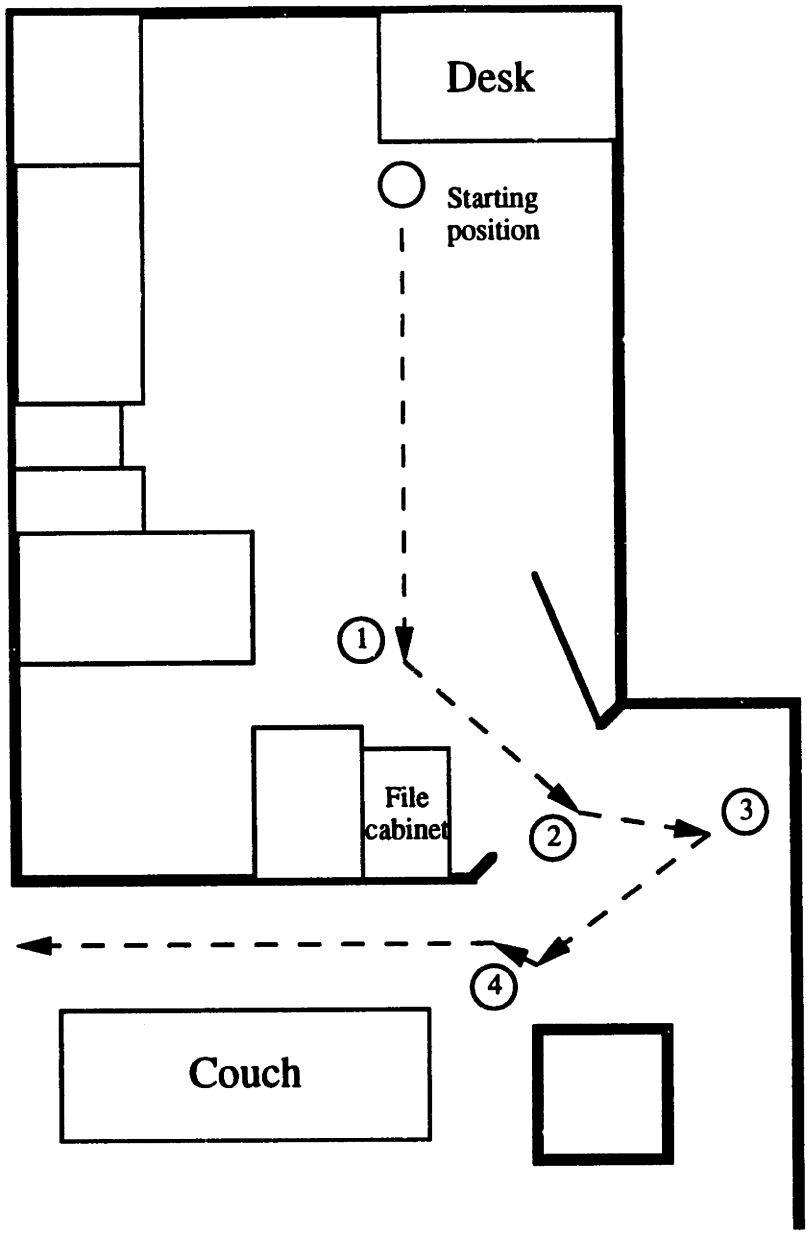


Figure 2-4: The robot's path as it leaves the office and enters the corridor. Both path and floor plan are only schematic—neither is drawn to scale.

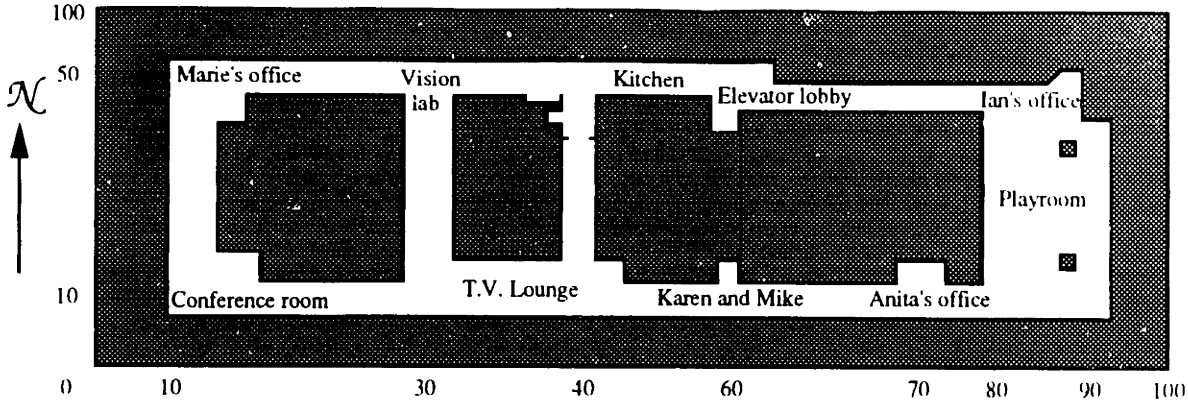


Figure 2-5: Polly's habitat, the 7th floor of the MIT AI laboratory. The diagram is not to scale, and the direction north has been defined for convenience, rather than geological accuracy.

When the robot reaches the vision lab, it detects another left turn and the HLN updates its position again. Now several things happen. First, the HLN notices that it has reached its goal. Then the wanderer notices that it has reached the vision lab and sets the goal to be the southern corner of the playroom. This, in turn, causes the HLN to wake up. The HLN determines that the goal is to the southeast. Since it is at a turn to the south, it instructs the LLN to turn 90 degrees to the left. The LLN turns left, and begins to align with the new (southern) corridor, as if nothing had happened, and proceeds along the corridor. When the robot reaches the southern wall (near the T.V. lounge in the figure), several things happen again. The LLN can no longer move any further and so it stops. The HLN notices that it has reached the southern wall, and so updates its position again. Since the goal is now east instead of southeast, the HLN tells the LLN to turn left again. The LLN turns, follows the wall through the T.V. lounge, and proceeds on its way. The robot continues, the HLN updating its position with the appearance of each left hand turn, until the robot reaches the playroom. Then the wanderer sets the goal back to the vision lab, the HLN tells the LLN to turn north, and the cycle repeats.

2.3.2 Giving tours

The robot continues to patrol until it encounters a person in the hallway. When this happens, a sequencer (`offer-tour`) is started. The sequencer inhibits the LLN and HLN and halts the robot. It then says "Hello, I am Polly. Would you like a tour? If so, wave your foot around." It then looks to see if there is any motion in the image. If not, it says "OK, have a nice day," and dis-inhibits the LLN and HLN. If there is motion, then it says "please stand to one side," and waits until it is no longer blocked. Then it says "OK, follow me," and starts a new sequencer, `give-tour`. `Give-tour` records the current position of the robot, and sets the robot's global mode to give-tour mode, which enables the `tour-chatter` and `tour-announce-place` processes. `Give-tour` waits for the robot to return to that position. In the mean time, the wanderer, HLN,

and LLN continue as if nothing had happened. Each time the HLN recognizes a new place, `tour-announce-place` wakes up and gives the speech listed in the map for that place. This leaves long periods of boring silence while the robot drives from one place to another, so the `tour-chatter` process inserts chatter messages in the pauses, roughly every 15 seconds. Eventually, the robot returns to the place it started from. Then `give-tour` wakes up, says "That's the end of the tour. Thank you and have a nice day," sets the global mode to "patrol" (which disables `tour-announce-place` and `tour-chatter`), and turns itself off. The robot then continues on its way, looking for a new visitor.

2.4 Programming language issues

While Polly is meant to be thought of as a parallel system, in practice it is implemented on a serial processor. The processor is programmed in a subset of Scheme.[2] The main loop of the program repeatedly grabs a new image, processes it, computes a new set of motor commands, transmits the commands, and waits for a new image. Each execution of the loop is referred to as a "clock tick."

Parallel processes are simulated using Scheme procedures, one procedure per process. Each procedure is called once per clock tick by the main loop. Occasionally, several simple processes are folded into a single procedure for efficiency. Communication channels between the processes are simulated with global variables. Nearly all of these variables are of thought of as wires carrying signals: each wire is updated on each clock tick by a single process. For example, the variable `direction` is written every clock tick by the process which reads the robot's rotational odometer. It always holds the current direction of the robot. A few global variables, such as the variable which holds the goal position of the robot, are set only rarely but may be set by many different processes. These are best though of as latches.

Nearly all of the vision code is implemented using a set of macros for vector processing built on top of Scheme. Images are represented as 1D vectors in row-major (raster) format so that the pixel at address (x, y) in a 64×48 image is the $x + 64y$'th entry in the vector. Two images can then be compared using a vector equivalent of `mapcar`:

```
(define (compare-images in1 in2 out)
  (map-vector! out
               =
               in1
               in2))
```

This procedure will take two input vectors of the same length and apply the = function to successive pairs of elements of the two inputs, writing the results (true or false)

[2]The actual dialect is called "Senselisp" (see Horswill [61][62]). Senselisp is essentially Scheme with garbage collection and run-time type checking removed, and compile-time type inference, full macros, and pointer arithmetic added. This modifications allow a relatively simple compiler to produce very efficient code. Until recently, the Senselisp compiler produced better code than the C compiler sold by the manufacturer of the computer.

to successive elements of the output. Vectors can be shifted around using pointer arithmetic, which allows adjacent pixels of an image to be compared using the vector macros. For example, the procedures

```
(define (bad-vertical-edge-detector in out)
  (map-vector! out
               =
               in
               (shift in 1)))
(define (bad-horizontal-edge-detector in out)
  (map-vector! out
               =
               in
               (shift in (raster-width in))))
```

detect vertical and horizontal edges by comparing a pixel with the pixel just to the right of it, and just below it, respectively. The function `raster-width` returns the number of pixels per line for a vector which is used to hold an image. `Shift` performs pointer-arithmetic. It takes a vector and an integer and returns the vector displaced by the specified number of elements. Note that this shifting will produce bizarre results at the boundary between the left and right sides of the image. In 2D, the left and right sides of the image are not connected, but in the 1D array, the first pixel of the 2nd line is right after the last pixel of the first, so the vertical edge detector above will find artifactual edges near the boundary. This could be avoided by using nested loops, but it is usually easier just to ignore the boundary pixels.

2.5 Hardware design

Because no commercial robot was available with an appropriate computer and frame grabber, I was forced to build the system myself from commercial board level components. This was not as terrible as it sounds. It mostly involved making connectors to link one board to another or screwing, glueing, tapeing, or velcro'ing the various components to one another. Roughly a year was required to build the robot, but nearly all of the time was spent reading catalogs, ordering parts, or waiting for parts to arrive. Less than a month was spent on actual assembly. Writing boot roms, down loaders, device drivers, and other systems software did take several months however.

The robot's hardware resources and physical layout are shown in figure 2-6. The robot's principal components are a commercial omni-directional robot base (Real World Interface model B12), a 32 bit digital signal processor (DSP) based on the Texas Instruments TMS320C30, with 65K words of SRAM (Pentek model 4283), a frame digitizing board (Data Translation model 1452), a black-and-white surveillance camera (Chinon CX-101), and a microcontroller front-end (Motorola MC68HC11) which connects the DSP to various low-speed peripherals (voice-synthesizer, input switches, LCD display).

The robot base contains motors, batteries, shaft encoders, a microcontroller to drive the motors, and a standard 9600 baud serial port over which to send commands

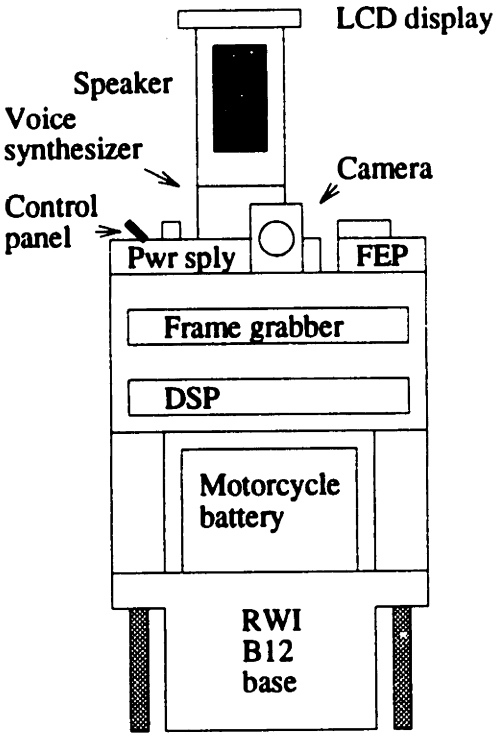


Figure 2-6: Basic components and layout of the robot.

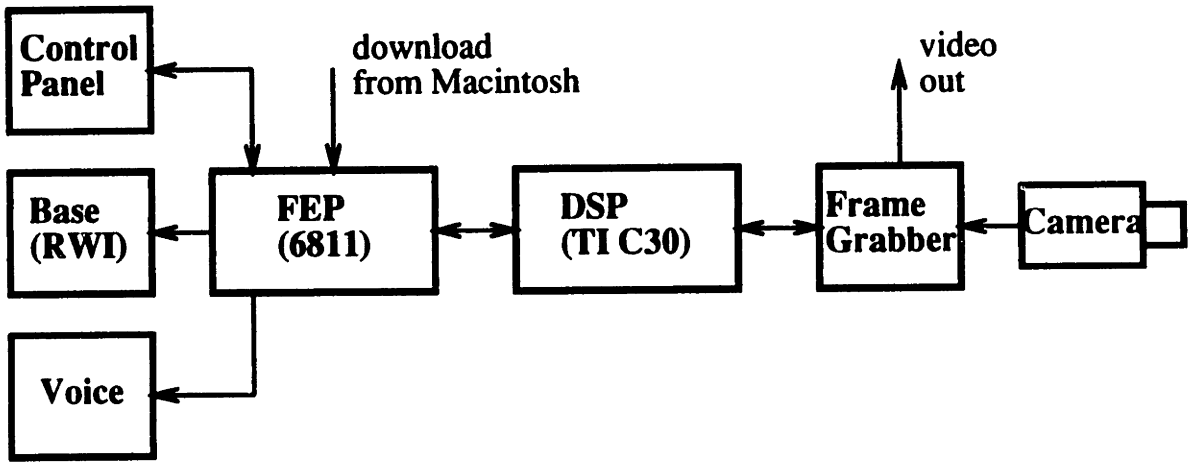


Figure 2-7: Computational components and data-paths.

to the microcontroller. The base has independent motors for driving forward and turning in place. Each motor can be controlled in force, velocity, or position space. The controller can also report the current position or velocity of either motor. The voice synthesizer accepts standard English text and converts it first to phonemes and then to audio.

Nearly all the interesting processing is done on board the DSP. The DSP is roughly a 16MIP machine. The DSP reads images from the frame grabber, performs all visual processing, and makes all control decisions. The DSP and the frame grabber are linked by a VMEbus. The camera generates the image at a resolution of roughly 350×250 pixels, but converts the image to a standard RS-170 analog video signal. The frame grabber then samples and digitizes the analog video signal at a resolution of 512×512 pixels. Since this is more data than the DSP needs or can possible process, the DSP subsamples the image at a resolution of 64×48 pixels. Internal limitations of the frame grabber and VMEbus limit the frame rate to 15Hz. The camera is fixed in place so the robot cannot look around corners without pausing to turn its entire body.

With the exceptions described below, the 6811 microcontroller, referred to as the FEP (Front End Processor), is used only to interface the DSP to its peripherals. The DSP communicates with the 6811 over a built-in high-speed serial port. The 6811 does not have hardware to speak the DSP's serial protocol, so the 6811 implements the serial protocol in software. The DSP controls the base and other peripherals by sending a byte stream to the 6811 over the serial line. The 6811 forwards the bytes to the peripherals. Escape codes in the byte stream allow the DSP to switch from one peripheral to another. Other escape codes cause the 6811 to transmit the status of the front panel switches and the base's rotational odometer to the DSP.

The FEP only does two intelligent things. First, it parses the status output of the base's serial port to find reports of the rotate motor's current position. It converts these reports from hexadecimal to binary and sends them to the DSP upon request. This was done because the DSP only has one DMA controller and so cannot read and write its link to the FEP without taking an interrupt for every byte. To have the FEP forward every character written by the base to the DSP would have slowed the DSP down. It also would have required more complicated device drivers.

The other non-trivial task of the FEP is the polling of the bump switches. The FEP constantly polls the bump switches to check whether the robot has hit something. When the robot hits something, the FEP stops its normal activities and sends a sequence of commands to the base to reverse direction and turn away from the bumper that was hit. When the bumper deactivates, the FEP returns to its normal activity.

The program running on the DSP is approximately 1500 lines of Scheme code, plus drivers and data. The load image for the DSP is under 7K words (28K bytes). The program uses 5 image buffers, each of which is 3K words in size (64×48 pixels), so the total scratch space is less than 20K words (80K bytes).

2.6 Other visually guided mobile robots

A number of mobile robots have been built which use vision for navigation. Early systems tended to use vision for all navigation, but more recent systems have often used sonar for obstacle avoidance, and have relied on vision only for localizing themselves within a map, or for detecting a road or other path.

2.6.1 Systems with geometric maps

Many of these systems used vision to build up detailed geometric maps of the environments. The earliest of these was certainly Shakey, see Nilsson [95]. Early visions of Shakey used vision to construct grid-based maps of its environment, which was restricted to contain matte polyhedra.

The Stanford Cart (see Moravec [92]), used a novel nine-eyed stereo algorithm to build and maintain maps of the locations of feature points in its immediate surroundings. It then used the maps to plan safe paths around the features. The system worked surprisingly well, but unfortunately, it required 15 minutes per frame to perform its computations because of the limited computing power available. It therefore had serious problems with environmental dynamics such as shadows which moved with the sun.

Thorpe's FIDO system [125] used stereo vision to build grid-based local models of its environment. The robot was then able use the model to plan and execute a safe path along a sidewalk.

Kriegman, Triendl, and Binford's Mobi system recognized hallways, walls, and doors in a two dimensional world-model [72][73]. The world model was built as the robot moved through its environment. The robot used 1D stereo to find the positions of strong vertical edges in the environment and tracked them over time using a Kalman filter. The system was able to achieve a cycle time on the order of ten seconds per frame.

Ayache and Faugeras [9], and later Kosaka and Kak [71] developed systems which used Kalman filtering with full 2D stereo to track the positions of vertical edges in a map and localize the robot's position. Kosaka and Kak were able to reasonable speeds using only a standard 16MIP workstation. Part of the efficiency of their system is due to the fact that it searches for edges only where its model says they should be. While this gives their system a large performance boost, it also means that it must rely on sonar for obstacle avoidance.

Braunegg's MARVEL system [23] used stereo data to build and maintain grid-based representations of its environment and determine its location by matching sensor data to the model. The most interesting thing about the system was its ability to tolerate slow changes in the environment by gradually mutating its model.

2.6.2 Non-geometric systems

Kortenkamp *et al.* describe a system for navigation using vision to match landmarks described by low resolution stereograms [70]. The system detects landmarks, which

are corridor intersections and doorways, using a sonar-based algorithm, and then distinguishes them using vision. The system navigates using the sort of topological map popularized by Kuipers and Byun [75] and Mataric [88].

Nelson [93] describes a novel system for navigation using a visual associative memory. The memory is trained to associate proper actions with overhead views from a planar world. The system can then return to a home position by repeatedly looking up the proper action in the memory and executing it.

Engelson and McDermott describe a system for recognizing places using easily computed image signatures, rather than complicated models [43]. The signatures are convenient hash functions which are sufficient to distinguish views of different places.

A number of purely reactive systems have been developed. Bellutta *et al.* developed a corridor follower which estimates the position of the vanishing point in the image and servos on it to align with the corridor [19]. Storjohan *et al.* [119] developed a simple stereo-based system for obstacle avoidance based on back-projection of images. The system is given the correct correspondence between pixels in the two images for the case where both cameras are viewing a floor with no objects on it. The system warps the left image onto the right image, and compares them. If the space in front of the robot is unoccupied, them the right image and the warped left image should be identical. If they are not, then the offending pixels must contain obstacles. Coombs and Roberts have described a system which uses the presence of optic flow in the image to avoid obstacles [33]. Horswill built systems for following moving objects and following corridors using monocular vision and knowledge of the appearance of the background [65].

2.6.3 Outdoor road following

A large segment of the work in visual navigation is devoted to the problem of outdoor road following. A great deal of early road following work involved the construction of explicit 3D models of the road. A large amount of work has been done on recovering the three dimensional structure of visible road fragments from a single monocular view. These systems use the extracted contours of the road, together with some set of a priori constraints of road shape to recover the shape of the road. Waxman, LeMoingne, Davis, Liang, and Siddalingaiah describe a system for reconstructing road geometry by computing vanishing points of edge segments [136]. Turk, Morgenthaler, Gremban, and Marra used a simpler system based on the assumption that the vehicle and the visible portion of the road rested on the same flat plane [130]. This was called the *flat-earth* model. The flat-earth model allowed much faster processing and was sufficient for simple roads. Later, they substituted a the constraint that the road have constant width for the constraint that the road lie in a plane. This allowed the system to handle roads which rose and fell and so it was termed the *hill and dale* model. Because the hill and dale model cannot account for curved roads, DeMenthon proposed an algorithm based on the *zero-bank* constraint which allows both hills and turns but does no allow the road to bank (turn about its own axis) [38].

Much of the work in outdoor road following has been done at Carnegie Mellon. Early work at CMU was done by Wallace, et. al. [135][134][133]. They use a sensor-

based algorithm driven in image coordinates for motor control. They take the approach of extracting road edges rather than segmenting the road and using the slicing technique however. They also make extensive use of color information [133]. Using the CMU Warp parallel processor, a 10-100 MIPS floating-point processor optimized for low-level vision, they have reported speeds up to 1.08 km/hour using a servo-loop time of one frame every three seconds. More recently, Crisman [35] implemented a color-based road tracker which was able to properly distinguish road pixels from other pixels, even in the presence of complicated shadows. Pomerleau [100] has described a neural network which efficiently learns to follow roads using low resolution images.

Arkin has reported an architecture based on the schema concept [8]. A schema is a description of what action is appropriate to a given situation, similar to my notion of a tactical routine. Arkin defined a number of motor-schemas for moving along a path and avoiding obstacles which, when run concurrently, were able to navigate about the UMass campus.

Dickmanns [39] has described a number of road following systems which can drive on the autobahn at speeds of up to 100km/hour. The systems use Kalman filtering to efficiently search for road edges within small windows of the image. By using multiple processors, his system is able to process images at 25 frames per second.

Chapter 3

Lightweight vision

In this chapter I will argue that vision can be very cheap and suggest some general techniques for simplifying visual processing. This is not an argument that all visual tasks can be solved cheaply. My goal is simply to convince the reader, particularly the reader who is not a vision researcher, that cheap real-time vision systems are feasible for a variety of tasks. This is, therefore, a theory of how to build task-specific vision systems as cheaply as possible, not a theory of how the human vision system works or of how to build a general programmable vision system. Nevertheless, the issues raised in building task specific vision systems overlap a great deal with recent discussions on the nature of general vision systems.

3.1 Background

Much of the previous work in vision has focused on the construction of modules of a hypothesized general vision system. Early work viewed vision as a domain-independent system for creating monolithic models of the outside world. Proposals vary as to the nature of the models, and the processing performed to create them, but the general approach is to use a series of transformations of the sensory input to move from what I will call the "surface structure" of the input to its "deep structure." I borrow the terms from Chomsky [31], who used them to refer to different levels of structure within sentences. I will use them as relative terms. For example, one might assume that a vision system takes images, transforms them into edge maps, then into depth maps via stereo matching, and finally into collections of geometric descriptions of individual objects. One often thinks of the objects as being "what's really out there," and of the image as being only a shadow; that the objects are the true reality and the images are mere appearance. Thus the object descriptions capture "what's really out there" better than the images, and so we can think of the object descriptions as the "deep structure" which is buried beneath the surface of the images.

In vision, the deep structure which people have traditionally tried to extract is the detailed geometry of the environment (see Aloimonos and Rosenfeld [7] for a historical survey, or Marr [87] or Feldman [45] for examples of specific proposals). This is sometimes referred to as the "reconstruction approach." (see Aloimonos and

Rosenfeld [7]). The approach has a number of appealing features. First, a complete description of the geometry of the environment, suitably annotated with other surface information such as color, seems to be a convenient form from which to compute any information about the environment which may be required. Another appealing characteristic is that is it fully domain independent, in the sense that any information needed about the environment can be derived from the model, provided the model is sufficiently detailed.

3.1.1 Problems with reconstruction

There are a number of problems with the reconstruction model of vision. The criticism that is the easiest to make and the most difficult to assess, is that reconstruction is hard to do from an engineering standpoint. Formally, it is an "ill-posed" inverse problem (Poggio and Torre [99]): the projection process loses information, and so inverting it requires additional assumptions such as smoothness of surfaces. These assumptions are enforced by requiring the spatial derivatives of the image or depth map to be small. This usually reduces the reconstruction problem to some form of constrained optimization in a high-dimensional space (see Poggio and Torre [99] or Horn [60] for discussions of general techniques for optimization in reconstruction). The resulting equations are frequently non-linear and so tend to be unstable (see Aloimonos and Rosenfeld [7]). Instability means that estimates of surface structure are highly sensitive to noise in image measurements. Estimates are also highly sensitive to deviations from the idealized smooth surface. Image discontinuities are a particular problem since discontinuities are the ultimate non-smoothness (see again Aloimonos and Rosenfeld, *ibid.*). To date, the reconstruction model of vision has never been fully instantiated, although many individual modules have been built with varying degrees of success. This does not prove that it cannot be done effectively and efficiently, however.

A deeper criticism of reconstruction is that it is, at best, a partial theory. As an agent goes about its life, it will need to assess different aspects of its immediate situation. This is rather like answering a series of questions: *Is something about to attack me? What direction is the hallway? Are there any free seats here?* These questions will vary from moment to moment and from situation to situation. The agent will never need to answer the question *what is the monolithic model of the current state of the environment?* other than to use that model to answer some other question about its immediate situation. The important questions are always task-dependent and quite specific. Any agent that can answer the right questions at the right times will be successful, regardless of the organization of its perceptual system.

One of the major attractions of the reconstruction theory was that it gave us a domain-independent theory of visual processing. However, the information an agent needs from the environment is always domain-dependent. Thus reconstruction cannot be a complete theory of vision, but rather a claim that the processing of domain-dependent queries should begin by transforming sensor data into a canonical form: a single domain-independent representation which allows domain-dependent queries to be processed efficiently. The truth of this claim rests on three engineering issues:

1. whether the use of a canonical form really does simplify query processing,

2. whether the canonical form that makes processing easiest is a surface reconstruction, and

3. whether the savings accrued by using the canonical form are sufficient to justify the processing required to build the canonical form.

These issues are hard to evaluate. It is likely that they will have to be answered empirically. There are reasons to be skeptical that building a canonical form will be the simplest solution, however. General experience with representation indicates that expressiveness and explicitness trade off with one another in representation languages (see Levesque and Brachman [81] for a general discussion of this phenomenon in the context of knowledge representation languages). The first thing which is taught in many AI classes is the idea that a good representation should make the information important to a task explicit, and nothing else. If this is to be taken seriously, then a single canonical representation for all tasks is unlikely.

If it is hard to choose a single representation for all processing, it is equally hard to choose a single algorithm for constructing the representation. For any given problem, such as edge detection, there will tend to be many different algorithms, each of which makes different performance trade offs. One edge detector may be optimized to localize an edge as well as possible, at the cost of speed and accuracy in detecting the edge in the first place. Another edge detector may do the opposite. Again, it is unlikely that there will be a single approach which will give the right performance for all possible high level tasks.

All these problems can be answered by diluting reconstructionism. Indeed, few people ever believed the reconstruction theory in its most absolute form. It is almost certainly the case that some sort of shared intermediate processing is needed to build a system for answering a wide range of queries (see section 3.2.2, below). Such intermediate processing need not involve building a single monolithic representation, nor need it involve a fixed set of processes.

As a final caveat, it should be noted that nothing I have said here applies if the task facing the designer is *to do reconstruction*; if one is trying to build digital terrain maps from satellite imagery, then one must, by definition, do reconstruction.

3.1.2 Active vision and task-based vision

Over the years, a number of researchers have focused on how agents can use both the structure of a task and their own activity to simplify visual processing. Gibson's theory of direct perception [54][53] was based on the notion that the an agent's physiology was designed by nature to "resonate" with the particular information inherent in its environment. Cutting has further developed Gibson's theory of direct perception to take into account the agent's role of choosing which parts of the stimulus to attend to ("directed perception," see Cutting [36]).

Recently, a great number of machine vision researchers researchers have proposed various approaches to "active" or "task-based" vision. Bajczy [10] emphasized the

use of active sensing strategies such as active focusing and zooming. In my master's thesis [65], I argued for basing the design of vision systems on concrete tasks. This section is an elaboration of that work. Ballard [11] and the vision group at Rochester University have developed an approach which they call "animate vision." They give many examples of how gaze control, agent motion, and so on can be used to simplify processing. Ikeuchi [66] has proposed an approach he calls "task oriented vision." Ikeuchi is in the process of developing a "vision compiler" which can generate custom vision applications given descriptions of visual tasks. Finally, Aloimonos [6] has developed an approach he calls "purposive and qualitative active vision," which stresses many of these same themes.

These proposals overlap a great deal. A common feature is the notion that visual machinery can be greatly simplified by computing only the information needed by the agent to perform its immediate task(s). Another common feature of these new proposals is the explicit use of agent dynamics to simplify visual processing. The earliest use of dynamics that I know of is Gibson's *The Senses Considered as Perceptual Systems* [53]. The earliest explicit use in the machine vision community of which I am aware is Bandopadhay *et al.* [13] who used active tracking of a feature point to simplify the equations for structure-from-motion. Perhaps the most systematic use of dynamics is in Dickmanns' Dynamic Vision methodology [39], in which vision is viewed as an optimal estimation problem in space-time. Dickmanns uses techniques, such as Kalman filtering, from control theory and optimal estimation theory to recover metric scene parameters.

The final common feature is a tendency to use qualitative information rather than metric information when possible. Qualitative information often has the twin advantages of being easier to extract from images and of being more numerically stable than metric information. The earliest proposal for the use of qualitative information of which I am aware is Thompson and Kearney [123].

3.2 Building lightweight vision systems

Often, we want to build the simplest possible system which can solve a given task in a given environment. In general, a given (task, environment) pair can be solved by a broad range of mechanisms with a broad range of cost and performance parameters. The better the designer's understanding of this range of mechanisms, the better she is able to choose the mechanism that best suits her needs. Anything which can broaden the range of mechanisms which can solve the problem is therefore a potential resource for the designer.

3.2.1 Resources for simplifying vision

If an agent need only compute the information it needs to perform its task, then we can treat that task as a resource which can be used to simplify visual processing. By "task" here, I mean the high level task of the agent. The task specifies, albeit loosely and indirectly, what information is required to perform it. It specifies it only

loosely and indirectly because a task such as obstacle avoidance does not require any particular representation of space, obstacles, or the distance to obstacles, but rather places a set of operational constraints on which representations are sufficient for the task. One might use a depth map, or a Cartesian elevation map, or simply the distance and direction of the nearest obstacle. The units of measurement might be feet, meters, or some other unknown, but repeatable, system. This looseness should be viewed as a positive feature, not as an absence of clarity which needs to be removed by further task specification. The fact that the task underdetermines the choice of representation means that the designer has a wide range of choices available, one of which may be very easy to implement.

The task also provides the designer with concrete performance constraints. Again, these constraints are a good thing. Performance constraints not only tell the designer what is required, but also what is *not required.* Without concrete performance constraints, the designer can never know when to stop: since the system will never have perfect performance, there will be room for endless improvement, but this additional performance may be to no good end. More importantly, there is no single aspect of a system which corresponds to performance. Performance has many parameters. These parameters trade off with one another and so cannot be optimized simultaneously. The performance constraints of a task allow the designer to make intelligent choices about which aspects of performance to optimize and which to sacrifice.

One particularly important performance trade off is the relation between spatial sampling rate and temporal sampling rate. For a fixed amount of computational power, the designer cannot increase one without decreasing the other. Since the temporal sampling rate places a lower bound on response time, this can be a critical trade off. Many researchers have traded temporal resolution for spatial resolution. Increasing spatial resolution can increase geometric accuracy, and so (one hopes) increase the general reliability of the agent. Unfortunately, this can lead to a vicious circle. Doubling the linear spatial resolution requires reducing the temporal sampling rate by at least a factor of four. If the agent makes control decisions at the same rate that it gets perceptual data, then the bandwidth of the control system will also have to drop by a factor of four. The agent will have to wait four times as long to find out if it made a mistake. If mistakes need to be corrected promptly, then the agent will also have less time in which to recover, and so mistakes become a much graver issue. Usually, the solution to this problem is to think very hard about a control decision before making it. But that means that the agent needs very accurate and detailed perceptual data about its environment, which means even more resolution, and so on.

High spatial resolution is needed for good geometric accuracy[1], but geometric accuracy is not always needed to perform the task. All the processing in Polly is performed on 64×48 images (or 16×12 for matching images of landmarks). Using these images, it would be difficult to estimate the robot's position relative to the walls with any precision. Fortunately, Polly's navigation task does not require it to know its exact position relative to the walls, only (1) whether it is moving roughly parallel

[1] It may also be needed in other cases such as when the only features in the scene are very very small.

to the walls and (2) whether it is too close to one of them (see chapter 8 for more discussion). Both of these can be checked efficiently using low resolution images.

The correlations between the deep structure of objects in the world and the surface structure of the image are another useful resource for simplifying vision. The orientation of a corridor is a property of its geometric structure, and so one way of finding it would be to determine the geometric structure of the corridor and then to extract the orientation from the geometric model. The geometric structure of the 3D world is not reflected in any simple manner in the structure of individual images. In the case of the corridor, however, its orientation happens to coincide with the location of the vanishing point of the corridor edges in the image. The vanishing point is a purely 2D notion which is only present in the (surface) image structure, but in this case, it happens to be correlated with the (deep) 3D structure of the corridor. Since the vanishing point is easily computed from the image, we can easily compute the orientation of the corridor.

The intended environment of the agent is another important resource for the designer. To survive in a wide range of environments, an agent has to be prepared for the worst case of the worst cases of all the environments. This means tighter performance constraints and larger information requirements. By restricting the range of environments in which the agent is to operate, we can often relax these constraints and use simpler systems. Surface structure correlations often hold only in restricted classes of environments. So, again, restricting the range of environments for an agent can allow optimizations which would not otherwise be possible (see chapter 4 for more discussion).

3.2.2 Distributed representation and mediation

One of the principle features of the recent work in active and task-based vision, and in many parts of AI in general, has been the move away from central mediating representations, such as monolithic models, toward more distributed representations in which different parts of the problem are solved by different computations which are largely independent (see Horswill and Brooks [64], Aloimonos [6], and Brooks[24]). Often these different computations are performed in parallel. One of the fears surrounding this practice is that a complicated agent which performs a wide range of tasks will need huge numbers of parallel processes. The issue of parallelism is outside the scope of this work, but the issue of distributed presentation is very important here.

Distributed representation and central mediating representations have complementary strengths. One advantage of using separate computations and representations to perform separate tasks is that it relaxes design constraints. A subsystem which computes two pieces of information may have to satisfy the agent's performance constraints (speed, accuracy, and so on) for both pieces of information simultaneously. Thus, solving the problems separately may be much easier than solving them simultaneously. On the other hand, many tasks may have similar performance constraints and may perform the same intermediate steps. In that case, there is no reason not to share those intermediate steps. The use of mediation thus depends on the mix of

tasks which the agent needs to perform, and the compatibility of their performance constraints. I have tended to build independent systems for extracting different pieces of information, and then fold together similar computations afterward.

Sometimes, it is useful to use multiple computations to perform a single task. If the performance constraints of the task are too tight for any particular computation, then a set of independent systems that make different performance trade offs can accomplish the task reliably. For example, one system can give an approximate answer immediately, while another gives a more accurate answer later. Also, two systems with independent failure modes can be used together to perform a task reliably. Polly can align itself with the corridor both by finding the vanishing point of the corridor and by turning so as to balance the distance to each wall. If the vanishing point computation fails, the wall balancing algorithm compensates until the vanishing point computation recovers.

3.3 Other related work

In recent years, a number of active and task-based vision systems have been built. A common task is the tracking of moving objects. Coombs [26] implemented a system for fixating and tracking objects using a stereo eye/head system. The system used disparity filtering to localize the object. Woodfill and Zabih [140] used optical flow to segment and track a moving object using monocular input. Horswill [65] used environmental constraints to segment and track a moving object. Aloimonos [6] describes a system, Medusa, which performs a number of tasks, including tracking.

Other researchers have built vision systems designed to extract useful high-level information without building detailed depth maps. Aloimonos [6] describes a number of motion algorithms which recover information useful for navigation without having to solve the general structure-from-motion problem. Nishihara [96] describes a minimalist approach to early vision, particularly stereo. Swain [121] describes a system for recognizing colored objects without using any geometric information whatsoever. Various researchers have developed systems for directly detecting occluding contours, without first measuring depth (see Mutch *et al.* [124], Spoerri [117], Toh and Forrest [128], and Wixson [139]). Horswill [63] used disparity filtering to implement proximity detection from stereo images. Aloimonos' Medusa system extracts a number of useful pieces of high-level information such as the presence of moving objects, the direction of translation, and looming [6].

Tsotsos has argued that task-based recognition is simpler than bottom-up recognition from a formal complexity standpoint [129]. Unfortunately, the proof is quite specific to the particular formalizations of the task-based and bottom-up problems. The former amounts to template matching when image position is not given and occlusion is not present, the latter to template matching when when the pose is given and occlusion is allowed. In the proof, it is the presence of occlusion which drives the complexity of the bottom-up approach. The results also depend on the requirement that the templates must match using both a correlation metric and a difference metric. If only one metric is used then, again, the complexity result falls.

One of the consequences of abandoning the model of vision as builder of monolithic models is that resource limitations are introduced into sensing. A robot can only point its camera in one direction at a time, the stereo system can only fixate one depth plane at a time, the color histogram unit can only process one task at a time, and so on. These computational and physical resources must be allocated and reallocated continually by the agent. The result is that sensing becomes a series of actions and so the traditional problems of action–planning, reaction, execution monitoring, management of multiple conflicting goals—are recapitulated for perception. This problem is generally referred to as the problem of the control of selective perception. It has recently spawned a vast literature (see the collections [37, section XI] and [113]). Approaches range from decision-theoretic models (see Rimey [127]) to full-blown symbolic planning (see Pryor and Collins [101]).

Visual routines are a popular framework for discussing these problems. The notion of visual routines is due to Ullman [131], who proposed the idea of a visual routine processor (VRP), which acts as a co-processor for higher levels of processing, and which has an instruction set of visual operations. To Ullman, visual routines were patterns of processing implemented by sequences of operations in the same manner as subroutines are implemented series of normal CPU instructions. Agre and Chapman modified and popularized the model within the greater AI community [4]. Agre and Chapman implemented a simulated VRP which they used for high-level vision. The control inputs of the VRP were tied directly to the central system and were treated in roughly the same way as any other effector. For Agre and Chapman, visual routines were not routines in the sense of "subroutines," but in Agre's sense of common patterns of activity [3]. Later, Chapman implemented a second simulated VRP for his system, Sonja [30]. Chapman proposed this system, the Sonja Intermediate Vision System or SIVS, as a provisional computational model of biological intermediate vision [28].

Reece and Shafer implemented their own simulated VRP which they used to model active vision for driving [103]. Since then, the use of the term has broadened considerably, in effect, to include any sort of co-processor architecture. Swain [122] discusses how a number of existing active vision systems can be thought of as "active visual routines," but in this case, there is no visual routine processor!

3.4 Summary

Very general vision problems, such as building geometric models of arbitray environments, can be very difficult and expensive to build. However, we can often build very simple and effective vision systems for specific tasks and environments. The structure of the task and the environment are important resources for designing these simple vision systems. As task and environment become more specific, the requirements on the vision system become looser, and so the space of possible mechanisms which can perform the task grows. Often, this space is very broad, and so some mechanism in the space is often very efficient. By making clear what design constraints do and do not hold, we can greatly simplify the design of our vision systems.

I have no general prescription for designing such systems, I have tried to outline the general resources available to the designer for simplifying vision problems. In the following chapters, I will give a number of examples of lightweight vision systems and discuss how they can be understood. With the proper analysis, the insights gained from the design of one system can be extracted and applied to the design of other systems.

Chapter 4

Analysis of specialization

4.1 Introduction

In the last decade, AI has seen a surprising number of negative theoretical results. Formal planning has been shown to be computationally intractable, or even undecidable (see Chapman [27]). Many perception problems have been shown to be highly numerically unstable (see Aloimonos and Rosenfeld [7] for a discussion of the stability of various subproblems of vision). In general, most AI problems amount to search of large spaces, be they discrete spaces, as in the case of reasoning, or continuous spaces, as in the problems of inverse optics. Unfortunately, search is fundamentally hard. It is not presently known how to solve search problems which are at least as hard as boolean satisfiability in polynomial time. There are deep theoretical reasons to suspect that it may be impossible (see Hopcroft and Ullman [59] for an introduction).

If traditional AI problems amount to search, but search is often formally intractable, then either AI is impossible, as Penrose argues [98], or there must be a loophole somewhere. There are two candidate loopholes. One is that there is actually a loophole in search. In this case, there must be polynomial-time algorithms for solving average search problems in the average case. This might be true if, for example, there was a good way of generating admissible heuristics for A^* in an automated manner. The other possibility is that there is a loophole "in life." In this case, the issue is not so much our algorithms as our problem formalizations—we've simply formalized harder versions of problems than people actually solve in their everyday lives. (There are also good reasons to suspect the very notion of treating life as a series of formal problems (see Agre [3]). I will ignore these for the moment and try to cast all the positions in terms of the framework of formal problems. This, inevitably, does violence to some of those positions, particularly Agre's, but I feel that it is worthwhile for reasons of clarity.)

This loophole-in-life strategy deserves some elaboration. It does not mean that humans never solve hard problems. People certainly prove theorems. The issue is rather what fraction of human activity is fundamentally hard, and what fraction is fundamentally easy. Clearly theorem proving is hard, but one hopes that we do not have to prove theorems to walk. The computations performed by a walking system are straight-forward from a complexity-theoretic standpoint, even though they may

be very difficult to prove correct (see Raibert *et al.* [44]). To put it glibly, walking is hard to understand, but easy to do. Theorem proving on the other hand has a relatively straightforward formal definition, and it is easy to see that systematic search will eventually find the answer if there is one. Doing the systematic search however requires a great deal of computation, and so theorem proving is easy to understand, but hard to do. Thus walking is easier to do than theorem proving, even if research on walking is not. (Research on walking may *be* theorem proving!)

Walking is a fairly clear case, others are less obvious. For example, people clearly plan, at least in some sense. However, if we formalize planning to be reasoning about the effects of actions so as to determine a fixed sequence of actions which will bring the world from one state of affairs to another, then it is not hard to see how we can reduce the problem to theorem proving, which is undecidable in the general case. Fortunately, everyone agrees that people are not commonly faced with the general case. As we start to restrict the possible problem instances—constraining the types of actions and effects that we may face or the types of goals we may have—then the problem gradually becomes more tractable. STRIPS planning, for example, is merely PSPACE-complete (Chapman [27]). There are two problems with this however. The assumptions embedded in STRIPS planning are not necessarily a good match with real environments. Also, PSPACE-completeness is still bad enough to make a polynomial-time algorithm unlikely.

The key question then is: what is it about the problem instances that we *do* encounter regularly which makes them easy to solve? In other words, what is the loophole? There are a number of possible overlapping responses to this. One, which has been most strongly advocated by Agre [3], is the idea that most of the problem instances we encounter are sufficiently similar that, whatever their intrinsic difficulty, a single solution can be adapted to solve a large variety of them, hence Agre's notion of "routine activity." Another, which I will consider here in detail, is the idea that the environment is more benign than one might suspect.

A number of authors have stressed the importance of environmental dynamics in explaining intelligent activity. Some authors stress the fact that the world is much simpler than the most general imaginable case (see, for example, Agre [3]). Other authors actively stress the complexity of the environment, arguing that the complexity of the environment allows agents to be simpler (see, for example, Simon [115]). These are not incompatible statements. The former is a statement about which of the imaginable possible worlds an agent must actually live in, whereas the latter is a statement about the presence of simplifying structures within the environment. Both groups agree that there are simplifying structures within the environment which allow agents to be simpler than would otherwise be necessary.

4.1.1 Computational minimalism

These observations have led a number of researchers to search for the simplest possible mechanism which can perform a given task in a given environment or class of environments (see, for example, Agre [3], Brooks [24], Chapman [29], Connell [32], and Rosenschein and Kaelbling [108]). This can be used as a methodology for engi-

neering, as with Connell's minimalist robotics, or a methodology for psychology, as with Agre's principle of machinery parsimony. In this work, I will mostly be concerned with engineering. In both cases however, one searches for the simplest class of mental machinery that is compatible with the desired class of tasks or interactions.

The advantage of the minimalist approach is that small changes in the task or the environment can greatly simplify the machinery needed to solve it. The disadvantage is that the same sensitivity to task and environment can make the minimalist systems very hard to build and understand. A system which works in one environment may fail to work in an environment which seems to be only slightly different. Worse, the effort expended in building a system for one environment may tell us nothing about how to build a system to perform the same task in a different environment.

Minimizing mechanism maximizes specialization. If computational minimalism is to succeed, then we need a set of theoretical tools for analyzing the specialization of systems to their environments.

4.2 Preview

We can understand specialization by analyzing it, in the sense of dissecting it, into discrete chunks which can be understood in isolation. We can perform the analysis by treating the specialized system as a transformed, or optimized, version of a general system which performs the same task. Each transformation required to optimize the general system into the specialized system will be dependent on a specific property of the agent's environment. I will call these *computational* properties of the environment, not because the environment is doing computation, but because they have computational ramifications. The result of the analysis is then a series of independent optimizations, each of which simplifies a specific computational subproblem using a specific property of the environment.

To take a simplistic example, animals often need to detect the presence of other animals in the environment. If the (non-living) objects in the animal's environment are all static then motion in the environment is a cue to the presence of another animal and so two animals that are identical except that one detects other animals using shape, color, and smell information and the other simply by treating any motion as another animal, may behave identically within that environment and thus be "behaviorally equivalent" given that environment. There are differences however. The animal which uses motion may be more "economical" in some sense. If it were a robot, it might be cheaper to build, or take less power or physical space. The other animal has the advantage that it can potentially operate in environments which violate the constraint that inanimate objects never move.

We can view the substitution of a motion-based animal detector for a shape-based one as a transformation of one mechanism into another, a transformation which preserves behavior in exactly the same way that rules of logical inference preserve truth value, substitution of identities in mathematics preserves denotation, or compiler optimizations preserve the input/output behavior of the program being compiled. An important difference however is that the transformation only preserves behavior in

environments which satisfy the constraint. We can view the constraint as representing a possible structure of environments, namely that inanimate objects do not move, and the transformation as the structure's computational significance[1]. We can characterize environments and compare them by giving the sets of useful constraints that they satisfy. We can characterize and compare agents by giving the sets of constraints that they require. The set of constraints which are assumed by an agent define the set of environments in which the agent can operate, that is, its "habitat," and for this reason I will refer to these constraints as "habitat constraints."

Application of a series of transformations produces a series of behaviorally equivalent systems which make different engineering trade-offs and different assumptions about the world. We can then compare behaviorally equivalent systems by giving a series of transformations by which one can be converted to the other. I will call such a series a "derivation" of the one from the other. A particularly useful way to analyze a specialized system S is to give a derivation of the system from a more general system G. Giving such a derivation

- makes explicit what additional assumptions are made by S,
- makes explicit what role those assumptions play in S's normal functioning,
- makes it easier to predict S's performance in novel environments,
- can make clearer how to modify S to operate in different environments.

Perhaps more importantly, the steps of the derivation can be reused in the design or analysis of other systems. In addition, if we define suitable notions of equivalence for components of agents, then we can apply the analysis recursively to subsystems of S and G.

The general approach is as follows. We first define spaces of possible mechanisms and environments. We then decide what aspects of the behavior of a mechanism make it behaviorally equivalent to another mechanism. Different criteria will be required for different kinds of systems. Given these definitions, we can find transformations between mechanisms which preserve behavioral equivalence, either conditionally or unconditionally. If the transformation yields more efficient mechanisms, then we can view it as an optimization. Important classes of environments can then be described in terms of the constraints they satisfy and the optimizations which those constraints facilitate, and those optimizations can be used in the design of future systems.

I propose this approach as a way of understanding specialized systems and applying that understanding to the design of new systems, not as a foundation for automatic programming. In particular, it is not a theory of how to discover constraints and optimizations from first principles, but of how to draw them out from an existing system in a formal manner and use them for the design of new systems. Once constraints and optimizations have been drawn out they can be used as cookbook methods. It is possible that the application of the cookbook methods can be automated, but it is unlikely that we can automate the writing of the cookbook in the foreseeable future.

[1] Or part of its computational significance, if it allows many different useful transformations.

In this next section, I will give a number of informal examples of useful habitat constraints and discuss two of them in great detail. Then I will develop the transformational approach in more detail, and given an example of how it can be applied in the case of simple control systems. In subsequent chapters, I will give more elaborate applications of the approach.

4.3 Habitat Constraints

We want to understand and describe the kinds of environments in which an agent can operate. I will take an *environment* to be a concrete thing, a particular place where an agent can be placed, and a *habitat* to be the set of environments in which an agent can operate properly. I will describe an agent's habitat in terms of a set of distinct "habitat constraints" which it satisfies. A habitat constraint is a predicate on environments. Its extension is the set of environments which satisfy it. Any habitat (set of environments) which is contained in the extension of the constraint satisfies the constraint, and so we can also think of it as a constraint on habitats. In the following sections, I will give a number of informal examples of useful habitat constraints.

4.3.1 Computational properties of large scale space

The space in which we live could be laid out in an arbitrary manner. It could be maze-like, constantly shifting, or full of objects moving rapidly in unpredictable manners. Fortunately, it tends to be more benign in practice. Indeed, people actively structure their environments, both intentionally and unintentionally, so to as to make them easier to live in (see Agre [3] and Hammond and Converse [56] for computational discussions of this general phenomenon). They build roads, wear paths in the grass, build navigation beacons, establish social conventions, and exploit existing regularities such as natural paths and landmarks. This structuring is sufficiently important that there are entire subfields of architecture and urban planning devoted specifically to understanding how people understand or fail to understand how to navigate in their environments, so that better environments might be designed. (See Passini [97] for an overview. Lynch [84] is perhaps the seminal work in the field. See also Kuipers [74] for a computational model of large scale space that was partly inspired by the work of Lynch).

When we consider large scale spaces such as cities or office buildings, we can see that they have a number of useful properties which are not logically necessary. One such property is the existence of a set of roads which effectively cover the entire city. The set of paths forms a graph. Thus roads (or paths, more generally) allow us to reduce the initial choice of a route to the choice of a series of roads, which is in turn no worse computationally than the search of a relatively small finite graph.

The regulation of change is another important property. While the geometric configuration of the world is constantly changing — people and cars move around, stores relocate, trees blow in the wind — the large scale structures of the environment such as roads, buildings, and highways change slowly if at all. We can rely on the

fact that following a road will predictably bring us from one place to another, even if we cannot rely on the configuration of the obstacles (cars, motorcycles, pedestrians) within the road. In other words, the graph doesn't change quickly. Let's call the property that the space is largely covered by a network of paths (roads in this case), and that their structure is largely stable over time, the "stable path constraint." The stable path constraint may seem trivial, but keep in mind that it is not true of all large scale environments. Forests and mountain ranges do not necessarily have useful networks of paths. Deserts and oceans generally do not have paths at all. All these environments are regularly navigated however.

Of course, there are things which change quickly and therefore cannot be anticipated in advance. It would be foolish to plan a complete trajectory from my home to my office in advance, since it might coincide with the trajectory of a car or pedestrian, resulting in a collision. Fortunately, these problems are relatively benign. The objects which are not predictable in the long term, such as cars and pedestrians, are generally predictable in the short term — they rarely reverse direction, or start or stop without warning. They also tend to be small enough so that one can get around them fairly easily. So as a pedestrian, I may have to walk around an unanticipated parked car, but not an unanticipated skyscraper.

Finally, there are a great number of social conventions, such as traffic lights and road lanes, which make life still more predictable. With the exception of an occasional bicyclist, we do not have to worry about oncoming traffic on a one-way road.

4.3.2 Computational properties of kitchens

Consider the problem of cooking an omelette[2]. The domain has a number of useful properties. One such property is that the objects in the kitchen fall into natural categories such as egg, spatula, knife, and so on. Objects of a given category have a characteristic set of possible states (*e.g.* "dirty," "clean," "hot," "cooked"). Objects in the world act as containers of state information. The functional state of the world is determined by the states of all the objects and their spatial relations. The possible vectors of object states is a much simpler state space than the space of possible collections of ground-instances of predicates (a common state space in planning systems). A simpler state space means that an agent has less to represent in its head. If the space is simple enough and structured well enough, then we will be able to design agents which decide what to do without representing the state space in their heads. However, we as designers will want to understand the state space so that we can understand its structure and relate it to the structure of the agent.

The choice of actions can be simplified by when neither actions nor goals are sensitive to the logical identity of an object. It doesn't matter *which eggs* you use to make an omelette, or even if they are the same eggs you started with. A demon could permute all of the uncooked and unbroken white eggs in the kitchen and it would make no difference to the agent, because the eggs are functionally indistinguishable. We can call this property "the indiscernibility of individuals," following Leibniz. One of

[2]This section is based on joint work with Phil Agre (see Agre and Horswill [2]).

the advantages of the indiscernibility of individuals is that you need never backtrack over the choice of which spatula to use because it doesn't matter which spatula its uses so long as it's clean. There are cases where the indiscernibility of individuals is insufficient to guarantee that an agent need never backtrack over the choice of an object, but it is nevertheless a very useful constraint. Note that this is *functional* indiscernibility, not absolute indiscernibility. One often *can* distinguish spatulas, but usually the agent does not need to. Indiscernibility of individuals means that the agent does not need extra machinery to keep track of which spatula is which. It can individuate objects indexically (see Agre [3]).

Another useful constraint is that certain types of objects can always be easily reset into their "normal" states, and so the agent can freely perturb them from their normal states. Mixing bowls, for example, can be thought of as having two states, clean and dirty, and usually need to be in the clean state when one starts to use them. Fortunately, mixing bowls can be easily washed, so an agent doesn't need to worry about dirtying a mixing bowl. It will always be able to return the mixing bowl to the clean state without changing the states of any of the other objects[3]. In particular, the dirtying of the mixing bowl is always reversible, and so dirtying it can never make an otherwise achievable goal suddenly unachievable. This reversibility means that the agent not need to think about the implications of dirtying a mixing bowl. It can go ahead and dirty the bowl, and clean it later if need be. That, in turn, means that it leave the state of the bowl out of the functional state of the world entirely, so agent and designer have even less representation to worry about. (The agent may have to keep track of whether the bowl is free for a given task, but that is another matter which can be optimized separately).

Another property is that the states of most objects are perceptually available. You can tell if the bowl is dirty by looking at it. You may even be able to choose a clean bowl simply by performing a visual search. The visibility of state means that the agent may not have to keep track the the states of of objects in its head, thus saving scarce short term memory.

For an example of a system which uses the special properties of cooking domains to choose actions in real time without preplanning, see Agre and Horswill [2].

4.3.3 Computational properties of office environments

Like cities, office buildings have a great deal of structure which simplifies the process of getting around. They satisfy the stable path constraint. They also have a number of properties which simplify place recognition. As you approach a landmark such as a junction, its appearance on your retina will be determined by the lighting conditions, its position in space, and your position in space. Visual recognition is a hard problem because all these can vary in the general case. However, lighting is less variable in office buildings because it is largely internally generated: there is a fixed set of overhead lights which have a fixed brightness. Furthermore, the fact that you are

[3]Ignoring objects such as sponges and dish-rags, which are themselves resettable into their normal states of being clean and wrung out.

walking down the middle of a corridor means that you can only see the object from a very restricted range of angles. Thus little or no computational search over possible angles or lighting conditions is required. The only remaining parameter is distance from the object. By walking down the corridor, you are effectively performing a mechanical search over the set of possible distances.

These properties conspire to make the recognition of the junction a much easier problem than the fully general recognition problem. This has three implications. First, one can imagine using very simple mechanisms for solving that particular problem[4]. Second, while the simple mechanisms might not suffice for the general case, an agent might prefer to use them when possible, *even if a more general system were available.* The simpler system might be more robust within its domain of expertise or it might simply require fewer cognitive resources, thus freeing up other resources for use on other problems. Finally, it means that one should not test one's general place recognition systems only in corridors. I will return to these points in the conclusion.

Another useful property of office buildings is that the floor of a given building will often have a very predictable appearance. For example, many buildings are carpeted and many carpets are regularly textured or even not textured at all. The predictability of the texturing of the carpet means that any region of the image which isn't textured like the carpet is likely to be an object resting on the ground (or an object resting on an object resting on the ground).

Finally, office buildings have the useful property that they have flat floors and so objects resting on the floor which are farther away will appear higher up in the image, provided that the objects rest on the floor. This provides a very simple way of determining the rough depth of such an object. I will discuss these last two properties in detail below.

4.3.4 Background texture constraint

Separating objects from the background is a common problem in visual processing. It can be an arbitrarily difficult task if we consider pathological backgrounds such as houses of mirrors or unusual objects such as chameleons. On the other hand, it can be very easy if there is some visible property that distinguishes objects from the background. Texture is one such property. A particularly simple case is when the background has no texture at all, in which case the presence of texture is a cue to the presence of an object. This can be made precise by considering the texturing, or surface markings, of a surface patch to be a linear combination of oriented sine waves. If we assume that there is a wide range of these (spatial) frequencies which are absent from the background, but which are present in the markings of the objects, then we can test for the presence of this range of spatial frequencies. We will say for the moment that an environment satisfies the background texture constraint if there is such a range of frequencies. Let that range be $\Omega = (\omega_0, \omega_1)$. We will call Ω the "zero band" because it is a frequency band with zero energy in the background.

[4]In fact, I have had reasonable success with raw template matching.

Many surfaces such as carpets, painted walls, stucco, wood paneling, and wall-papered surfaces with fine patterns have such zero bands. Other surfaces such as checkerboard floors do not. The backgrounds of many office environments are comprised of large surfaces whose textures share common zero bands. These environments satisfy the constraint.

To see the computational value of this constraint, consider the projection of a surface patch on to the image plane. The simplest case is when the surface patch is a plane whose surface reflectance is a sine wave grating with orientation defined by the vector $\vec{\omega}$:

$$R(x,y) = \frac{1}{2}\left(\sin\frac{x}{\omega_x} + \sin\frac{y}{\omega_y}\right) + \frac{1}{2}$$

If this patch is viewed in the fronto-parallel plane at unit distance through a lens of unit focal length, with even illumination of unit intensity, then the image intensity $I(x,y)$ at a point (x,y) in the image will simply be $R(x,y)$. If we increase the distance or decrease the focal length by a factor c, then we will have that

$$I(x,y) = R(\frac{x}{c},\frac{y}{c}) = \frac{1}{2}\left(\sin\frac{x}{c\omega_x} + \sin\frac{y}{c\omega_y}\right) + \frac{1}{2}$$

Thus the projected image will still be a sine-wave grating but of frequency $c\vec{\omega}$. If we instead rotate the patch around the x axis by an angle θ and ignore perspective effects, then the effect is simply to shrink the image along the y axis:

$$I(x,y) = R(x, y\cos\theta) = \frac{1}{2}\left(\sin\frac{x}{\omega_x} + \sin\frac{y\cos\theta}{\omega_y}\right) + \frac{1}{2}$$

resulting in the appearance of yet another grating, this time of frequency $(\omega_x, \frac{\omega_y}{\cos\theta})$. Similarly, a rotation about the y axis will result in multiplying the apparent x component of $\vec{\omega}$. Finally, a rotation about an axis perpendicular to the image plane will simply rotate the frequency vector by the same amount.

Thus any simple grating viewed from any position will appear as a grating with identical amplitude but with a frequency vector modified by a scaling of its two components and possibly a rotation. Since the projection process is linear in that the projection of the superposition of two signals is the superposition of their projections, this rule will carry over to the surface patch's (two-dimensional) Fourier transform: the transform of the patch's projection will be the transform of the patch, rotated and then stretched or contracted along each axis (see figure 4-1).

The result of the foregoing analysis is that frequency bands of the surface patch are transformed into elliptical regions of the frequency domain of its projection. Thus if the background has a zero band, its projection should have a zero ellipse. In general, the distance, focal length and angles of the camera must be known to predict the ellipse.

An important special case where these parameters need not be known is when the texture of the background has no low frequency component other than DC, that is, other than a constant term. Then the zero band is simply the open interval

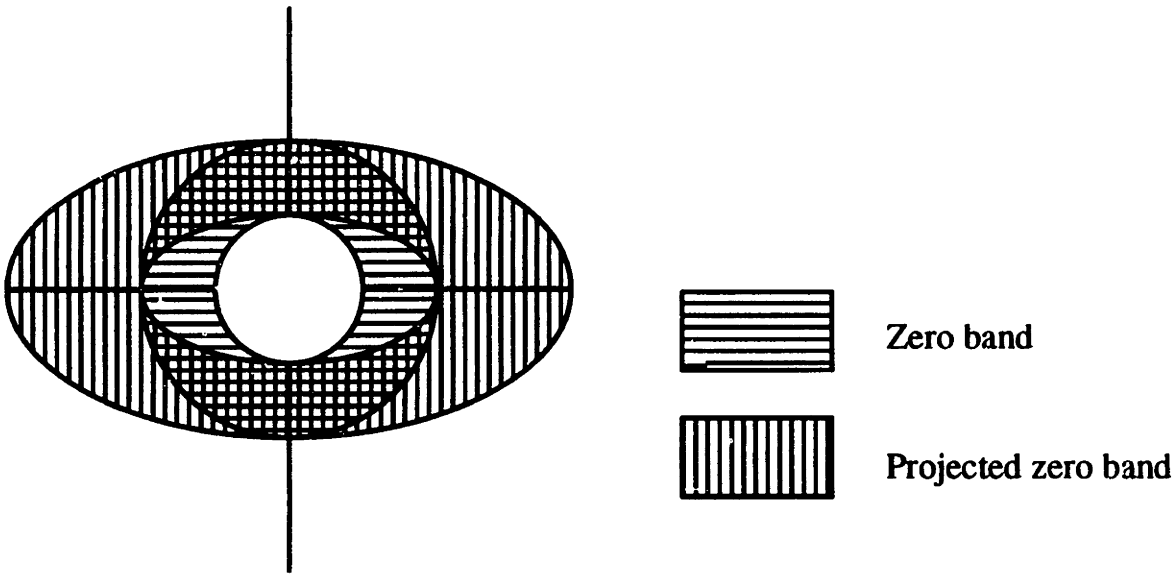


Figure 4-1: The effect of perspective projection on local frequency distributions.

$\Omega = (0, \omega_1)$. Suppose we view the texture from a distance greater than some d_{min} and bandpass filter the image. Since viewing the texture from an angle can only increase the apparent frequency of texture in the image, our bandpass filter should give zero energy whenever its cutoff frequency is below

$$\frac{d_{min}}{f}\omega_1$$

where f is the focal length of the camera. Thus if nothing closer than d_{min} focal lengths is in view, the image of the background will produce a zero response in the bandpass filter. A nonzero response indicates that there is an object at that point in the image.

This gives us a very simple solution to the figure/ground problem. We simply run any of a number of bandpass filters over the image and check for a response. It is an example of a habitat constraint making an otherwise "AI complete" problem trivial.

4.3.5 Ground plane constraint

Another common problem in vision is to determine the distance, or depth, to various objects or image points. This may be to adjust a parameter for an action, or to compare the depths of two objects, or to separate nearby objects from ones which are more distant. Depth recovery is often performed by computing a "depth map" that gives a distance from the camera for each pixel in the image. However, such computations are expensive and unreliable, at least at present (see for example, Horn [60] or Ballard and Brown [12] for introductions to standard depth perception techniques and the computational complexity involved. See also Barnard and Fischler [14] for a survey of stereo vision research).

In land environments the agent rests on, and travels about, some sort of ground.

Often the ground is a plane perpendicular to the force of gravity. When the camera is pointed forward, distant ground points project to points "higher up" in the image than nearer ground points. The correspondence between height and distance can be seen by observing that lines on the ground plane drawn perpendicular to the optic axis (that is, the line of sight) of the camera will project to horizontal lines in the image. The y (vertical) component of a point's projection in the image will be a monotonically increasing function of the point's distance from the robot (see section 5.4.2). If objects can be counted on to rest upon the ground plane, then their y coordinates can be used as a sieve to separate distant objects from nearer objects. We will call this the ground plane constraint: that all objects rest on a flat floor[5].

Built environments such as houses and office buildings generally satisfy the ground plane constraint. Like the background texture constraint, the ground plane constraint, when it holds, allows us to reduce a very difficult problem (depth recovery) to a very easy one (determining the y coordinate). This will be made more precise in section 5.4.2.

4.4 Specialization as optimization

We can view the computational significance of a habitat constraint as being that it allows an expensive system to be transformed, or "optimized", into a cheaper one. We can then describe a specialized system in terms of a series of optimizations of a more general system. Such a description makes the additional constraints assumed by the specialized system explicit and makes clear the computational significance of each one. The analysis can then be used as inspiration in the design of other systems.

Transformations over spaces of possible programs are common in computer science. Optimizing transforms are common in compiler technology, both at the level of intermediate code (see Aho, *et al.* [5] for a good overview), and at the source level (see Steele [118] for an extreme example). Dynamic programming can be viewed as an optimizing transformation over recursive functional routines which removes redundant computations (see "memoization" in Abelson, Sussman and Sussman [1], or the section on dynamic programming in Cormen, *et al.* [34]). Perhaps the concept closest to the one discussed in this paper is the notion of "partial evaluation," in which a program is transformed into one which is more efficient, provided that certain conditions specified to the partial evaluator are met. See Berlin and Weise [20] and Berlin [21] for examples relating to improving performance on scientific computation.

4.4.1 Transformations and optimizations

Suppose we have a space M of possible mechanisms. These mechanisms can be agents or components of agents. For example, M could be the space of mechanisms which compute mappings from images to real numbers. Some of these mechanisms roughly will "do the same things" and others will not. Whatever our notion of "doing the same

[5]This needs to be complicated somewhat. See section 5.4.2.

thing," we can define an equivalence relation, $\equiv$, over M, which holds exactly for pairs of mechanisms which do the same thing. Many pairs of mechanisms will technically be behaviorally distinct, but will nevertheless be indistinguishable within a given range of environments. If C is a habitat constraint (predicate over environments), then we will say that two mechanisms are conditionally equivalent given C, written $\equiv C$, if they behave equivalently in all environments satisfying C.

The transformations over these mechanisms are simply the functions from M to M. We are interested in the transformations which map mechanisms to equivalent mechanisms, and in particular, to mechanisms which are more efficient in some way. We will call these transformations "optimizations." Similarly, transformations which conditionally preserve equivalence will be called conditional optimizations. Most of our attention will be focused on conditional optimizations. Our goal in analyzing a specialized mechanism will be to find a series of conditional optimizations which transform a general mechanism into the mechanism in question. To formalize this process for any particular class of machinery, we must define its equivalence and conditional equivalence relations.

4.4.2 Derivations

Suppose we have a mechanism S which can be viewed as an optimized form of some mechanism G, and for which we can perform a series of transformations yielding intermediate results $S_1, S_2, \ldots, S_{n-1}$ such that

$$G \overset{C_1}{\equiv} S_1 \overset{C_2}{\equiv} S_2 \overset{C_3}{\equiv} \ldots \overset{C_{n-1}}{\equiv} S_{n-1} \overset{C_n}{\equiv} S$$

so that

$$G \overset{C_1 \cap \ldots \cap C_n}{\equiv} S$$

We can call this series of transformations a *derivation* of S from G. It should be noted that it is not strictly necessary that the transformations in a derivation generate mechanisms with strictly equivalent behavior. Often times, approximations of one form or another are more useful. The basic framework is easily extended to handle such cases.

Derivations are a useful way of presenting specialized mechanisms. They divide S's specialization into a set of distinct constraint/optimization pairs. In addition, the individual transformations generally deal with individual submechanisms or subproblems so the derivation puts assumptions in correspondence with the subproblems they solve. This correspondence (1) makes it clearer what roles the individual assumptions play in the correct operation of the mechanism, (2) help predict how the mechanism will behave in novel environments, and (3) can help the designer determine how to modify the mechanism to work in a novel environment. Often times, one can show in a single step that a large class of mechanisms are all equivalent and so may be freely interchanged (see the decalibration lemma below for an example), thus making clear what aspects of the design of the specialized mechanism are necessary to its proper functioning and what parts are arbitrary.

Derivations are also useful as correctness proofs, or rather, as *conditional* correctness proofs. It is often easier to prove the correctness of S by first proving the correctness of some G and then simplifying G down to S. The derivation shows that S is correct if G is correct and the specified constraints hold. Derivations can also be more edifying than direct correctness proofs of S in that they can give the reader a better sense of *why* the mechanism works properly.

Perhaps my arguments for the use of derivations are best summarized as follows. It is rare that someone within the research community builds a mechanism, specialized or otherwise, which works so well and is so intrinsically useful that people will simply want to use it directly. People are more concerned with how they can use the design of the mechanism to help them design new mechanisms. The best way to do this is to distill the specialization into a set of lemmas which show how similar problems can be reduced in similar ways. The lemmas are usually much more interesting that the source code for the system itself.

4.4.3 Synthesis versus post hoc analysis

This work is frequently taken to be a framework for synthesis of new mechanisms rather than analysis of old ones. Perhaps it is best taken as a framework for analyzing existing mechanisms *post hoc* with the goal of future synthesis. It says that however you build the first mechanism, you should reduce it to some recyclable lemmas for simplifying other mechanisms. While it may or may not provide inspiration for building the first mechanism, it does not itself give any rigorous methodology for building that first mechanism. It certainly is not a strong enough framework for doing automatic programming of the first mechanism. One might imagine automatic programming systems which use a stock of preexisting optimization lemmas to simplify designs. However, the task of doing the original design and analysis is no easier, or harder, than many other tasks in engineering or mathematics.

4.5 Analysis of first order, 1 degree of freedom control systems

Let us apply this framework to the case of a specific kind of feedback control system, the first order control systems with one degree of freedom. To do this, we need to specify what it means for two systems to be equivalent.

Suppose we have a first order system with a single degree of freedom x. By "first order" I mean that we have direct control of $\frac{dx}{dt}$. Suppose we want to adjust $\frac{dx}{dt}$ so as to cause x to have some desired value. If the system converges to the desired value in finite time, then we'll say that the system is *stable* for that desired value. We might consider two control systems to be equivalent if they are stable for the same sets of values.[6]

[6]This equivalence condition is not a necessary choice. One might want to include rate of convergence, maximum speed, or some other condition in the criteria.

While second order control problems (problems where we only have control of $\frac{d^2x}{dx^2}$) can be difficult, it seems that controlling the first order system should be more or less trivial. Intuitively, all we should have to get right is the sign of the control signal. While this isn't technically true,[7] the control signal can tolerate a wide range of variation and still converge. As a result, we do not actually need to have an accurate estimate of the the error. In particular, we do not have to have a precisely calibrated measurement of the error, any monotonic measure which accurately estimates zero error will suffice. We can formalize this as an optimization which allows us to substitute uncalibrated estimates for calibrated ones. Thus we can call the optimization "decalibration."

Theorem 1 *(Control decalibration) Let x be a physical system with one degree of freedom whose rate of change car be directly controlled. Then all control systems whose control laws are of the form*

$$\frac{dx}{dt}(t) = -f(x(t) - x_{set}) \tag{4.1}$$

where $f{:}\mathcal{R}\rightarrow\mathcal{R}$ is nondecreasing and $f(x) = 0$ iff $x = 0$, will cause $x(t)$ converge to x_{set} for arbitrary values of x_{set} and so all such control systems are equivalent under our definition.

Proof: Without loss of generality, we will assume that $x_{set} = 0$. We want to show that $\lim_{t\rightarrow\infty} x(t) = 0$. This equation must have a unique, continuous solution for a given initial value (see Brauer and Nohel [16], theorem 1.1). Note that x and its derivative must always have opposite sign, except when one is zero, in which case the other must also be zero. Thus x must be strictly decreasing over any interval in which x is strictly positive, and strictly increasing in any interval in which x is strictly negative. Suppose that $x(t_0) = 0$ and that $x(t)$ becomes positive before ever becoming nonpositive in some interval $[t_0, t_1]$. Then $x(t_1) > 0$ and x is nonnegative in $[t_0, t_1]$. By the mean value theorem, there must be some $t \in [t_0, t_1]$ for which $x'(t) > 0$, a contradiction. Similarly, x cannot first become negative, and so $x(t)$ must be zero for all $t > t_0$. Thus x can never cross zero.

Since x will stay at zero one it reaches zero, we need only consider the case in which x stays either positive forever or negative forever. Supppose $x(t) > 0$ for all $t > t_0$, and so $x(t)$ is strictly decreasing for all $t > t_0$. Let $\epsilon > 0$. We want to show that there exists some t_ϵ such that $x(t) < \epsilon$ for all $t > t_\epsilon$. Since x is strictly decreasing, we need only show that $x(t)$ eventually reaches ϵ. Suppose it does not. Then x must always be larger than ϵ, and so x' must always be more negative than $-f(\epsilon)$, meaning that x is bounded above by $x(t_0) - (t - t_0)f(\epsilon)$. But this drops below ϵ, and so x must too. Thus $\lim_{t\rightarrow\infty} x(t) = 0$. The limit holds by similar reasoning when x remains negative, and so must hold in general. □

The caveat to this theorem is that we live in a second order universe and so few physical systems can be accurately modeled as being first order systems with zero

[7]The system could converge to an incorrect value.

delay. However, it is often the case that the rate of change of the second order system is quickly and easily measured, while the absolute (or relative) position of the system is much more difficult to measure. An example might be a robot driving in a room. The robot can easily measure its speed by sensing currents or reading shaft encoders, but accurate information about the robot's position relative to obstacles might require the use of a vision system or other sensing modality whose latency is large compared to the accelerations which the motors can produce. In such situations, we can reduce the second order control problem to two first order control problems: one to control velocity using motor torques, and the other to control position by adjusting velocity. Velocity control can often be done fast enough to make the robot look like a first order system to vision. It is still necessary for the visual system to run fast enough to prevent oscillation however.

An important implication of this theorem is that such a system will be insensitive to errors in the calibration of their perceptual systems provided that (1) the system's estimate of the error is still monotonic in the actual error, and (2) the system still recognizes when there is zero error. Calibration can be a major problem for perceptual systems, particularly when the systems are driven around on robots which periodically crash into things, thus getting knocked out of calibration.

4.6 Related work

Recent years have seen a resurgence in interest in the analysis of specialized systems. Most of this research has involved the use of dynamical models of intelligent action or perception.

The three general movements in which dynamical models have been most common are the *artificial life* movement (see Langton [76][77][78], and Varela and Bourgine [132]), the biologically oriented AI movement (see Meyer and Wilson [89][90]), and the *situated action* movement (see Agre [3] and Suchman [120] for representative work). The term "nouvelle AI", due to Ballard, has also been used. Within the vision community, a host of researchers have developed action oriented theories of perception (see section 3.1.2). Most of these groups have indeterminate boundaries and there is a considerable amount of overlap between them, so I will not worry about placing individual researchers in camps.

There has been a great deal of interest in the using the theoretical machinery of nonlinear-dynamics to understand intelligence. Beer [18] has explicitly argued that nonlinear dynamics is a useful framework in which to understand intelligent systems, particularly biological systems. Beer also contrasts dynamical systems with computational systems. He argues that while a computer is certainly a dynamic system, a dynamic system need not perform any computation. Beer uses the example of the solar system and points out that one would not want to argue that the planets compute their orbits. A number of researchers have suggested using the theory of nonlinear control to understand intelligence, particularly learning. Reinforcement learning can be viewed as an instance of adaptive nonlinear control, see Barto *et al.* [126]. Potential field models, such as those of Khatib [68] or Arkin [8], when

performed on-line, amount to gradient desent in a nonlinear dynamic system.

Other researchers have used automata-theoretic models of interaction. Rosenschein and Kaelbling model agent and environment as interacting finite state automata [108]. Given knowledge of the workings of the environment, one can define the semantics of a state of the agent automaton in terms of the constraint it places on the possible states in which the environment automaton can be. Lyons and Arbib [85] model agents and environments as interacting collections of port automata.

Other researchers have used less formal approaches. Agre has proposed studying not just the space of possible machinery, but the space of possible interactions between machines and environments, and adopting the methodology of using the simplest machinery which can perform a given interaction (the principle of machinery parsimony, see [3, p. 26]). Agre argues, in effect, that it is much simpler to design a system which gradually adapts its control policy to perform a set of basic routine interactions than to design a system which redetermines from scratch how to perform a task each time it needs to perform it. Hammond and Converse have discussed how agents can actively maintain invariants in their environments to simplify further processing [56].

Along with the interest in modeling agent/environment interactions, there have been a number of proposed agent architectures which emphasize the relationship of agents to their tasks and environments (see for example Brooks [24], Lyons and Arbib [85], Gat [49][50], Firby [47], Simmons [114], Agre and Chapman [4], Rosenschein and Kaelbling [108], Kaelbling [67], and Schnepf [111], Spector and Hendler [116], Hendler and Sandborn [58], or the collections [89][132][77][78]). These architectures share a common concern with the need to adapt to changes in an uncertain environment. All involve some sort of reactive component for dealing with changes in the environment.

The interest of these researchers in dynamical models usually stems not from an interest in specializing a system *per se*, but from an interest in understanding intelligent action as arising from the interaction of an agent and its environment. Therefore, research rarely emphasizes the specific properties of the environments with which it is concerned. Many researchers have focused on designing that gradually evolve their reasoning methods and control policies to fit solve the typical mix of problems which they face. Perhaps the earliest example of this is the MACROPS feature of Shakey's planning and execution systems (see Fikes, *et al.* [46]). Whenever Shakey's planner, STRIPS, developed a plan to solve a particular goal, it would generalize that plan as much as possible. The generalized plan involved some conditionalization which made execution easier. It also formed a higher level operator which was used by STRIPS as a macro (a MACROP) to simplify later planning problems. A closely related example of this is the *chunking* feature of SOAR, a complete architecture for cognition developed by Newell, Laird, and Rosenbloom (see Rosenbloom [106], Laird *et al.* [79], or more recently, Newell [94]). Agre's running argument system [3], the forerunner to Pengi, uses a learning mechanism based on a truth maintanance system (see Doyle [42]) to similar ends. Another similar mechanism is *case-based reasoning* (see Riesbeck and Schank [105]). Case-based reasoning is similar to these other strategies in that it involved recycling old bits of thinking to solve new problems, but both the matching processed involved in determining the correct case to use, and the processing required

to modify it to fit the current situation are much more complicated.

Marr [87] used constraints to analyize the computational problems involved in reconstruction problems in vision. Since a given image could be generated by an infinite number of scenes, the stimulus underdetermines interpretation. Fortunately, most of the possible scenes never occur in practice. If we add additional constraints on the interpretation of images, then we can often force the image to have a unique interpretation. Marr's constraints were intended to hold of nature in general, not simply of individual environments and so were constraints on interpretation, not on habitat. The use of habitat constraints here is an extension of Marr's approach.

Chapter 5

Analysis of simple visual systems

We need a formal notion of what it means for two vision systems to be equivalent. In the next section, I will develop a formal semantics for a simple class of perceptual systems called instantaneous state estimators. Then, in section 5.2, I will use the semantics to formalize equivalence for the perceptual systems. Given this notion of equivalence, we will then go on to define some useful optimizations over these systems.

The semantics, while necessary for formal rigor, is not particularly original or enlightening. The reader less concerned with formalism may wish to skip to section 5.3 or even chapter 7.

The semantics is similar in spirit to denotational semantics (see Schmidt [110]) in that the possible semantic values are functions over a fixed domain, rather than sets or specific individuals. The purpose is somewhat different however in that it is not a semantics of the program text, but rather a semantics of the output values of the perceptual system. Denotational semantics is also intended to be used to mechanically derive meanings from program texts, whereas this semantics is intended only to be used to prove that certain transformations preserve semantics. An attempt to directly specify the semantics of the program text can be found in Dixon [40]. Dixon's approach is similar to the approach used here. For example, it attempts to model the sorts of assumptions which a system makes about its environment. Dixon's interest however is in what an individual program means, rather than on comparing competing programs. In addition, Dixon derives his semantics from first order logic, in which the world comes individuated into objects and relations, rather than on the state-space models used here. Dixon's "open" approach also avoids the need to define the environment as a single mathematical structure.

The reader may also note obvious similarities between this semantics and the standard Tarskian semantics of classical logic (see Quine [102] for a good introduction). Its closest relationship however is to the work of Rosenschein and Kaelbling (see Rosenschein [107], and Rosenschein and Kaelbling [108]). Both are formal, causal semantics. The major difference is that the semantics presented here is much simpler than Rosenschein's, but consequently much more limited. More recently, Donald and Jennings [41] have presented a similar semantics derived within the "LMT" framework for fine motion planning (see Lozano-Pérez, Mason, and Taylor [82]). Like Rosenschein's, it has broader coverage than the one presented here and is conse-

quently more complicated. All three systems give rise to similar algebraic structures, including lattices of possible semantic values.

5.1 Semantics

We will think of the world as a system which evolves through a series of identifiable states. We can then think of a perceptual system as being a subsystem of the world which in turn has its own subsystem, its output, whose state is causally determined by the state of the world in some interesting way. Thus we can determine something interesting about the state of the world by examining only the state of the output. A simple but important class of such systems are those whose state at any point in time is completely determined by the present state of the world. Such systems can be interpreted as delivering information about the instantaneous state of the world and so will be referred to as *instantaneous state estimators.* We will restrict our attention to such systems.[1].

More formally, let W be the set of possible world states, and let S be the set of possible outputs of the system. If the state of the output is completely determined by the world state at all times, then the system effectively computes a function from W to S. We will call such functions *pieces of information.*

Definition 1 *A* piece of information $I{:}W{\rightarrow}S$ *is a total function over the set of world states* W *and some image set* S. *A piece of information is* finite *if its range is finite.*

The fact that I is a total function simply means that it assigns a value for every element of W, since the system must be in some state for every possible world state.

It would appear that in assuming that there is a single set of world states W, we are limiting ourselves to considering only a single environment. In the sections below, we will assume that W is rich enough that we can describe a number of different environments by imposing restrictions on which elements of W can occur in practice. Thus an environment will correspond to a subset of W, and a habitat to a set of subsets. While this is not entirely satisfactory, some such move is necessary in order to make formal sense of what it means for a system to operate in different environments.

Note that the concept of a piece of information should not be confused with the concept of information as inverse probability used in classical information theory (see Hamming [55]). While the two theories are certainly compatible, classical information theory is concerned with measuring quantities of information, whereas our concern here is with distinguishing among different types of information.

[1]Some subtlety is required here since the computational state of the perceptual system is itself part of the state of the world. This makes it difficult to say what it means to compare two different systems which compute the "same" information about the state of the world. This problem can be overcome by leaving the computational state of the perceptual system out of the world state, or equivalently, to say that the perceptual system gives information about the state of the quotient space of the true world state and the set of possible computational states of the perceptual system. Then different perceptual systems run in concert with the same environment will produce distinct sets of possible world states, but isomorphic quotient spaces.

5.1.1 Derivability and equivalence

Often what is interesting about a piece of information is what other pieces of information can be computed from it.

Definition 2 *A piece of information I' is* derivable *from another piece of information I if there exists a total onto function f such that $I' = f \circ I$. We will call $f: S \rightarrow S$ a* derivation function. *I_1 and I_2 are* equivalent *(written $I_1 \equiv I_2$) if they are interderivable.*[2]

(Note that derivability is both reflexive and transitive, and that interderivability is also symmetric. Thus interderivability is a valid equivalence relation).

Notice that for the purposes of derivability, it does not matter to what set a piece of information projects, that is, what its range is. We can arbitrarily rename the elements of the range of the function without changing which pieces of information can be derived from it. In a sense, what really matters is what world states the piece of information allows us to distinguish, and what states it folds together. We will say that a piece of information, I, induces an equivalence partition

$$P_I = \{A \subseteq W \mid x, y \in A \Leftrightarrow I(x) = I(y)\}$$

on the set of world states[3]. Two world states are in the same equivalence class within the partition iff they project to the same value under I, and so the elements of the partition are the sets of world states which cannot be distinguished given only the piece of information I. We are now in a position to prove the following lemma:

Lemma 1 *The following statements are equivalent:*

1. *I_1 and I_2 are equivalent, that is, interderivable.*
2. *X is derivable from I_1 iff it is derivable from I_2, for all X.*
3. *The partitions P_{I_1} and P_{I_2} are identical.*
4. *I_1 and I_2 differ only by a bijection (a 1:1 onto mapping).*

Proof: First, recall that derivability is transitive. Now suppose that (1) holds. Then if X is derivable from I_2 and I_2 is, by (1), derivable from I_1, then X is derivable from I_1. Similarly, if X is derivable from I_1, then it must be derivable from I_2. Thus (1) implies (2). Now assume (2). Since derivability is reflexive, I_1 and I_2 must be interderivable, and so, equivalent. Now suppose that (3) is false. Then there must be a pair of world states w and w' such that either $I_1(w) = I_1(w')$ but $I_2(w) \neq I_2(w')$ or vice versa. Without loss of generality, we may assume that it is I_2 which differs. By interderivability, there is a derivation function f such that $I_2 = f \circ I_1$ and so $f(I_1(w)) \neq f(I_1(w'))$, which contradicts the assumption that $I_1(w) = I_1(w')$. Thus

[2]The composition operator $\circ$ maps functions f and g to a function $f \circ g$ such that for all x, $(f \circ g)(x) = f(g(x))$.

[3]A partition P of a set S is a set of subsets of S that are pairwise disjoint but whose union is S. It is easy to show that P determines an equivalence relation on S in which $a, b \in S$ are equivalent iff there is a set in P which has both a and b as members.

(3) must hold, and so (2) implies (3). Now note that for every piece of information $I{:}W{\rightarrow}S$, there is a trivial bijection between S and P_I given by the rule $s \mapsto I^{-1}(s)$ for all $s \in S$. Since the inverses and compositions of bijections are themselves bijections, (4) must follow from (3). Finally, note that (1) follows trivially from (4) since the 1:1 mapping would itself be the derivation function. □

To formalize the notion of conditional equivalence of systems, we will need a notion of conditional equivalence of pieces of information. We can do this by taking habitat constraints to be predicates (*i.e.* subsets) over the set of world states W and taking conditional equivalence to be equivalence within the extension of the predicate:

Definition 3 *The restriction $I_D{:}D{\rightarrow}S$ of a piece of information $I{:}W{\rightarrow}S$ to $D \subseteq W$ is defined by $I_D(w) = I(w)$ for all $w \in D$. Two pieces of information I and I' are* conditionally identical/equivalent *within a set of world states $D \subseteq W$ if their restrictions are identical/equivalent. We will write this as $I \overset{D}{=} I'$ or $I \overset{D}{\equiv} I'$, respectively.*

It is easily shown that for all pieces of information I_1, I_2 and I_3, and for all $D, D_1, D_2 \in W$,

1. $I_1 = I_2 \Rightarrow I_1 \equiv I_2$
2. $I_1 = I_2 \Rightarrow I_1 \overset{W}{=} I_2$
3. $I_1 \equiv I_2 \Rightarrow I_1 \overset{W}{\equiv} I_2$
4. $I_1 \overset{D_1}{=} I_2$ and $I_2 \overset{D_2}{=} I_3 \Rightarrow I_1 \overset{D_1 \cap D_2}{=} I_3$
5. $I_1 \overset{D_1}{\equiv} I_2$ and $I_2 \overset{D_2}{\equiv} I_3 \Rightarrow I_1 \overset{D_1 \cap D_2}{\equiv} I_3$

5.2 Equivalence of perceptual systems

We can now use pieces of information as our model of the meaning of the output of a simple visual system, and thus as our test of equivalence:

Definition 4 *A system* computes *a piece of information $I{:}W \rightarrow S$ if for all $w \in W$ the output state of a system is $I(w)$ whenever the world is in state w. Two systems, M_1 and M_2, are* strongly equivalent *(written $M_1 \equiv M_2$) if they compute identical pieces of information. Two systems are* weakly equivalent *(written $M_1 \approx M_2$) if they compute equivalent (but not necessarily identical) pieces of information,* i.e. *if they are identical up to a renaming of their output states.*

Thus two systems are equivalent if their input-output behaviors are indistinguishable.

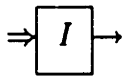
We can also extend the notion of equivalence of systems to conditional equivalence in the straightforward manner:

Definition 5 *Two systems, M_1 and M_2, are conditionally strongly/weakly equivalent given D if the pieces of information they compute are conditionally identical/equivalent within D. Conditional strony equivalence is denoted $M_1 \overset{D}{\equiv} M_2$, conditional weak equivalence by $M_1 \overset{D}{\approx} M_2$.*

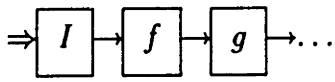
We will generally restrict our attention to strong equivalence and will simply refer to systems as being equivalent or conditionally equivalent.

5.3 Unconditional equivalence transformations

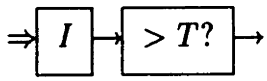
For the purpose of expressing transformations over perceptual systems, it is useful to represent these systems diagrammatically. We will use the notation



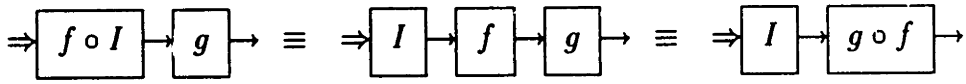
to represent a perceptual system which (somehow) computes the piece of information I. The double arrow is meant to represent the fact that the input of the system is connected to the environment, whereas the single arrow leading out is simply a signal being carried on, say, a wire. When we want to expose the internal structure in the system, we will use the notation



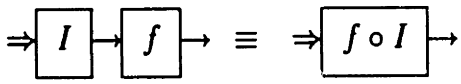
To represent a system which first computes I and then applies the transformations f, g, ...to it. Finally, we will denote predicates with the "?" syntax often used in lisp, thus



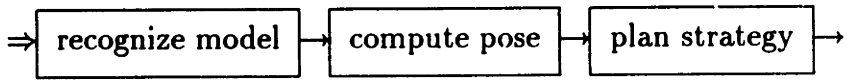
denotes a system which outputs true when $I(w) > T$, and false otherwise, for all world states w. These diagrams inherit the associativity of function composition:



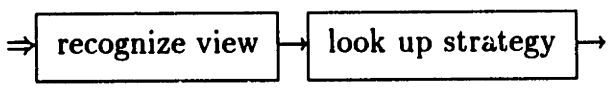
and so a simple optimization, which we might call "folding" (after constant-folding in compiler optimization), is the replacement of a series of computations with a single computation:



While folding is formally trivial, the technique is quite useful in practice. For example, rather than computing pose information from an image and then running a grasp planner to choose a grasping strategy, one might use the object's 2D appearance to index directly into a table of grasping strategies. To abuse our notation somewhat, we might say that



could be reduced to



Another example of an optimizing transformation is what might be called "decal-

ibration." Estimating precise parameters such as depth can be very difficult and can require very precise calibration of the sensor. Often all that is done with this information is to compare it to some threshold which has been determined experimentally. For example, we might estimate the distance to an obstacle to decide whether we should swerve around it, or whether it is too late and we must brake to avoid collision. That cut-off point is not generally calculable analytically. Either the designer arbitrarily chooses a threshold, or determines it experimentally, or perhaps lets the robot learn it by trial and error. In any case, what is important to the robot is not the actual distance to the obstacle in centimeters, but simply whether it is or is not closer than the threshold. Given that fact, we could substitute the image-plane height measure discussed above, provided that the ground-plane constraint holds. In fact, we can substitute *any* mechanism which computes a measure of distance, provided that we alter the threshold to be expressed in the same scale as the new measurement. We can express this transformation in the general case as follows:

Lemma 2 *(Decalibration) For any piece of information $I{:}W{\to}\mathcal{R}$ ($\mathcal{R}$ is the set of real numbers) and any strictly increasing function $f{:}\mathcal{R}{\to}\mathcal{R}$,*

1. $\Rightarrow \boxed{I} \rightarrow \approx \Rightarrow \boxed{f \circ I} \rightarrow$

2. $\Rightarrow \boxed{I} \rightarrow \boxed{> T?} \rightarrow \equiv \Rightarrow \boxed{f \circ I} \rightarrow \boxed{> f(T)?} \rightarrow$

Proof: Part 1 follows from that fact that any strictly increasing function over the reals must be a bijection and so the pieces of information I and $f \circ I$ must be equivalent. Part 2 follows from the fact that, by associativity, the right hand side is equivalent to

$$\Rightarrow \boxed{I} \rightarrow \boxed{(> f(T)?) \circ f} \rightarrow$$

and for all x, $f(x) > f(T)$ iff $x > T$, thus $(> f(T)?) \circ f = (> T?)$. □

Decalibration allows an expensive mechanism which computes some piece of information I to be replaced with a cheaper mechanism which computes $f \circ I$, when such a mechanism exists.

Decalibration is a useful special case of a more general technique of substituting an inexpensive computation for a weakly equivalent, but more expensive computation. A more general case of this is *hashing*:

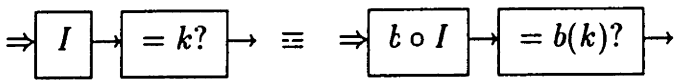
Proposition 1 *(Hashing) If I and I' are finite and equivalent pieces of information, then there exists a system computing a finite bijection b such that:*

$$\Rightarrow \boxed{I} \rightarrow \equiv \Rightarrow \boxed{I'} \rightarrow \boxed{b} \rightarrow$$

That is, weakly equivalent systems can be made strongly equivalent.

Proof: By lemma 1, I and I' differ only by a bijection, so let that bijection be b. If they are finite, then b is also finite. Since all finite bijections are computable, e.g. by table lookup, there is a system which will compute b. □

Here the computation of I' loses any relational structure which I might impose on the equivalence classes of the world states, but it does generate the same equivalence classes and so can distinguish exactly the same world states that I can. Thus it is, in principle, possible to recover I from I', say by table lookup and/or learning. If the relational structure is unnecessary, then computing I is unnecessary. Such might be the case if all we were concerned with was whether the current situation was the same as a previous situation. Then all that is necessary is to test the value of I' to see if it is the same as on the previous occasion. Formally, we have that for any bijection b,



by the same reasoning used in part 2 of lemma 2.

5.4 Conditional equivalence transformations

Having formalized the notions of world states, pieces of information, and conditional equivalence, we can now formalize some of the notions found in section 4.3. As mentioned previously, we will assume that W is large enough to encompass a wide range of environments, and so we can specify an environment simply by restricting W.

Since this section concerns vision algorithms, we will also have to assume the existence of some special functions. Let $R \subset \mathcal{R}^2$ be the set of points in the image plane of our camera and for an arbitrary set X, let $\mathcal{I}(X)$ be the set of mappings from R to X. A black and white image can be though of as a mapping from image plane points to intensity values and so is an element of $\mathcal{I}(\mathcal{R})$, while a color image would be an element of $\mathcal{I}(\mathcal{R}^3)$ because it has red, green, and blue components. We will assume that the set of world states has been formalized sufficiently for a world state to determine the output of a camera. Thus a black and white camera can be though of as providing a mapping from W to $\mathcal{I}(\mathcal{R})$, a piece of information.[4] Let us restrict ourselves to the black and white camera case, and call the piece of information computed by the camera the projection function, or P. Since P is a piece of information, it must have a generalized inverse, $P^{-1}{:}\mathcal{I}(\mathcal{R}) \rightarrow 2^W$, which gives the set of world states which could cause the camera to produce a given image. We may generalize the inverse further to cover sets of images:

$$P^{-1}(S) = \bigcup_{i \in S} P^{-1}(i)$$

Thus we can convert constraints on images to constraints on world states, at least in

[4]This is an idealization. In particular, it ignores camera noise.

principle. It is often useful to phrase habitat constraints in terms of images rather than world states, and so we will be somewhat cavalier about mixing assertions about world states, images, surfaces in view, *etc.*

5.4.1 Salience functions and figure/ground separation

We discussed in section 4.3.4 how the background texture constraint can make it easier to solve the figure/ground problem. Clearly, any easily computed local function of an image which happens to discriminate between object pixels and background pixels could be used. For example, color (see Turk et al. [130], for an example) or motion (see Horswill and Brooks [64]) could be used. Let us refer to such functions as "salience functions."

Suppose the world consists of a set of objects in front of a background and suppose that we want to determine what pixels in an image correspond to objects and what pixels correspond to background. We will call this the figure/ground separation problem.[5]

Definition 6 *The figure/ground function for a set of objects O, FG_O:$W \rightarrow \mathcal{I}(\{T,F\})$, is that piece of information which maps a world state w to that image which maps a point to T iff that point was imaged from the surface of one of the objects O in world state w.*

Definition 7 *A function s:$\mathcal{I}(\mathcal{R}) \rightarrow \mathcal{I}(\{T,F\})$ is a* salience function *under $D \subseteq W$ for a class of objects O if $s \circ P$ is locally identical to FG_O within D. D is referred to as a* saliency constraint.

We can place the analysis of section 4.3.4 in this framework as follows:

Lemma 3 *Let $\omega, d > 0$ and let BGT be the set of world states in which the surface markings of all local patches of the background have zero energy in the interval $(0,\omega)$, the surface markings of all patches of the objects O have nonzero energy in the interval $(0,\omega)$, and no surface is in view which is closer than d focal lengths from the camera. Then any linear filter with a low cutoff above DC and a high cutoff less than $d\omega$ composed with a point-wise test for non-zero energy is a salience function for O given BGT.*

Proof: By the reasoning in section 4.3.4.

Note that this is a rather restrictive formulation in that it assumes that all objects are heavily textured and that we need a perfect segmentation. In practice, the former is usually false, but the latter is often unnecessary. We could then speak of "weak" saliency constraints or of approximate equivalence to FG_O, but it is not worth formalizing such things at this point since the metric used to measure closeness to FG_O is likely to be highly task dependent.

[5] "Figure/ground separation" has been used to refer to many distinct problems. The one used here is not meant to be definitive.

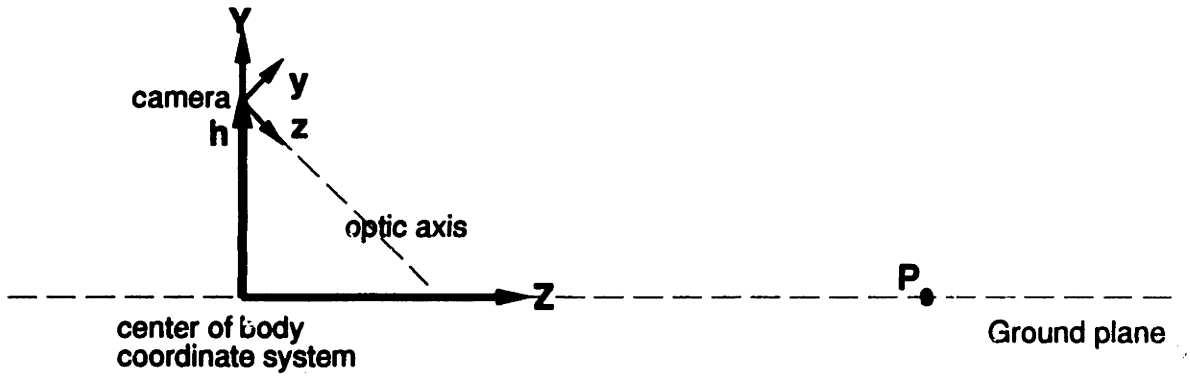


Figure 5-1: A camera viewing a ground plane.

5.4.2 Depth recovery

One way of formalizing depth recovery is as the computation of a *depth map*. A depth map assigns for each point in an image the distance from the image plane to that point in three-space which was imaged by the camera at the point.

To make this problem more precise, we need to define the coordinate systems of the agent and its eye/camera. Let $\mathbf{X}, \mathbf{Y}$, and $\mathbf{Z}$ be the axes of the agent's body-centered coordinate system as in figure. We will assume here that the agent is resting on the plane $Y = 0$. Suppose the camera is mounted a distance h above the origin and facing forward, with its optical axis in the $Y - Z$ plane. If we let $\mathbf{X}, \mathbf{y}$, and $\mathbf{z}$ be the axes of the camera-centered coordinate system (note that the camera coordinate system and the body coordinate system share the $\mathbf{X}$ axis), then we will have the arrangement shown in figure 5-1. In the body-centered coordinate system, the location of the projection of a point $\mathbf{P}$ in the image plane will be given by

$$\mathbf{p} = \frac{f(\mathbf{P} - \mathbf{h})}{\mathbf{z} \cdot (\mathbf{P} - \mathbf{h})}$$

where f is the focal length of the camera, and so the image-plane x and y coordinates of the projection will be $\mathbf{X} \cdot \mathbf{p}$ and $\mathbf{y} \cdot \mathbf{p}$ respectively.

It is worth noting at this point that there are a number of measures which one might want to call "depth" or "distance" here. The distance from the image plane to a point, that is $\mathbf{z} \cdot (\mathbf{P} - \mathbf{h})$, is the traditional definition of depth, and the one used above in the definition of a depth map. On the other hand, a more behaviorally relevant value might be the distance of the point to the fronto-parallel plane of the agents body, given by $\mathbf{Z} \cdot \mathbf{P}$. I will call these two measures "camera depth" and "body depth" respectively. Although they are interconvertible, we will be most interested in body depth since it determines the time to collision with an object.

Suppose that all objects in the world are flat and lie on the ground plane. Then all points in an image will be generated by points in the world which lie in the ground plane and so have a zero Y coordinate. Thus such a point must be a linear combination of X and Z. Since both $\mathbf{z}$ and $\mathbf{Z}$ are perpendicular to $\mathbf{X}$, the X component of the point will make no contribution either to camera depth or to body depth and so we

can consider the one dimensional case, shown in the figure, of a point $\mathbf{P} = n\mathbf{Z}$. Here the body depth is simply n, while the camera depth $\mathbf{z} \cdot (n\mathbf{Z} - \mathbf{h})$ will depend on the placement of the camera. We can see by inspection, however, that the camera depth will be a linear function of n and so camera depth and body depth will be related by monotonic mappings.

More surprising is the fact that the image plane height, given by

$$\mathbf{y} \cdot \left(\frac{f(n\mathbf{Z} - \mathbf{h})}{\mathbf{z} \cdot (n\mathbf{Z} - \mathbf{h})} \right) = \frac{\mathbf{y} \cdot (fn\mathbf{Z} - f\mathbf{h})}{n\mathbf{z} \cdot \mathbf{Z} - \mathbf{z} \cdot \mathbf{h}} = \frac{n\alpha - \delta}{n\beta - \gamma}$$

where $\alpha = f\mathbf{Z} \cdot \mathbf{y}$, $\beta = \mathbf{z} \cdot \mathbf{Z}$, $\gamma = \mathbf{z} \cdot \mathbf{h}$, and $\delta = f\mathbf{h} \cdot \mathbf{y}$, is also a monotonically increasing function of n. This can be seen by differentiating with respect to n, yielding

$$\frac{\alpha(n\beta - \gamma) - \beta(n\alpha - \delta)}{(n\beta - \gamma)^2} = \frac{\beta\delta - \alpha\gamma}{(n\beta - \gamma)^2}$$

If we restrict ourselves to the case where the camera looks forward and the point is in front of the agent, then we have that $n, \beta, \delta > 0$, and γ and α must have opposite signs. Thus the above quantity, the derivative, must be strictly positive, and so the y coordinate must be a strictly increasing function of n. Thus for points on the floor, image plane height, body depth, and camera depth are all related by monotonic functions. Thus if we define Camera-Depth:$W \rightarrow \mathcal{I}(\mathcal{R})$, Body-Depth:$W \rightarrow \mathcal{I}(\mathcal{R})$, and Y:$W \rightarrow \mathcal{I}(\mathcal{R})$ to be those unique pieces of information which give the proper camera depth map, body depth map, and y coordinate map for a given world state, then we have that these are all locally equivalent given the constraint that the camera is viewing a ground plane, and that further, any map can be obtained from any other map by some pointwise monotonic mapping. (Note that Y is a constant function, meaning that it maps all world states to the same value, and thus the camera and body depths are also constant functions when the constraint holds).

Of course, this is not a particularly interesting case. A somewhat more interesting case is the case where we have a set of flat objects O (such as sheets of paper) resting on the floor. If we have already solved the figure/ground problem, that is, we have marked each pixel "object" or "background," then the y coordinate of the lowest pixel marked "object" will be a measure of the distance to the closest object. We can loosen this analysis by assuming only that the projection of any object in the ground plane is the same as the set of points at which it touches the ground plane.[6] Thus pyramids resting on their bases would satisfy the restriction, but not pyramids resting on their points (see figure 5-2).

Given this constraint, we can use least y coordinate as a measure of the depth of the closest object. Let us define Body-Depth$_O$ to be the piece of information which gives the correct body depth for pixels generated by one of the objects O, or ∞ for pixels generated by the background. If we let $\min_{\text{image}}$:$\mathcal{I}(\mathcal{R}) \rightarrow \mathcal{R}$ be the function

[6] Formalizing the notion of "touches" can be difficult (see for example Fleck [48], chapter 8) but we will treat the notion as primitive, since the particular formalization is unimportant for our purposes.

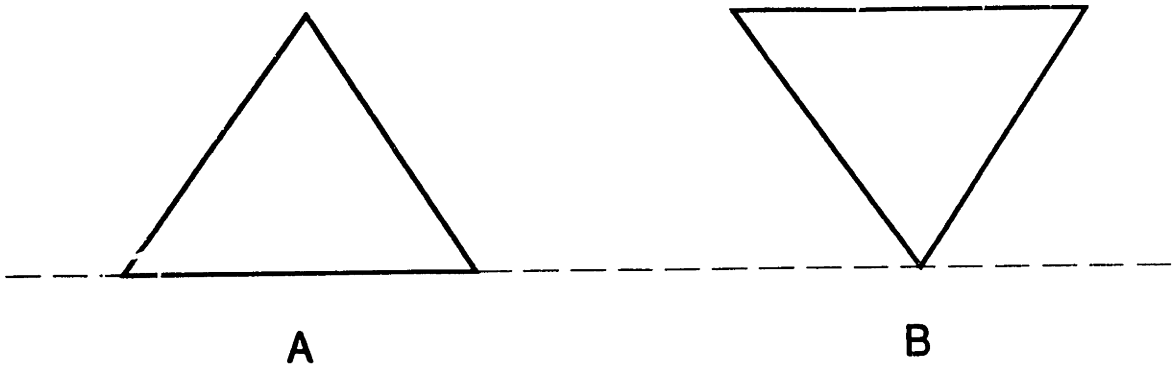


Figure 5-2: The projection of the pyramid A onto the ground plane fits within its points of contact with the ground plane, whereas the inverted pyramid B's projection does not.

which maps an image to its smallest value, then $\min_{\text{image}} \circ \text{Body-Depth}_O$ will give us the body depth of the closest object in view.

Theorem 2 *Let O be a set of objects and let GP be the set of world states in which the camera is viewing a subset of the objects O resting on a ground plane G, and for each $o \in O$, o's projection in G is its set of points of contact, and let* $\min_y : \mathcal{I}(\{T,F\}) \rightarrow \mathcal{R}$ *be the function which maps a binary image i to* $\min\{y | i(x,y) = T \text{ for some } x\}$. *Then* $\min_{\text{image}} \circ \text{Body-Depth}_O$ *is locally equivalent, and a monotonic function of,* $\min_y \circ FG_O$ *given GP. That is,* $\min_y \circ FG_O$ *gives the body depth of the closest object when GP holds.*

Proof: First, notice that there can only be one minimal body depth, but there can potentially be many object points with that depth. Now notice that it must be the case that some object point contacting the floor must have minimal body depth. If this were not the case, then no minimal depth object points would be points of contact, but then their projections onto the ground plane could not be points of contact (since a point and its projection must have the same body depth), a contradiction. Let p be a minimal depth contact point. We want to show that no object point can have a smaller projected y coordinate than p. Since the y coordinate is invariant with respect to changes in the X coordinate, a point which projects to a lesser y coordinate than p must have either a smaller Z coordinate or a smaller Y coordinate. The first would contradict p's being a minimal body depth point, and the latter would place the point below the ground plane. Thus p must have a minimal y projection. Since we have already shown that for points on the floor the y projection is monotonically increasing in body depth, the theorem must hold. □

A useful observation is that columns in the image correspond to distinct angles in the ground plane, and so the x coordinate of a pixel is a measure of direction. If we consider a column of pixels to be a very thin image, then by lemma 2, we can use $\min_y \circ FG_O$ to determine the distance to the closest object in any given direction.

Chapter 6

Analysis of discrete control problems

Recent years have seen a great deal of controversy over the issue of planning. The so-called "classical planning" theory of action states that all intelligent action is due to the use of plans. The theory is due to the psychologists Miller, Galanter, and Pribram [91] in response to the radical behaviorism theory, which held that all intelligent action was due to stimulus-response mechanisms trained by general learning mechanisms. Miller, Galanter, and Pribram argued that Plans[1] were essentially like computer programs. They argued that activity could be divided into two subactivities, "planning" and "execution." Planning was the process of choosing a plan to fulfill some goal, while execution was taken in the sense of program execution. Within the AI community the planning phase was interpreted as automatic programming—given a description of a task, *i.e.* a goal, derive a program which will provably achieve it. (Interestingly enough, Miller, Galanter, and Pribram did not take planning to be automatic programming.)

Planning, *qua* automatic programming, has two fundamental problems. First, it is generally intractable and often formally undecidable [27]. This means that it is very unlikely that it can be done in a timely manner in the general case. Second, the planning/execution dichotomy, at least when taken in its purest form, implies a temporal separation of thought and action—when you're thinking, you don't do, and when you're doing, you don't think. The temporal separation of planning and execution can have serious consequences in a dynamic environment.

The problems of radical behaviorism spawned the cognitive science movement (of which Miller, Galanter, and Pribram's planning theory was a part) as a reaction. The problems of the planning theory spawned a number of reactions of its own. One reaction was a return to stimulus-response models of activity, but this time with no assumption of a general learning system to configure them. Such systems were referred to as "reactive systems." The strong form of the "reactive theory," holds

[1] Miller, Galanter, and Pribram used the convention of capitalizing when referring to Plans in the technical sense of the word, and using lower case, when referring to the non-technical sense. This convention has recently been called back into use by Agre and Chapman [4].

that most or all of activity is done without planning and simply involves mapping immediate sensor data to actions. Reactive systems use constant-time algorithms which compute mappings from sensors to actuator commands without little or no internal state. The advantages of reactive systems are that they tend to be fast, simple, and respond quickly to changes in the environment. The disadvantages are that they need to be hand-crafted by programmers, that they often get caught in loops and local minima, and that some problems simply cannot be solve without internal state. The reactive/planning controversy has led to extensive research into which problems are best suited to which strategies (see Hayes-Roth [57]), and a number of attempts to develop hybrid architectures which preserve reactivity while also allowing for symbolic reasoning about goals and actions (see Firby [47], Gat [50], Georgeff and Lansky [52], Hendler and Sandborn [58], Maes [86], Schoppers [112], Simmons [114], Spector and Hendler [116], or the collections [89][132][77][78]).

The awful truth is that depending on how one defines ones terms, either everything falls into both the planning and the reactive categories, or almost nothing falls into either. This is because the two categories are interreducible. A reactive system can be viewed as a system which repeatedly computes the first step of a plan and executes it. A planning system can be viewed as a reactive system which reacts to the absence of a plan by making a plan, and which reacts to the presence of a plan by dequeuing an action from it and executing it.

Most real systems of any complexity have some amount of internal state used to guide actions and do some amount of what one might call reacting. The planning and reacting strategies sit at opposite ends of a continuum of *temporal commitment*. The stereotypical planner/executive architecture commits completely to its future course of action once it finishes planning. The stereotypical reactive system never commits any farther than one clock tick. In between is a huge space of possibility for different amounts and different *kinds* of commitment. It is this space that I would like to understand.

To understand this space, we need to put planning systems and reactive systems on equal theoretical ground. The simplest way to do this is to reduce one to the other. For a number of reasons, I have chosen to reduce the planning systems to reactive systems. I will take action selection problems to be the problem of finding a mapping, called a *policy*, from sensor readings to actions which will bring the agent and the environment into some goal state. The stored state of planning systems can then be modeled as an odd kind of sensor/effector pair, one which can be read and written perfectly. (This will turn out to have useful theoretical consequences). Having built up the formal system for describing these problems, we can then formalize special properties of the problems which admit simpler policies than the general case.

The resulting formalization looks much like standard control theory. Given a dynamic system with inputs, one wants to find a control policy, that is, a mapping from the current state of the dynamic system to a new input for the dynamic system, such that the dynamic system will exhibit some desired behavior when run together with the control policy (see Luenberger [83] for a general introduction to the theory of dynamics systems and their control). Such a behavior might be to converge on a given state or family of states. A number of people have analyzed autonomous agents using

dynamic systems formalisms. Cybernetics, the forerunner of control theory has always used such characterizations (see Wiener [138]). Potential-field methods in robotics (see Khatib [68]) are another example. Beer has recently argued that autonomous agents are best understood using the specific language of dynamic systems theory [18]. Rosenschein and Kaelbling [108] and Lyons and Arbib [85] have both formalized the world and the agent as interacting finite state machines.

The formalism given here differs from most of the control-theoretic models in that it does not assume that the underlying state space is metric. It also assumes that the agent has several qualitatively different actions that it can perform, rather than just one parameterized action. As such, it is most similar to the work of Rosenschein and Kaelbling. The major difference from Rosenschein and Kaelbling is that I will not assume that the agent has any internal state. The agent is therefore not a finite state machine, *per se*. A finite state machine can be simulated by adding an extra component to the state of the world, a component which can be perfectly sensed and changed. The advantage of this move is that it is agnostic as to whether the state is physically realized within the agent's body, or whether it is realized on scratch paper or elsewhere.

Our goal will be to find structures in the world which will let us use simple, reactive strategies. I will discuss three such structures—progress functions, separability, and reducibility—which allow the designer to reduce action in a complicated state space to action in a number of simple spaces, most of which can be dealt with using simple reactive mechanisms. In chapter 9, I will use each of these techniques to reduce Polly's navigation problem to a set of smaller problems, each of which can be solved reactively.

6.1 Definition

We will call a *discrete control problem* (DCP) the problem of getting from one of a set of states to another by choosing from a set of actions. Here, the states are intended to be states of the world, not of the agent.

Definition 8 *A* discrete control problem *is a tuple* $D = (S, A, G)$ *where* S *is a state space (a set),* A *is a set of actions (mapping from* S *to* S*), and* $G \subseteq S$ *is a set of goal states. A finite sequence* $a = (a_1, a_2, ..., a_n)$ *is a* solution sequence *from an initial state* s *iff* $a_n(a_{n-1}(...a_1(s))) \in G$*. The problem is solvable from* s *iff such a sequence exists. It is solvable iff it is solvable from all initial states.*

This is largely the same as a traditional search problem. However, search problems are generally thought of as being solved in the agent's head without performing any actions in the world. The agent can do things in its head that it cannot do in the world, such as instant backtracking or searching multiple paths at once. In this chapter we will consider search (followed by execution) to be one possible strategy for solving discrete control problems and one of our goals will be to understand under what conditions it is necessary or desirable to use search. Note that formally a DCP is effectively a deterministic automaton, except that it need not be finite.

Our job will be to find an easily computed mapping from the current state to an action which will bring us closer to the goal. We will call such a mapping a *policy*:

Definition 9 *A* policy $p: S \rightarrow A$ *for a DCP* $D = (S, A, G)$ *is a mapping from states to actions.* p *generates a sequence of states* s_i *when* $s_{i+1} = (p(s_i))(s_i)$ *for all* i. p *generates an action sequence* a_i *when it generates a state sequence* s_i *for which* $a_i = p(s_i)$ *for all* i. p *solves* D *from state* s *iff* p *generates a solution sequence from* s. p *solves* D *iff it solves* D *from all states.* p *solves* D *and halts iff it solves* D *and for all* $s \in G$, $p(s) \in G$.

Note that if a problem is solvable then from any state there is a path from that state to the goal, and so for every state there is a first action along that path, thus there is a mapping from states to actions which move along a path to the goal. Thus there must be a policy for any solvable problem.

6.1.1 Discrete control with hidden state

The foregoing discussion assumed that the policy had perfect knowledge of the state of the world, which is not generally the case. Generally agents only have some projection $T: S \rightarrow X$ of the state which is delivered by the agent's sensors. Thus we are not concerned with all policies which solve the problem, but only ones which have the form $p = p_T \circ T$ for some p_T. In this case, T is effectively executed by the agent's perceptual system, which we will take to be given, and we must search for a p_T which can be executed by the agent's control system to solve the problem.

Definition 10 *Let* $D = (S, A, G)$ *be a discrete control problem and let* T *be a mapping from* S *to some set of sensory states,* X. *A* T-policy $p: X \rightarrow A$ *for* D *is a mapping from sensory states to actions.* p T-solves D *iff* $p \circ T$ *solves* D.

(Note that T is a piece of information in the sense of section 5).

6.1.2 Externalization of internal state

We have also assumed that the agent itself has no internal state–that its actions are determined completely by the state its sensors. In real life, agents generally do have internal state. State is particularly important when there is insufficient information in the sensor data to determine the proper action. Fortunately, we do not need to change our formalism greatly to cope with internal state. We need only treat the internal state of the agent as a part of the state of the world over which the agent has perfect control. The action chosen by the agent will then have two components: one which changes the world state, and here its options are greatly restricted, and one which changes the internal state to an arbitrary value.

Definition 11 *The* augmentation *of a DCP* $D = (S, A, G)$ *is the DCP* $D' = (S \times B^*, A', G \times B^*)$ *where* $B = \{0, 1\}$, B^* *is the set of finite binary strings, and* $A' = \{a' : a': (s, b) \mapsto (a(s), b')$ *for some* b' *and some* $a \in A\}$.

Thus the augmentation of D is a new DCP whose states are states from D paired with arbitrary binary strings and whose actions perform some action from D on the S part of the state while setting the binary string part to some constant. Its goal states are all states whose S part is in G. When giving policies for augmentations we will treat the policies as mappings from pairs to pairs. So the policy

$$p(s,b) = (f(s,b),\ g(s,b))$$

will, when in state s and with internal state b, perform the action $f(s,b)$ and move to the new internal state $g(s,b)$. For clarity, we will ignore the issue of how symbols are represented in binary. We will treat g as a mapping into an arbitrary discrete set and take it for granted that this set can be mapped into B^*.

Clearly, we can model a Turing machine interacting with the environment since we can use the binary string to represent the tape, and we can use the policy as the TM's state transition function. However, I will not provide a full proof that all agents with internal state can be modeled in this framework.

This shows that we can reduce the case of agents with internal state to the case of agents without internal state that operate in a special kind of world: one in which they have total control of some aspect of the external world's state. This equivalence is not simply a mathematical artifice, but is very important. The equivalence brings out the fact that any agent can use the world as an external memory. The agent need only isolate some portion of the world's state, such as the appearance of a sheet of paper or the location of a cooking utensil, which the agent can sense and control directly. Humans do this routinely. Appointment books allow people to keep their plans for the day in the world, rather than in their scarce memory. Bartenders use the position of a glass on the bar to encode what type of drink they intend to mix and how far they are into the mixing (see Beach [17]). The Toast program written by Agre and Horswill [2] also uses the position of objects to encode their relationship to the program's goals.

6.2 Progress functions and hill-climbing

Progress functions are a useful notion from the theory of formal verification of computer programs. A progress function is a mapping from program state to the positive reals which decreases steadily such that the program must terminate when the progress function reaches zero. The idea must be modified somewhat for our purposes since there is no well-defined notion of termination for a policy, and since the important state is the world state, not the program state:

Definition 12 *A function* $\pi: S \rightarrow \mathcal{R}$ *is a* progress function *for a discrete control problem* $D = (S, A, G)$ *iff*

1. $\pi(s) \geq 0$ *for all* s.
2. $\pi(s) = 0$ *implies* $s \in G$.
3. *For every state* s *from which* D *is solvable there exists a solution sequence* $(a_1, ...a_n)$ *for which* π *decreases monotonically, that is, for which*

$\pi(a_j(...(a_1(s)))) > \pi(a_{j+1}(a_j(...a_1(s))))$ *for all* j.

Progress functions are also very similar to Liapunov functions (see Luenberger [83]), admissible heuristics (see Barr and Feigenbaum [15], volume 1, chapter II), and artificial potential fields (see Khatib [68] or Latombe [80]). The definition is somewhat stronger than the definition of a discrete Liapunov function since it requires convergence in finite time (because ϵ is bounded below).

We will say that a policy p *honors* some non-negative function $\pi: S \rightarrow \mathcal{R}$, if it steadily decreases it until it gets to zero, that is if for all states s and some $\epsilon > 0$, either $\pi((p(s))(s)) < \pi(s) - \epsilon$ or $\pi(s) = \pi((p(s))(s)) = 0$ A policy which honors π can be thought of as doing hill-climbing on π and so will run until it reaches a local minimum of π. When π also happens to be a progress function for the DCP, that local minimum will be a global minimum corresponding to the goal:

Lemma 4 *Let* $\pi: S \rightarrow \mathcal{R}$ *be non-negative. If* p *honors* π *then* p *solves* D *exactly when* π *is a progress function on* D.

Proof: Consider the execution of p from an arbitrary initial state i. On each step, the value of π decreases by at least ϵ until it reaches 0, after which it must remain zero. Thus π must converge to zero within $\frac{\pi(i)}{\epsilon}$ steps after which the state of the system is confined to the set $\pi^{-1}(0)$. We need only show that $\pi^{-1}(0) \subseteq G$ iff π is a progress function for D. Since $\pi^{-1}(0) \subseteq G$ follows from the definition of a progress function, we need only show the converse. Suppose $\pi^{-1}(0) \subseteq G$. Since π is non-negative, we need only show that from every state from which D is solvable, there is a solution sequence which monotonically decreases π. Since the sequence generated by p is a such a sequence, and is a solution sequence, π must be a progress function for D. □

A very important case is when we can determine the effect of an action on the progress function given only the sensory state.

Definition 13 *A progress function* π *is* observable *given* T *if there exists some* g *such that* $\pi = g \circ T$. π *is* differentially observable *given* T *if there exists an* $\epsilon > 0$ *such that for all sensory states* $x \in X$ *and all actions* $a \in A$, *either* $\pi(a(T^{-1}(x))) > \pi(T^{-1}(x))$, $\pi(a(T^{-1}(x))) < \pi(T^{-1}(x)) - \epsilon$, *or* $\pi(a(T^{-1}(x))) = \pi(T^{-1}(x)) = 0$.

(Note that observability does not imply differential observability.)

Lemma 5 *If a DCP is solvable and has a progress function* π *which is differentially observable given* T, *then it is solvable given* T.

Proof: Let $\Delta: (A \times X) \rightarrow \{+1, -1, 0\}$ be defined by the rule:

$$\Delta(a,x) = \begin{cases} +1 & \text{if } \pi(a(T^{-1}(x))) > \pi(T^{-1}(x)) \\ -1 & \text{if } \pi(a(T^{-1}(x))) < \pi(T^{-1}(x)) - \epsilon \\ 0 & \text{if } \pi(a(T^{-1}(x))) = \pi(T^{-1}(x)) = 0 \end{cases}$$

Note that Δ must exist by the definition of differential observability. Now let p be some T-policy for which for all $x \in X$, $p(x) \in \{a : \Delta(a,x) = -1 \text{ or } \Delta(a,x) = 0\}$.

Such a p must exist, since otherwise either D would be unsolvable, or π would not be a proper progress function. We want to show that p solves D given T. By lemma 4, we need only show that $p \circ T$ honors π, that is, that for any state s, either $\pi(p(T(s))) < \pi(s) - \epsilon$ or $\pi(s) = \pi(p(T(s))) = 0$. By the definition of p, $\Delta(p(T(s)), T(s))$ is either -1 or 0. In the former case, $\pi(p(T(s))) < \pi(s) - \epsilon$, and in the latter, $\pi(s) = \pi(p(T(s))) = 0$. Thus $p \circ T$ honors π and so p solves D given T. □

Progress functions can be generated directly from policies. If we have a policy p which solves a problem D, then we will use $\pi_{p,D}$ to denote the progress function which maps states of D to the number of steps in which p will solve D when started from that state. We will call $\pi_{p,D}$ the *standard progress function* for p on D. The standard progress function is useful for combining policies for different problems to solve new problems.

A final useful observation about progress functions is that their actual values don't really matter, only the partial ordering they impose on the states. Therefore we can tolerate monotonic variations in them.

Lemma 6 *(Decalibration) Let π be a progress function for D, and let $f: \mathcal{R} \to \mathcal{R}$ such that $f(0) = 0$, and $f' > 0$. Then $f \circ \pi$ is also a progress function for D.*

Proof: Observe that $f(\pi(s)) \geq 0$ for all s and $f(\pi(s)) = 0$ iff s is a goal state. We need only show that for every state from which D is solvable, there is a solution along which $f \circ \pi$ strictly decreases. Since π is a progress function, there must exist such a solution along which π strictly decreases. Since $f(\pi(s)) > f(\pi(s'))$ iff $\pi(s) > \pi(s')$ by monotonicity, $f \circ \pi$ must strictly decrease along the path also. □

6.3 Reduction and isomorphism

The first thing one usually wants to do when classifying problems is to put them in some sort of taxonomy. This requires that we make precise the notions of what it means for two problems to be the same and of what it means for one problem to include another problem. Obviously, renaming the elements of the state space or action space of a DCP has no effect on the problem's intrinsic difficulty. We can define isomorphism for DCPs in the standard way: DCPs are isomorphic if there is a bijection b between their state spaces such that for each action a of one, $b^{-1} \circ a \circ b$ is an action of the other, and the goal of one is the result of element-wise mapping the goal of the other. Isomorphic DCPs are effectively equivalent.

Reduction of one DCP to another is a more complicated issue. Suppose we have two DCPs, $D = (S, A, G)$ and $D' = (S', A', G')$ Often, we can interpret the states of D' as being regions of the state space of D. In that case, the actions of D' are like higher-level actions of D and so if we have a policy which solves D' and some way of "implementing" its actions using actions from D, then we can solve D using these components and we have effectively reduced D to D'.

We will say that a *projection* of D into D' is a mapping $P: S \to S' \cup \{\perp\}$ such that $P^{-1}(G') \subseteq G$. Informally, P is the function which tells what state of D' corresponds

to each state of D, if any. Thus the region of D's state space corresponding to a state s' of D' is simply $P^{-1}(s')$. If $P(s) = \perp$, then there is no state in D' corresponding to s. $P^{-1}(G')$ is simply the set of states of D which project to some goal state of D'. In order for P to be a projection of D into D', these states of D must all be goal states of D. Note that $P^{-1}(S')$ is the set of all D states which project to some state of D'.

Now we must make sense of what it means to implement an action a' of D' inside of D. The behavior of a' for any given state s_1 is to move to some new state s_2. To implement this in D, we must have a way of getting from any place in the region corresponding to s_1 to someplace (it doesn't matter where) in the region corresponding to s_2. But this is simply another DCP, $(S, A, P^{-1}(s_2))$. If we have a policy which can solve this new DCP from any state in $P^{-1}(s_1)$, then that policy will faithfully implement the transition from s_1 to s_2. In general, a' will have transitions between lots of pairs of states, not just one. Therefore, we will say that *a policy $p_{a'}$ P-implements a' in D*, if for all $s_1 \in S'$ for which a' is defined, $p_{a'}$ solves the DCP

$$(P^{-1}(\{s_1, a'(s_1), \perp\}), A, P^{-1}(a'(s_1)))$$

from all states in $P^{-1}(s_1)$. The bizarre restriction on the state space of the DCP is to prevent the policy from getting to any other states of D' before getting to $a'(s_1)$.

Having set out these preliminaries, we can now show how to construct a solution to the augmentation of D from an arbitrary solution to D'. The augmentation is needed so that the policy can store the name of the high level action it is presently executing.

Lemma 7 *Let $D = (S, A, G)$, $D' = (S', A', G')$ be DCPs, P be a projection of D into D', and for each action $a' \in A'$, let $p_{a'}$ be a P-implementation of a' in D. If p' is a policy which solves D', then the policy*

$$p(s, a) = \begin{cases} (p_a(s),\ a) & \text{if } P(s) = \perp \\ (p_{p'(P(s))},\ p'(P(s))) & \text{otherwise} \end{cases}$$

solves the augmentation of D from any state in $P^{-1}(S')$.

Proof: Let $\pi_{p',D'}$ be the standard progress function for p' on augmentation of) D', that is, the number of steps in which p' will solve D' from a given state. For states in $P^{-1}(S')$, $\pi_{p',D'} \circ P$ will then be the number of high level steps needed to solve D from a given state. We can prove the theorem by induction on $\pi_{p',D'} \circ P$. Let $s \in S$ be the initial state in D. If $\pi_{p',D'}(P(s)) = 0$, then the problem is already solved. Now suppose that p solves the problem for $\pi_{p',D'}(P(s)) = n$, and consider a state for which $\pi_{p',D'}(P(s)) = n + 1$. The policy p will immediately compute $p'(P(s))$, store it into its state. Call this action a'. The policy p will also immediately begin executing $p_{a'}$. Since this policy is an implementation of a', the system will eventually reach a state in $P^{-1}(a'(P(s)))$, which is to say that it will reach the next state in D'. By assumption, p' can solve D' from this high level state in n steps, and so p must be able to solve D from the low level state. Thus p can solve the augmentation of D from low level states for which $\pi_{p',D'}(P(s)) = n + 1$. Thus p must solve D. □

We will say that D is *reducible* to D' if there exists a projection P of D into D' and there exist P-implementations of all of actions in D'. If D is reducible to D' then we can easily convert any solution to D' into a solution to D.

6.4 Cartesian products and separability

Often, the state space of the world is structured into separate components. For example, my position in space can be divided into x, y, and z components. Often, these components can be acted on independently. We can formalize this as follows. Let us define the Cartesian product of two functions f and g to be $f \times g: (a,\ b) \mapsto (f(a),\ g(b))$. Also, let i be the identity function. Suppose we have two DCPs, $D_1 = (S_1, A_1, G_1)$ and $D_2 = (S_2, A_2, G_2)$. We will call their *serial product* the DCP

$$(S_1 \times S_2, \{a_1 \times i : a_1 \in A_1\} \cup \{a_2 \times i : a_2 \in A_2\}, G_1 \times G_2)$$

and we will call their *parallel product* the DCP

$$(S_1 \times S_2, \{a_1 \times a_2 : a_1 \in A_1, a_2 \in A_2\}, G_1 \times G_2)$$

For both products, the state space consists of the pairs of states from D_1 and D_2, the components are changed using the actions of D_1 and D_2, and the goals are the pairs whose components are both themselves goals. However, for the serial product, the actions change one or the other component, but not both. The actions of the parallel product change both components at once.

An interesting observation is that the augmentation of a DCP is simply its parallel product with the trivial DCP (B^*, C_{B^*}, B^*), where C_{B^*} is the set of all constant functions from B^* to B^*.

The nice thing about product DCPs is that we can construct progress functions for them from progress functions which solve the components:

Lemma 8 *If π_1 is a progress function for D_1 and π_2 is a progress function for D_2, then $\pi: (x, y) \mapsto \pi_1(x) + \pi_2(y)$ is a progress function for the serial product of the two DCPs.*

Proof: First, we want to show that the product is solvable from a state (x, y) iff the component problems are solvable from x and y, respectively. Consider a solution to the product. It will consist of some interleaving of a series of actions from the first problem and another series of actions from the second (each extended to operate on pairs, of course). Since the sequence is a solution, it leaves the product problem in a state which is a solution to both problems, thus each subsequence must be a solution to its respective problem, and thus we can always solve both component problems if we can solve the product. Consider the converse: if we have solutions to the component problems, then we can simply extend their actions to pairs, and catenate them. The resulting sequence will be a solution to the product. Thus the product is solvable iff both components are solvable.

Now consider a state (x, y) from which the product is solvable. There must then exist solutions to the component problems which start from x and y, respectively, and which monotonically decrease π_1 and π_2, respectively. Again, the catenation of these two solutions is itself a solution. Since it decreases the sum of the progress functions monotonically, that sum must be progress function for the product. □

The parallel case is more complicated because there is not in general way to stop acting on one subproblem when it is solved but the other is still unsolved. This can lead to oscillation between one subproblem being solved and the other subproblem being solved. Thus it is not true in general that the parallel product of two solvable problems is itself solvable. If we add the further condition that the component DCPs "be able to stop" then we can prove the lemma. In this case, being able to stop means that once in a goal state there is always an action which will map to some (possibly different) goal state.

Lemma 9 *If π_1 is a progress function for D_1 and π_2 is a progress function for D_2, and for every goal state of D_1 and D_2 there is an action which maps that state to a goal state, then $\pi: (x, y) \mapsto \pi_1(x) + \pi_2(y)$ is a progress function for the parallel product of the two DCPs.*

Proof: Again, we must first show that the product is solvable iff the components are solvable. As with the serial case, we can see that a solution to the product contains solutions to the components. This time, they are combined by taking corresponding elements of the two solutions and forming pairs from them. Similarly, we can form a solution to the product from solutions to the components by combining them element-wise. To do this, the solutions to the components must be of the same length. This will not be true in general. By assumption, however, there will always be actions which can be performed which keep the components within the goal regions, once the goal regions are reached. Thus we can always pad the shorter component solution with such actions to make it the same length as the longer. The combination of the padded solution and the long solution, must then be a solution to the product. Now consider a state (x, y) from which the product is solvable. Again, there must exist solutions to the component problems which start from x and y, respectively, and which monotonically decrease π_1 and π_2, respectively. The shorter of these can be padded and combined with the longer to produce a solution to the product. Since the padding cannot change the value of $\pi_1 + \pi_2$ (because it moves from one component goal state to another), the combination must monotonically decrease $\pi_1 + \pi_2$. □

Product DCPs are important because we can construct policies for them from policies which solve the components:

Lemma 10 *Let p_1 be a policy which solves D_1 and halts from all states in some set of initial states I_1, and let p_2 be a policy which solves D_2 and halts from all states in I_2. Then the policy*

$$p(x, y) = p_1(x) \times p_2(y)$$

solves the parallel product of D_1 and D_2 and halts from all states in $I_1 \times I_2$. (Note that here we are using the convention of treating p, a function over pairs, as a function over two scalars.)

Lemma 11 *Let p_1 be a policy which solves D_1 from all states in some set of initial states I_1, and let p_2 be a policy which solves D_2 from all states in I_2. Then any policy for which*

$$p(x, y) = p_1(x) \times i \text{ or } p_2(y) \times i$$

and

$$\begin{aligned} y \in G_2, x \notin G_1 &\Rightarrow p(x, y) = p_1(x) \times i \\ x \in G_1, y \notin G_2 &\Rightarrow p(x, y) = i \times p_2(y) \end{aligned}$$

will solve the serial product of D_1 and D_2 and halt from all states in $I_1 \times I_2$.

We can prove both theorems using progress functions. Let π_{p_1,D_1} and π_{p_2,D_2} be the standard progress functions for p_1 and p_2 on D_1 and D_2, respectively. Their sum must be a progress function for the product. This follows directly for the serial case, and from the fact that p_1 and p_2 halt for the parallel case. Since the policies for both products clearly honor the sum, they must solve their respective products. Note that the constraint given in the second lemma is sufficient, but not necessary.

We will say that a DCP is *serially separable* if it is isomorphic to a serial product. We will say that it is *parallel separable* if it is isomorphic to a parallel product.

6.5 Optimization of control policies

Given a complete model of the state space and complete information about the present state, we could solve any discrete control problem using search. This is effectively the approach used by classical planners: planners represent states as collections of predicates and choose actions by searching the state space (or some abstraction space). The disadvantages of planning are that it is often expensive, and that complete information is not always available. In such cases, we would like to limit planning as much as possible.

The basic requirement for constructing a reactive control policy is that there be some observable progress function for the state space.[2] That is, it must be possible to determine from the sensor inputs alone (remember that we can model internal state as an odd kind of sensor/effector pair) what action will bring the agent closer to the goal (it must also be able to do so efficiently). If such a progress function can be found directly, then the problem is easily solved. In general, any time we can find a progress function and a reactive control policy which honors it, we can replace the planner with the control policy.

If there is no obvious observable progress function, then it may still be possible to break the problem into smaller problems, some or all of which can be solved reactively.

[2]This is overly strong if one move to nondeterministic domains.

One way to break the state space up is to separate it. If the state space is separable, then it consists of non-interacting subproblems which can be solved independently. Lemmas 10 and 11 allow us to optimize systems by replacing a complicated policy for the product space (such as a planner) with a combination of policies for the component spaces.

Another way to break up the problem is to reduce it. Lemma 7 allows us to optimize a planning problem by finding an embedding of a simpler space whose operators can be implemented by individual control policies. It may be that search is necessary to solve one or more of these subproblems, but if reactive systems can be found for the rest, then optimization may be quite useful. This is equivalent to the notion of planning in an abstraction space, which has a long history in AI (see Sacerdoti [109], Knoblock *et al.* [69], or Spector and Hendler [116]). It is also similar to the use of layering in distributed control architectures (see Brooks [24] or Spector and Hendler [116]).

Chapter 7

The core visual system

Nearly all perceptual processing in Polly is done by a small group of processes which I will call the core visual system (CVS). The core visual system is responsible for computing depth information, the direction of the corridor, detecting people, detecting motion, and sanity checking of the visual system's inputs. The CVS processes a 64×48 image every 66ms and generates a large number "percepts" (see table 7.1) which are updated continuously. Most percepts are related to navigation. The CVS is implemented in the Polly code by the procedures `low-level-vision` and `derived-aspects`, which call other procedures in turn, to compute the various intermediate quantities.

Figure 7-1 shows the principal components of the CVS. The CVS subsamples the image, smoothes it, and then passes it to several parallel computation streams. One stream (in grey) computes a depth map of the scene and compresses the map into a number of scalar values such as the distance to the closest object in the robot's path (`center-distance`) and whether there is a left turn in view (`left-turn?`). The computation of the depth map assumes the presence of a single, textureless carpet. The depth map computation will underestimate depth at the places in the lab where two different textureless carpets abut. The CVS compensates for this problem using a second, parallel, computation which checks whether the robot has reached the boundary between two carpets. When the carpet boundary detector finds a boundary, it instructs the depth map computation to compensate by ignoring the boundary. The CVS also computes the vanishing point of the lines forming the corridor. The depth map and vanishing point are used by the low level navigation system. The symmetry detector searches the image for tall skinny symmetric regions. When the robot is in a corridor, these regions will typically correspond to people's legs. The symmetry detector reports the direction of the most symmetric region (`person-direction`) and a binary value indicating whether the size and symmetry of the region are strong enough to be a person (`person?`). Finally, the motion unit computes the total change in the image from frame to frame. This is used to detect gestures. A pair of nod detectors were implemented and tested, but could not be used on the robot because of hardware problems (see 7.5).

Table 7.1 shows the suite of high level "percepts" which are generated by the CVS. Most percepts are generated from the depth map. `Blocked?` is true when the

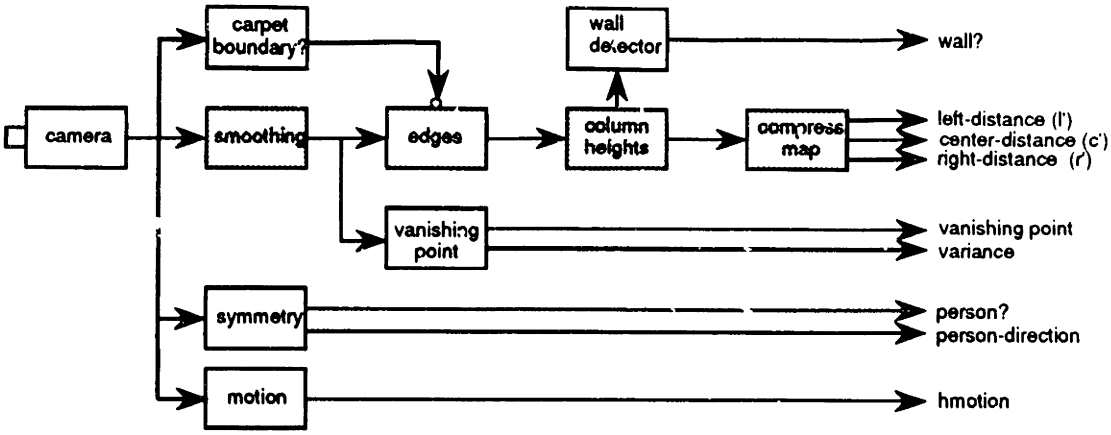


Figure 7-1: The major components of the core visual system (CVS).

`open-left?`	`open-region?`	`person-direction`
`open-right?`	`blind?`	`wall-ahead?`
`blocked?`	`light-floor?`	`wall-far-ahead?`
`left-turn?`	`dark-floor?`	`vanishing-point`
`right-turn?`	`person-ahead?`	`farthest-direction`

Table 7.1: Partial list of percepts generated by the visual system.

closest object ahead of the robot is nearer than a threshold distance (about three feet). `Open-left?` and `open-right?` are true when the there are no objects in the respective direction closer than a threshold distance. `Left-turn?` and `right-turn?` are true when `open-left/right?` is true and the robot is aligned with the corridor. Alignment with the corridor is actually reported by the low level navigation system. The LLN judges the robot to be aligned if the robot has driven straight for a sufficient period of time. It would be preferable to use visual data, but the robot cannot always determine the axis of the corridor from a single image: if the corridor is blocked, or the robot has reached the end of the corridor, there will be insufficient information to accurately judge the corridor's orientation. `Wall-ahead?` and `wall-far-ahead?` are true when there is a flat region ahead in the depth map. `Dark-floor` and `light-floor` are true when the intensity of the pixel at the bottom center of the image is above or below a given threshold. Both percepts are computed with hysteresis to compensate for transient variations in lighting or albedo.

7.1 Computation of depth

Since most of Polly's computation is devoted to navigation, a large fraction of its visual processing is devoted to finding the empty space around it. The central representation used for this purpose is the radial depth map (RDM). The RDM gives the distance to the nearest obstacle for each column of the image. Since image columns correspond to directions in the ground (X-Z) plane, the representation is equivalent to the sort of radial map which one would get from a high resolution sonar ring (hence the name).

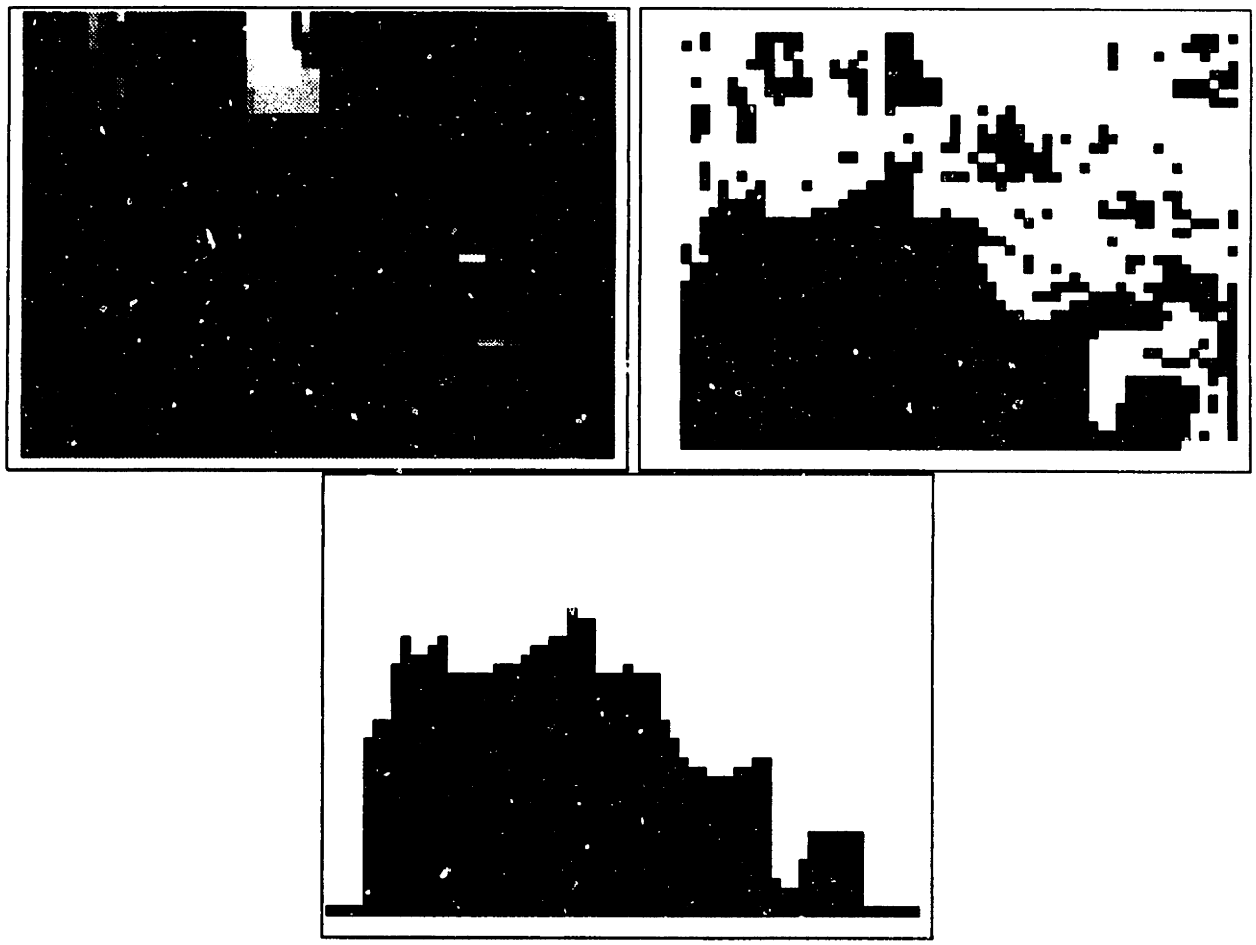
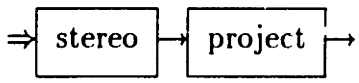


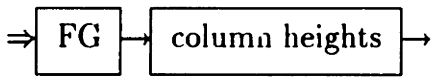
Figure 7-2: The robot computes a depth map by first labeling every pixel as being floor-like or not (above right) and then finding the image plane height of the lowest non-floor pixel in each column. The result is a monotonic measure of depth for each column (above center).

The central pipeline of the CVS (shown in grey in figure 7-1) computes depth information. The robot starts by finding the edges in the image. Then, for each column of the image, it computes the height of lowest edge pixel in that column. Under the right set of assumptions, this height will be a measure of the distance to the closest obstacle in view within that column (see section 5.4.2 and below). Figure 7-2 gives an example of the depth map calculation.

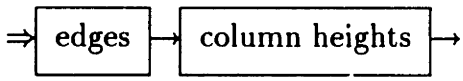
We can understand the assumptions made by this system and their ramifications by performing a derivation. Consider an arbitrary system for computing a radial depth map. For example, we might first use a stereo system to extract a 2D depth map, then collapse the 2D data into a radial map:



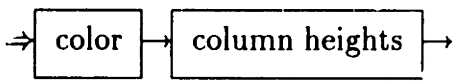
Such an approach might be quite effective, but it makes its own set of assumptions about the world (e.g. smoothness of surfaces or presence of dense texture). By lemma 2, p. 71, we can reduce any such system to



where "FG" is some computation which performs figure ground separation. Fortunately, Polly's environment also satisfies the background texture constraint because the carpet has no visible texture. By lemma 3, p. 68, we can use any linear filter whose pass band is restricted to the zero-band of the carpet to solve the figure-ground problem. An edge detector is such a filter, so we can reduce the system to



The derivation is summarized in table 7.2. The derivation shows that the the background texture constraint is used to to simplify figure ground separation. More importantly, it shows that the constraint is used for nothing else. If we wish to run the system in an environment which does not satisfy the background texture constraint, but which does satisfy the ground plane constraint, then we can use any salience constraint which holds of the domain. For example, if the background has a distinctive color or set of colors, then we can use a color system such as Swain's color histogram method [121] to find the carpet:



If we wanted to build a system which worked in both domains, then we could implement both the color system and the edge detector and switch between them opportunistically, provided there was sufficient information to determine which one to use. One could even implement the stereo system in parallel with these systems and add another switch.

The particular edge detector used by Polly is a thresholded gradient detector.

Constraint	Computational problem	Optimization
Ground plane	depth perception	use height
Background-texture	figure/ground separation	use texture

Table 7.2: Habitat constraints used for depth-recovery.

The detector was chosen because it compiled to a very short inner loop on the DSP. Because any edge detector should work (by lemma 3), we are free to make the choice based on computational efficiency. The exact test is for $|\frac{\partial I}{\partial x}(x,y)| + |\frac{\partial I}{\partial y}(x,y)| > 15$ after smoothing with a 3×3 low-pass filter to remove noise. The possible range of gradients is 0 to 510, so this is a very sensitive detector. To avoid driving into dark regions, where edge detection is unreliable (because the estimates of the intensity derivatives become noisy), the edge detector automatically marks any pixel darker than 40 grey levels. This helps prevent the robot from driving into dark rooms. The edge detector is implemented by the `find-dangerous` procedure, whose source code is shown in figure 7-3. The procedure is so named because its purpose is to label pixels which should not be driven into.

The CVS then compresses the depth map into three values—`left-distance`, `right-distance`, and `center-distance`—which give the closest distance on the left side of the image, right side, and the center third, respectively. Other values are then derived from these. For example, `open-left?` and `open-right?` are true when the corresponding distance is over threshold. `Left-turn?` and `right-turn?` are true when the depth map is open on the correct side and the robot is aligned with the corridor.

7.2 Detection of carpet boundaries

The background texture constraint fails at the boundaries between two carpets. The failure causes the edge detector to fire even though there is no object present, and so causes the robot to underestimate the freespace in front of the robot. The robot then thinks that it is blocked when it is not. Polly explicitly detects this condition and treats it as a special case. The procedure `carpet-boundary?` (shown in figure 7-4) checks for the presence of a carpet boundary. If it returns true, then `find-dangerous` will ignore horizontal edges. Since the carpet boundary is horizontal in the image when the robot approaches it, this is sufficient to cause the robot to pass the carpet boundary. Once it passes the boundary, `carpet-boundary?` returns false, and `find-dangerous` once again becomes sensitized to horizontal lines.

The actual test used by Polly is overly simplistic, but adequate for the job. The robot examines a 10×15 window at the bottom center of the image. It searches for a horizontal edge of medium contrast within the window. If there is such an edge, and no pixel is brighter than 120 grey levels, then it treats the image as a carpet boundary scene. The requirements that the edge have medium contrast and that not pixel be too bright prevent the system from classifying baseboards, which are dark next to a bright wall, as carpet boundaries. A more intelligent test would certainly

```
(define (find-dangerous image out)
  (let ((dark-threshold 40)
        (edge-threshold 15))
    (if suppress-horizontal
        (map-vector! out
                     (lambda (up left center)
                       (if (and (< (abs (- left center))
                                   edge-threshold)
                                (> center dark-threshold))
                           0 255))
                     (shift image (- *image-width*))
                     (shift image -1)
                      image)
      (map-vector! out
                   (lambda (up left center)
                     (if (and (< (+ (abs (- left center))
                                    (abs (- up center)))
                                 edge-threshold)
                              (> center dark-threshold))
                         0 255))
                   (shift image (- *image-width*))
                   (shift image -1)
                   image))))
```

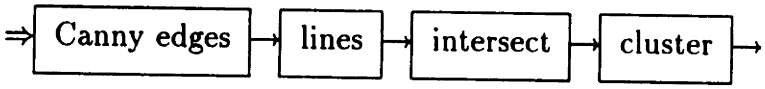
Figure 7-3: Source code for finding obstacle pixels. The code has been simplified by removing compiler declarations. **suppress-horizontal** is an wire set by the carpet boundary detector when it detects a horizontal boundary between two carpets. This causes the robot to ignore horizontal lines (see section 7.2).

be a good idea.

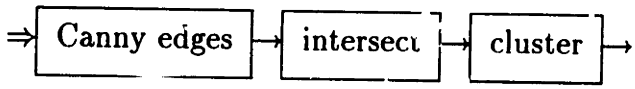
7.3 Vanishing-point detection

When Polly is in a corridor, the visual system also generates the x-coordinate of the vanishing point of the corridor. The vanishing point coincides with the axis of the corridor and so can be used for steering (this will be discussed further in section 8.2). The vanishing point computation works by finding each edge pixel in the image, fitting a line to the edge, intersecting it with the top of the screen, and then computing the mean of the intersections (see figure 7-5).

As with the other visual computations in Polly, the computation of the vanishing point is simplified by Polly's knowledge of the appearance of the environment. We can make this knowledge explicit by deriving Polly's vanishing point computation from a more general computation. I will start from the system of Bellutta *et al.* [19], which extracts vanishing points by running an edge finder, extracting straight line segments, and performing 2D clustering on the pairwise intersections of the edge segments. We can represent the system schematically as:

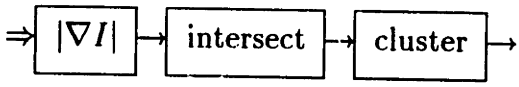


We can simplify the system by making stronger assumptions about the environment. We can remove the step of grouping edge pixels into segments by treating each edge pixel as its own tiny segment:

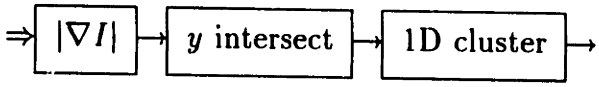


This will weight longer lines more strongly, so the lines of the corridor must dominate the scene for this to work properly.

If the edges are strong, then any edge detector will suffice. Polly uses a gradient threshold detector simply because it compiles to very efficient code:



Here I is the image, a function from position to intensity values, ∇I is the spatial gradient of the image, and so $|\nabla I|$ is the magnitude of the gradient of the image. If the tilt-angle of the camera is held constant by the camera mount, then the vanishing point will always have the same y coordinate, so we can reduce the clustering to a 1D problem.



Finally, if we assume that the positions and orientations of the non-corridor edges are uniformly distributed, then their y-intersections will have zero mean. If we replace the clustering operation, which looks for modes, with the mean of the y-intersections,

```
(define (carpet-boundary? image)
  (with-vars-in-registers
    (let ((edges 0)
          (edge-thresh 9)
          (bad-thresh 25)
          (region-width 10)
          (region-height 15)
          (top-line -1)
          (bad-edges 0)
          (bottom-line 0))
      (let ((im (shift image 1947)))
        ;; im points to row 30, pixel 27 of image.
        (countdown (lines region-height)
            ;; Process a line.
            (countdown (pixel 10)
              ;; Process a pixel; check intensity and gradient.
              (let ((center (in-register data (vector-ref im 0))))
                (let ((delta (abs (- (vector-ref im 64) center))))
                  (when (> center 120)
                    (incf bad-edges))
                  (when (> delta bad-thresh)
                    (incf bad-edges))
                  (when (> delta edge-thresh)
                    (incf edges)
                    (when (< top-line 0)
                      (set! top-line lines))
                    (when (> lines bottom-line)
                      (set! bottom-line lines)))
                  (shiftf im 1 ))))
            ;; Move to next line.
            (shiftf im 54)))
        (and (> edges 7)
             (< edges 30)
             (= bad-edges 0)
             (< (- bottom-line top-line) 7)))))
```

Figure 7-4: Source code for the carpet boundary detector. Note that `countdown` is the same as **dotimes** in CommonLisp, except that it counts backward.

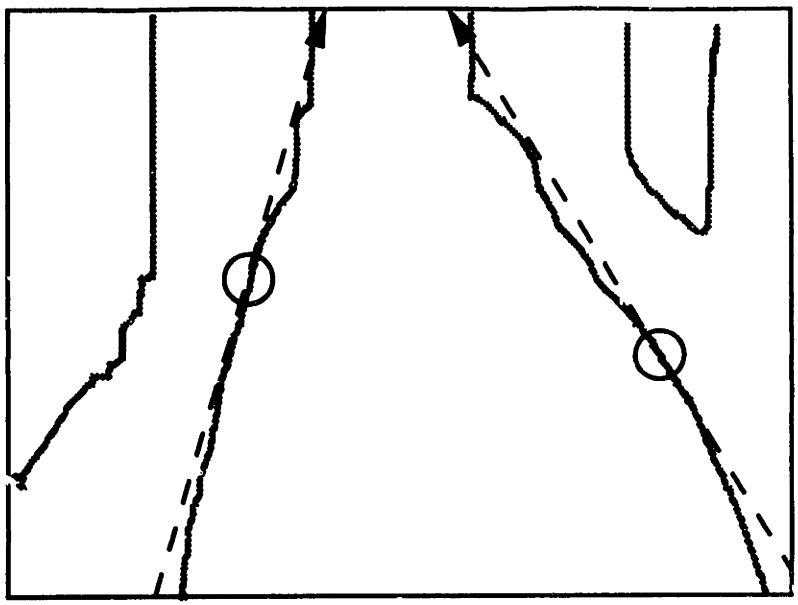
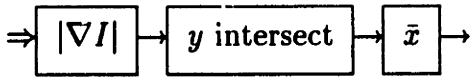


Figure 7-5: The vanishing point computation: edge fragments at individual pixels (shown in circles) are extended (dashed lines) and intersected with the top of the image to find the horizontal location of their intersections (arrowheads). The mean of the horizontal locations is used as the vanishing point.

Constraint	Computational problem	Optimization
Long corridor edges	line finding	use pixels as lines
Strong corridor edges	edge detection	use cheap detector
Known camera tilt	clustering	1D clustering
Uniform non-corridor intersections	clustering	use mean

Table 7.3: Habitat constraints used by the vanishing point computation.

the result will be a weighted sum of the means of the corridor and non-corridor edges. Since the mean of the latter is zero, the result will be the mean of the corridor edges multiplied by an unknown scale factor which will depend on the ratio of corridor to non-corridor edges. Thus, while the result will typically underestimate the magnitude of the vanishing point, it will get its sign right. As we will see in the next chapter, that will be sufficient for our purposes. The resulting system is thus:



The derivation is summarized in table 7.3.

The CVS also reports the variance of the y-intersections as a confidence measure. This entire computation is performed in the Polly code by the procedure **vanishing-point**. The procedure computes the mean and variance directly in a single pass over the input array. The procedure contains its own edge detector which is tuned to find diagonal edges (see figure 7-6).

```
(define (find-vanishing-point image)
  ;; Start at the end of the 45th line of the image.
  (let ((image (shift image (- (* 46 *image-width*)
                               2)))
        (sum 0)
        (sum-squares 0)
        (points 0))
    (countdown (y 45)
      ;;; Scan a line.
      (countdown (x (- *image-width* 1))
        (let* ((dx (- (vector-ref image 1)
                      (vector-ref image 0)))
               (dy (- (vector-ref image *image-width*)
                      (vector-ref image 0))))
          (when (and (> (abs dx) 10)
                     (> (abs dy) 10))
            ;; We have a reasonable edge point.
            (let ((x-intercept (+ (quotient (* y dy)
                                            dx)
                                  x)))
              (when (and (> x-intercept -1)
                         (< x-intercept 64))
                (set! sum (+ sum x-intercept))
                (set! sum-squares (+ sum-squares
                                     (* x-intercept x-intercept)))
                (set! points (+ points 1))))))
          ;; Next pixel.
          (set! image (shift image -1)))
        ;; Skip over the pixel at the beginning of the line.
        (set! image (shift image -1)))
    ;; Done with the image.  How compure the mean and variance.
    (if (and (> points 20)
             (< points 256))
        (let ((mean (quotient sum points)))
          (let ((variance (- (quotient sum-squares points)
                             (* mean mean))))
            (make-pair mean variance)))
        (make-pair 31 1000))))
```

Figure 7-6: Source code for the vanishing-point computation. The code walks over the image, from bottom to top, end of line to beginning, finding all the reasonable edges and computing the mean and variance of their y intersections. The mean and variance are returned as a pair (two 16-bit quantities in a 32-bit word). If there are too many or too few edges, the code gives up and returns a large variance. Note that the code has to skip the first pixel of every line because there is no pixel to the left of it.

7.4 Person detection

In order to offer tours to people, Polly needs to detect their presence as it moves through the corridors of the lab. At present, the robot can only look downward, so it can only see people's legs. Thus person detection reduces to leg detection. Polly relies on the fact that people generally stand, and so it need only search for straight, vertical legs, which appear as tall, narrow symmetric objects. Symmetric objects are found in two passes. First, it searches for tall, skinny symmetric regions of the image. Then, it tests the most symmetric region to see if it is a distinct object.

7.4.1 Symmetry detection

This is performed by the Scheme procedure `find-symmetric`. It is a simplified version of the technique used by Reisfeld *et al.* [104]. The simplest measure of symmetry about the vertical axis at a point (x, y) is:

$$-\int_0^l \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)dr$$

where l is the width of the region being searched for symmetry. This will give a large value if the pixels $(x+r,y)$ and $(x-r,y)$ tend to have significant horizontal intensity gradients with opposite signs. Thus both a bright region in front of a light region and a dark region in front of a light region will have positive scores, but a homogeneous region will have zero score and a region with a ramp-shaped intensity will have a negative score.

One disadvantage of this measure is that it is linear. If we superimpose a non-symmetric pattern on a symmetric one, their scores will cancel one another. This is a problem because the edges of a leg may be symmetric, while the surface markings of the pant leg may not be. We can alleviate this problem to some extent by only counting pixel pairs which are symmetric. We can do this by adding a min to the integrand:

$$-\int_0^l \min\left(0, \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)\right)dr$$

Another problem with the linearity of the computation is that a single pair of strong edges can generate a huge symmetry value for an otherwise non-symmetric region. This can be alleviated by adding a compressive nonlinearity to the integrand. Reisfeld *et al.* [104] used a log function for the nonlinearity. Polly uses a max because it is somewhat faster to compute:

$$-\int_0^l \max\left(-\alpha, \min\left(0, \frac{\partial I}{\partial x}(x+r,y)\frac{\partial I}{\partial x}(x-r,y)\right)\right)dr$$

Here α is the maximal symmetry value which the system will give to a single pixel pair.

Having scored all pixels for the vertical symmetry of the local regions centered around them, we can then find vertical lines about which the image is strongly sym-

metric by integrating the pixel scores along columns of the image. This is also done by `find-symmetric`. The result is then a vector of symmetry values for each column defined by:

$$s(x) = \sum_{y} \sum_{r=i}^{l} \max\left(-\alpha, \min\left(0, \frac{\partial I}{\partial x}(x+r, y)\frac{\partial I}{\partial x}(x-r, y)\right)\right)$$

where the derivatives are estimated as differences of adjacent pixels.

7.4.2 The protrusion test

Having tested the different image columns for symmetry, the visual system still needs to distinguish between objects which are symmetric and subregions of objects which are symmetric. Since Polly is specifically looking for people who are standing in hallways, and such people will show up as distinct bulges in the radial depth map, the visual system requires that a symmetric region align with a protrusion in the depth map. This test is performed by the procedure `protrusion-near?` which tests the depth map for a negative depth edge followed by a positive depth edge within a specific width around the region. If such edges are found, then there is such a protrusion and the visual system asserts the `person?` signal and drives the `person-direction` wire with the x-coordinate of the column with maximal symmetry.

7.4.3 Reliability

The protrusion test is not actually a sufficient test to guarantee that a given region is a person's leg. Any tall, skinny, vertical object, such as a chair leg, will pass the protrusion test. Thus the person detector can generate false positives. The pick-up system filters these false positives by only picking up people who are in hallways. That is, it will only trust the `person?` signal when the `corridor?` signal is also asserted.

The protrusion test also generates false negatives, but these are generally features. If a person is standing against the wall, the protrusion detector will generally not be sensitive enough to detect them and so it will not fire. Fortunately, this is useful for our purposes since we only want the robot to offer tours to people who actually approach the robot.

7.5 Gesture interpretation

When Polly offers a visitor a tour, it needs to be able to receive an answer. Since it cannot head, it looks for a particular gesture. Originally, this was to be a nod of the head. I built two working nod detectors which ran in real time on a macintosh using a static camera. In the end, however, I was forced to use waves of the foot instead of nods. This was essentially for hardware reasons. The robot is so short that the upward-looking camera saw only the bottom of the visitor's chin and so thus nods ended up being Z-axis oscillations rather than Y-axis oscillations. Z-axis oscillations

are much more difficult to measure. Also, because of the wide angle of the lens, and the huge variation in depth, it was very difficult to design a motion detector which could accurately measure the motion of a 4-pixel wide head moving a quarter pixel per frame and also accurately measure the motion of a 40 pixel wide arm moving at 30 pixels per frame. The upward looking camera also has had to cope with a great deal of glare from the overhead lights. This glare made it very difficult to properly image the face. Finally, the upward-looking camera broke late in the design of the system so it was decided to give up on the nod detector and use the wave detector.

7.5.1 Foot waving

Detecting foot waves turned out to be a trivial problem because anyone who was actually interacting with the robot was standing still. Further, there is typically no other motion in the scene. Thus the only motion in the scene is intentional shaking of the leg and so a simple motion detector sufficed as a wave-of-the-leg detector (in contrast to the various nod detectors). The motion detector used simply measures total image change from frame to frame:

$$m_{\text{total}}(t) = \sum_{x,y} |I(x,y,t) - I(x,y,t-1)|$$

and applies a low-pass filter to the resulting series m_{total}.

The low-pass filtered motion is computed by the procedure `tctal-hmotion` and its instantaneous value is store in `hmotion`.

7.5.2 The first nod detector

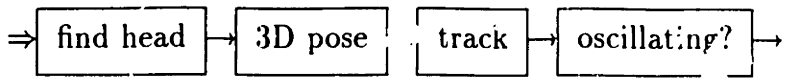
Detecting nods is a more difficult problem. Suppose that there is a person facing the camera. For simplicity, we will assume orthographic projection so that a point $(x,y,z)^T$ in the world is projected to a point $(x,y)^T$ in the image, and points higher in the image have larger y coordinates. Actual camera projections are more complicated, but these considerations are unimportant for our purposes (see Ballard and Brown [12] or Horn [60] for introductions to the mathematics of camera projection). Finally, we will assume that the objects in view are rigid or at least piecewise rigid. We will treat the motion of a given object at a given point in time as a combination of a translation $\mathbf{T}$ and a rotation ω about a point $\mathbf{P}$.[1] A point $\mathbf{R}$ on the object thus moves with an instantaneous velocity of

$$\mathbf{T} + (\mathbf{R} - \mathbf{P}) \times \omega$$

Since a nod is an alternating rotation of the head, a nod is then a motion of the head in which ω maintains a constant direction, but a varying velocity, $\mathbf{R}$ remains constant, and $\mathbf{T}$ remains zero. The conceptually simplest approach to detecting nods

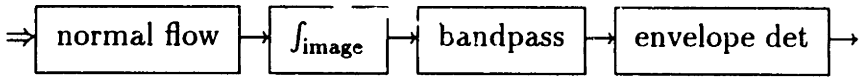
[1] This is a gross oversimplification of the kinematics of jointed objects, but it will suffice for our purposes.

would be to first find the head in the image, then determine its pose (position and orientation in 3-space), and then to track the pose through time to recover the motion parameters **T**, **R**, **P**, and ω, and then test these parameters for rotational oscillation:



This system would be expensive and difficult to implement, although it would certainly be conceptually possible. The main problems are with finding faces and determining the pose. A particular problem with the latter is that it is easy to confuse translation with a rotation. In particular, it is very difficult to distinguish the cases of my bending (rotating) my head downward, and my head moving down 3cm, but still pointing forward. Both would involve the points of my facing moving downward along the Y axis, while staying at the same point along the X axis. Of course, the translation would require my neck to suddenly grow 3cm shorter or for me to do deep knee bends.

An alternative system which is much simpler is the following:



First we compute the vertical component of *normal flow field* (see below) of the image, then we integrate the field over the entire image to obtain a single net vertical motion, and then we look for oscillations in the net vertical motion. The optic flow of an image is the apparent two-dimensional motion of the texture in the image. It is the projection into the image plane of the motion vectors of the actual objects in the world. Since the 3-space motion of a point **R** is simply

$$\mathbf{T} + (\mathbf{R} - \mathbf{P}) \times \omega$$

the optic flow of **R**'s projection is simply the x and y coordinates of this vector, or

$$\mathbf{f_R} = \begin{bmatrix} T_x + (R_y - P_y)\omega_z - (r_z - p_z)\omega_y \\ T_y + (R_z - P_z)\omega_x - (R_x - P_x)\omega_z \end{bmatrix}$$

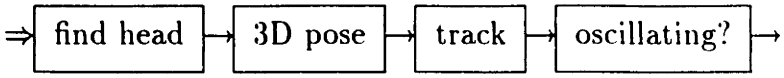
The normal flow field is the component of the optic flow field in the direction of the image's intensity gradient. The useful property of the normal flow field is that it is very easy to compute. If we let $I(x, y, t)$ represent the brightness of a point (x, y) in the image, at time t, then the vertical component of the normal flow field is given by

$$-\frac{\frac{\partial I}{\partial t}(x, y, t)}{\frac{\partial I}{\partial y}(x, y, t)}$$

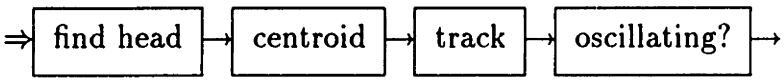
Since the derivatives can be approximated by subtracting adjacent points in time or space, respectively, we can compute this value very quickly.

We can explain why (and when) this system works by performing a derivation.

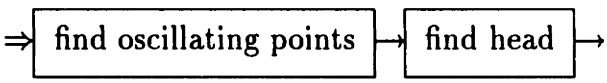
We start from the original system:



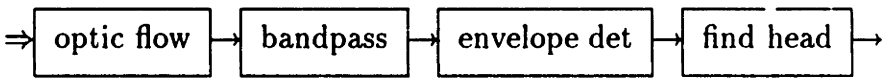
Recall that computing the 3D pose is difficult, since translations and rotations both generate vertical motions in the image. However, since the head is not physically capable of translating, any up/down motion of the head in the image *must* be due to a rotation, unless the person is bending their knees, jumping up and down, bowing rapidly, or standing on an oscillating platform. If we assume that people don't do these things, then we may safely interpret any vertical oscillation of a head as a nod. Let's call this the *head kinematics constraint*: that the translation vector **T** of the head is nearly always zero, and that it never oscillates. The head kinematics constraint allows us to use the 2D position of the head (*i.e.* the position of the head in the image itself), instead of its 3D pose. An oscillation of the 2D position will indicate a nod. Thus we can replace the "3D pose" module with a module which finds the centroid of the image region occupied by the head:



Thus we've reduced the problem to finding 2D oscillations of the head. Finding the head can be difficult in itself however. Fortunately, we're not looking for arbitrary heads, only oscillating ones. Thus we need not bother looking in parts of the image which aren't moving. Indeed, if we assume that vertical oscillations are generally uncommon, *i.e.* people don't nod their hands or play with yo-yos, then we can use the oscillatory motion itself to find the head. We'll call this the *motion salience constraint*: no motion parameter of any object is allowed to oscillate, save for the ω parameter of a head. The motion salience constraint removes the need for a full recognition engine and allows us to use a system like:



First we find the pixels in the image which are oscillating, and then we look for a head-shaped region of oscillating pixels. We can find the oscillating points by first computing the 2D motion (optic flow) at each point in the image, and then testing the motion for vertical oscillation. The test can be done, in turn, by applying a bandpass filter and an envelope detector at each point. The resulting system is thus:

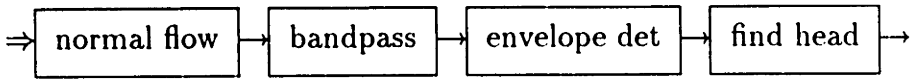


Computing optical flow can still be fairly expensive however. The normal flow field is much easier to compute and, fortunately, its vertical component will always have the same sign as the vertical component of the optical flow, provided that the normal flow is non-zero. The normal flow will be zero when the actual motion is nonzero only if there is no texture in the image at that point, or if the y (vertical)

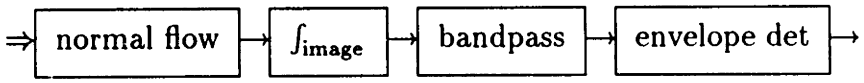
Constraint	Problem	Optimization
Head kinematics	motion disambiguation	use 2D motion
Motion salience	head detection	use oscillation
Horizontal lines	flow computation	substitute normal flow
Nod dominance	head detection	use net flow

Table 7.4: Constraints used in the derivation of the first nod detector, and the problems they were used to simplify.

derivative of the image intensity is zero. Fortunately, faces have considerable vertical intensity variation and so this is not a problem. Thus we can reduce the system to



Even looking for oscillations at each point might be too expensive, however. If we assume that the motion of the head will dominate any other motion in the image, then we can look for oscillations in the net vertical motion of the entire image

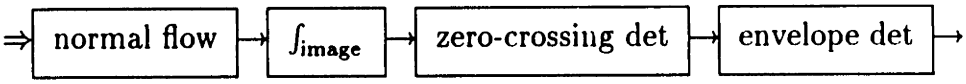


which is the system we were trying to derive. The assumption that the head motion dominates the motion in the rest of the image is needed to rule out the case of motions in different parts of the image canceling with one another to generate the appearance of oscillatory motion when in fact, no single part of the image was oscillating.

The derivation is summarized in table 7.4. I have implemented this system on a Macintosh personal computer. The system uses 64×48 grey-scale images at a rate of approximately 5 frames per second. The system performs reliably provided that the subject nods their head several times to give the bandpass filter a chance to respond. Unfortunately, this can lead to dizziness if it is used frequently, so a different system was needed.

7.5.3 The second nod detector

The first nod detector was not adequate for general use because the bandpass filter required many cycles to decide that a head nod was really happening. This is a well known problem with linear filters: the narrower the pass band, the longer it takes the system to respond. A non-linear oscillation detector such as a zero-crossing detector would probably do better. The result would be something like this:

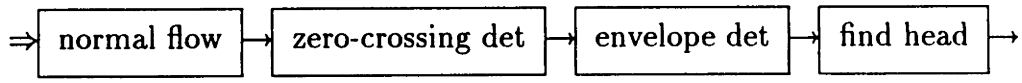


However we would expect the net flow to have a relatively large number of random zero crossings, even when the head is not moving. We can deal with this problem by backing off of the net flow optimization and returning to computing the oscillations

Constraint	Computational problem	Optimization
Ground plane	depth perception	use height
Background-texture	figure/ground separation	use texture
Long corridor edges	line finding	use edge pixels
Strong corridor edges	edge detection	use cheap detector
Known camera tilt	clustering	1D clustering
Uniform intersections	clustering	use mean
Head kinematics	motion disambiguation	use 2D motion
Motion salience	head detection	use oscillation
Horizontal lines	flow computation	use normal flow
Nod dominance	head detection	use net flow

Table 7.5: Habitat constraints assumed by the core visual system and the problems they helped to simplify. Note that "known camera tilt" is more a constraint on the agent, than on the habitat.

on a per-pixel basis:



The flow and zero-crossing detectors have been implemented on Polly. Unfortunately, the robot is so short, and the viewing angle of the camera so wide, that it can only see the bottom of a person's chin, so it has been difficult to get the flow detector to respond to any head movements, much less to detect nods.

7.6 Summary

Polly's vision system uses a number of parallel processes to efficiently compute just the information which is needed for its task. These processes are made more efficient by taking advantage of the special properties of the environment. We can understand the use of these properties by deriving the systems from more general ones. The complete list of constraints used in the derivations is given in table 7.5.

Chapter 8

Low level navigation

The LLN is the bottom layer of the robot's control system. It consists of a speed (forward-velocity) controller, an open-loop (ballistic) turn controller, a corridor follower, a wall follower, and a small amount of code to arbitrate between them (see figure 8-1). These systems are controlled by the signals `speed-request`, `turn-request`, and `inhibit-all-motion?`. `Speed-request` controls the maximum forward velocity and has no effect on steering, other than to disable the wall follower and corridor follower when it is zero. When `turn-request` is zero (no turn requested), the corridor follower (or wall follower if only one wall is visible) has complete control of steering. When `turn-request` is driven with a desired turn angle, the open-loop turn controller issues a turn command to the base and inhibits all steering for a time proportional to the size of the turn. `Inhibit-all-motion?`, when asserted, inhibits all parts of the LLN and forces the robot to stand still. `Inhibit-all-motion?` is asserted during visitor pickup (see section 9.7) to prevent the low-level navigation system from avoiding the visitor.

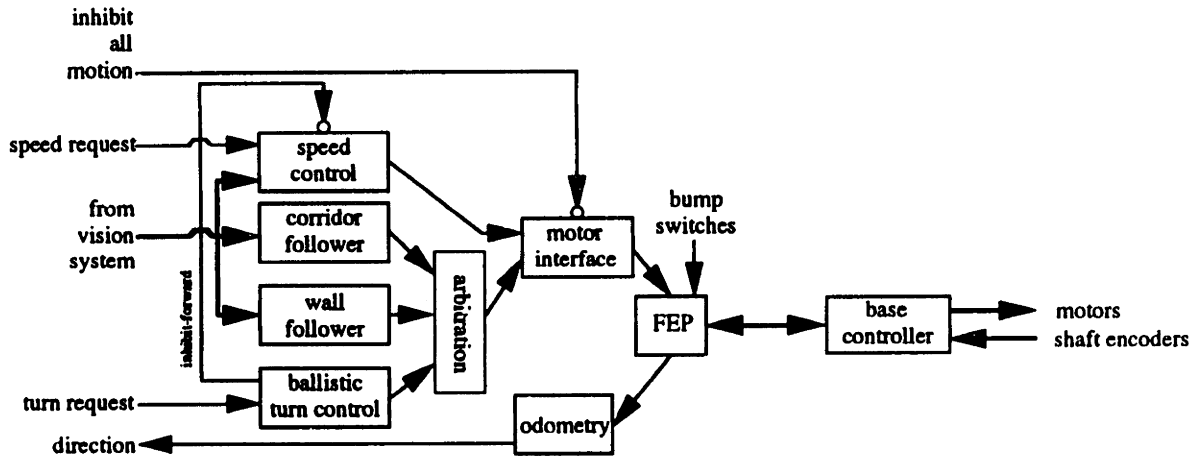


Figure 8-1: Components of the low level navigation system (LLN).

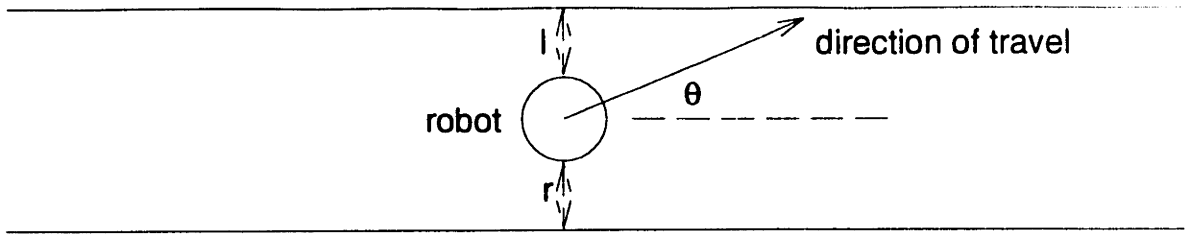


Figure 8-2: The corridor following problem. The robot needs to turn so as to simultaneously keep l and r large and θ small.

8.1 Speed controller

The robot's rate of forward motion is controlled by the procedure `speed-control`. The two constraints on forward velocity are that the robot should move forward when there's nothing in its way, and that it should stop when there is something close to it. We want the robot to move at full speed when the nearest obstacle is more than some safe distance, d_{safe}, and to stop when it's less than some distance d_{stop}. We use the rule

$$speed = \min\left(v_{max}, \frac{v_{max}}{d_{safe} - d_{stop}}(d_{center} - d_{stop})\right)$$

where d_{center} is the distance to the closest object in the center of the image (the **`center-distance`** output of the visual system). The robot smoothly decelerates as it approaches an obstacle and will back up if it gets too close to the obstacle. Backing up is useful because it allows one to herd the robot from place to place. The maximum speed, v_{max}, is set by the input **`speed-request`**.

The speed controller is modulated by two other inputs. If **`inhibit-forward?`** is asserted, then the robot will only drive backward. **`Inhibit-forward?`** is driven by the ballistic turn controller (see section 8.5). When **`inhibit-all-motion?`** is set, then all motors are disabled, and the robot will not even back up from approaching threats.

8.2 Corridor follower

Corridor following consists of two basic subproblems: aligning with the axis of the corridor and staying comfortably far from the walls on either side. In figure 8-2, these problems correspond to keeping θ small and l and r large, respectively. The robot base gives us control of the turn rate ($\frac{d\theta}{dt}$) and the speed (s) and so the variables l, r, and θ are coupled in the kinematics of the base:

$$\frac{dl}{dt} = -\frac{dr}{dt} = s\sin\theta$$

and so we cannot change l or r without first making θ nonzero. Polly's corridor follower uses separate control systems for wall distance and θ and sums their outputs.

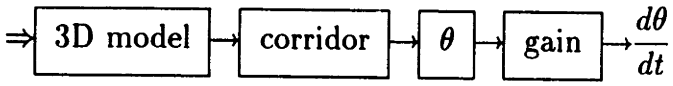
8.2.1 Aligning with the corridor

Polly uses a trivial algorithm for aligning with the corridor. It uses the control law

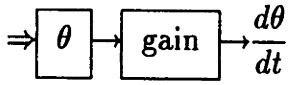
$$\frac{d\theta}{dt} = -\alpha\left(\mathrm{vp}_x - \frac{\mathrm{width}}{2}\right)$$

where vp_x is the core vision system's estimate of the vanishing point (carried in the variable `vanishing-point`), α is a gain parameter, and "width" is the width of the image in pixels. The vanishing point needs to be biased by width/2 because the coordinate system of the image is centered on the left-hand side of the image, not in the middle.

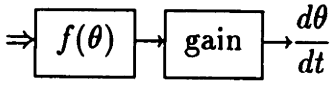
An obvious and direct way of performing this task would be to first construct a 3D model of the environment, then find the walls of the corridor in the model and compute θ, and finally multiply θ by some gain to drive the turning motor. We can represent this schematically as:



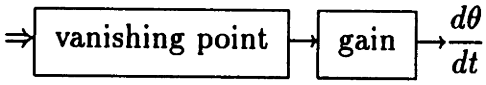
This is not a particularly efficient design however, since 3D models are both difficult and computationally expensive to build. Intuitively, building an entire model of the environment, only to compress it down to a single number, θ, seems a waste of energy. Any system which computes θ, in whatever manner, and then turns so as to minimize θ, that is, any system of the form,



will work. By decalibration (lemma 1, p. 57), we can substitute *any monotonic function of* θ for our estimate of θ, provided that we get zero right. Thus we can use any system of the form:



where f is a monotonic function for which $f(0) = 0$. Since $\mathrm{vp}_x - \frac{\mathrm{width}}{2}$ is such a function, we can use it, and reduce the above system to



which is the system that Polly uses.

8.2.2 Avoiding the walls

One way to avoid the walls is to drive away from the wall that looks closer. If we let l' be the distance to the closest point in view on the left, and r' the distance to the closest point in view on the right (see figure 8-3), then we might steer so as to make

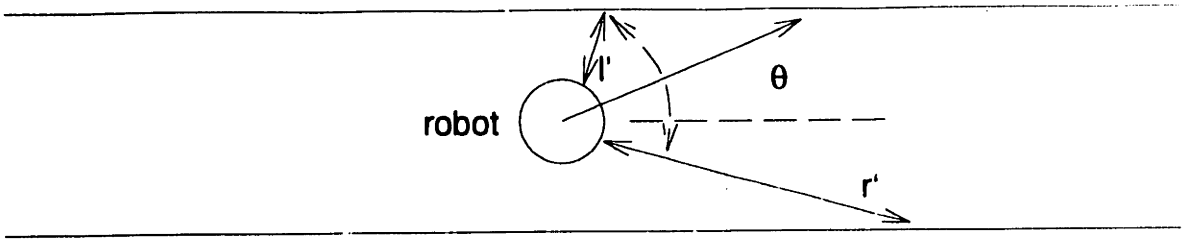


Figure 8-3: The nearest points in view of the robot. The dashed curved line indicates the robot's field of view. l' and r' are the distances to the nearest points in view on the left and right walls, respectively. Note that as the robot rotates clockwise in place, l' gets larger and r' gets smaller.

l' and r' be equal.. Polly does this using the control law

$$\frac{d\theta}{dt} = -\beta(\texttt{left-distance} - \texttt{right-distance}) \tag{8.1}$$

where β is a gain parameter. Recall however, that `left-distance` and `right--distance` are not the same as l' and r', rather they are equal to $f(l')$ and $f(r')$ for some unknown monotonic f. We would like to show that this doesn't matter. Unfortunately, we can't use lemma 1 directly, because it only allows us to replace $l' - r'$ with $f(l' - r')$, *not* $f(l') - f(r')$. Fortunately, for any given position of the robot, l' will be a strictly increasing function function of θ, while r' will be a strictly decreasing function of θ. Thus their difference is also strictly increasing. But then $f(l') - f(r')$ must also be a strictly increasing function of θ. Moreover, $f(l') - f(r')$ is zero exactly when $l' - r'$ is zero. Thus both $l' - r'$ and $f(l') - f(r')$ are effectively decalibrated versions of θ, and both will converge to the same value of θ for a given position in the corridor.[1]

8.2.3 Integrating the control signals

As was mentioned above, the wall-follower sums the control signals to align with the corridor and avoid the walls, yielding the final control law:

$$\frac{d\theta}{dt} = -\alpha\left(\texttt{vanishing-point} - \frac{\text{width}}{2}\right) - \beta\,(\texttt{left-distance} - \texttt{right-distance})$$

The integration is modulated by the variance of the vanishing point and the presence of obstacles. If the variance of the estimate of the vanishing point is high then α is set to zero. This prevents the robot from steering in odd directions or oscillating when the robot enters non-corridor areas. When the robot is blocked by an obstacle, it will make all steering turns at full speed. This allows the robot to quickly turn around

[1]This reasoning assumes the robot's field of view is wholly to the right of the robot in the figure, and that both walls are in the field of view. For very large or small fields of view, or for large values of θ, this may not be true.

the obstacle without causing stability problems at other times.

8.3 Wall follower

When the robot enters an open room or a very wide corridor, it will be unable to see the opposite wall. In such situations, the corridor follower would treat the opposite wall as being at infinity but would still try to balance the distances of the two walls. Thus it would drive until *neither* wall could be seen and then continue on in a straight path. The robot then has no idea where it is going. The robot has a separate control system for this case which turns to keep the wall that is in view at a fixed distance. The control law is then simply

$$\frac{d\theta}{dt} = -\gamma(f(l') - d_0)$$

for the case where the left wall is in view, or

$$\frac{d\theta}{dt} = \gamma(f(r') - d_0)$$

when the right wall is in view. Here γ is a gain parameter and d_0 is the desired (decalibrated) distance. Again, $f(l')$ and $f(r')$ are stored in the `left-distance` and `right-distance` variables.

8.4 General obstacle avoidance

There is no general obstacle avoidance routine in the system. However, both the corridor follower and the wall follower work to control the distance to the nearest thing on either side. In the normal case, the nearest things on either side will be the walls. However, when there is an obstacle in the way, the obstacle will be the nearest thing on one or both sides and the robot will avoid it. This is a local navigation strategy equivalent to the method of artificial potential fields (see Khatib [68]). It has the advantage of being fast and easy to compute and the disadvantage that the left and right distances can sometimes balance exactly, even when then robot is blocked by an obstacle. Such cases are dealt with by the unwedger unit in the high level navigation system (see section 9.3.2).

8.5 Ballistic turn controller

When the robot reaches a junction and needs to switch from one corridor to another, it does this by issuing an open-loop turn command to the base. This command is issued by the ballistic turn controller whenever the input line `turn-request` is non-zero. `Turn-request` is normally held low by the ballistic turn controller, but can be driven with a specific value (a number of degrees to turn) by the high level navigation system to force a turn. After issuing an open loop turn, the controller

maintains control of the turning motors for a time proportional to the size of the turn. This gives the microcontroller in the base time to servo to the correct direction and decelerate.

One potential problem with ballistic turns is that the robot can turn to face a wall. If the wall is textureless and completely fills the visual field of the robot, then it will appear to be an empty field and the robot will happily try to drive into it. In theory, the speed controller should back away from the wall as it turns toward it. In practice, it is possible for the robot to turn so fast that the speed controller literally never sees the wall coming. To prevent this, the ballistic turn controller asserts the signal `inhibit-forward?` during large turns. This prevents the speed controller from ever driving forward in the first place.

8.6 Steering arbitration

Arbitration is performed by the procedure `turn-controller`. The turn controller polls the ballistic turn controller to determine whether a ballistic turn is in progress. If so, it will inhibit the other control systems and allow the ballistic turn controller to finish its turn. If the ballistic controller is inactive, then the turn controller will use either the control signal from the corridor follower, if both walls are in view (both `open-left?` and `open-right?` false), or the wall follower, if only one wall is in view. As with the speed controller, the turn controller will inhibit all control systems if the input `inhibit-all-motion?` is asserted.

8.7 The FEP bump reflex

In addition to the visual collision avoidance implemented in the DSP, the FEP (the 6811 front-end processor) also monitors a pair of bump switches. If either switch closes, the FEP immediately performs a fixed sequence of actions: first it halts the base, then it reverses the base, and finally, it issues a 45 degree open-loop turn away from the activated bumper. During this time, the DSP is disconnected from the base. When the bump switch opens, the FEP resumes normal processing and allows the DSP to drive the base.

The bump reflex was implemented in the FEP because processing latency is a critical issue. Polly's normal top speed is 1m/s. Since the DSP only samples its sensors at 15Hz, the robot can move up to 7cm between samples. The bump switches extend roughly 12cm from the base and require 1cm of travel to trigger. Even at top speed, the DSP could not initiate a braking action until there were only 4cm between the robot and the obstacle. There is no way the robot can possibly brake in 4cm. Fortunately, the FEP can sample the bump switches at approximately 1kHz, and initiate braking actions immediately upon bumper contact. In practice, even this is insufficient however, and so the robot often collides with obstacles, albeit at lower velocity. While the base is physically capable of decelerating fast enough to stop, the required deceleration will cause the robot to topple. This is not a pleasant experience for either robot or owner.

8.8 Odometric sensing

The only other sensing on the robot is odometric. The RWI base provides high resolution shaft encoders for dead reckoning turns and forward motions. The vast majority of this information is ignored by the current system. The inherent unreliability of odometry was one reason for its limited use. In the course of driving from one end of the building to the other, the rotational odometer can drift by as much as 45 degrees, even when the base is reasonably well aligned. Another reason was simply that I was more interested in vision than odometry and so chose to spend my time engineering the vision side.

All that said, Polly does use the rotational shaft encoder to determine direction. The robot assumes that it starts out pointed south and so the first shaft encoder reading, call it ω_0, it receives will be a south reading. It can then compute its rotation relative to that direction by taking the difference of its current reading and ω_0. Because of encoder drift, the robot only computes either orientation to within 90 degrees. It then drives the wire `direction` with one of the values `north`, `south`, `east`, or `west`. Odometer drift can still cause the robot to misestimate its direction however, so the system recalibrates itself whenever possible. Whenever the robot determines that it is in a long corridor and the visual system reports that it is aligned with the axis of the corridor, then the robot assumes that it is exactly aligned with one of the compass points, and recomputes ω_0. This has proven very effective for compensating for drift.

Chapter 9

High level navigation

Navigation is the central problem of current mobile robot research. If you have a robot that can drive from place to place, then the first thing you probably want it to do is to drive to a particular place. The problem remains largely unsolved. Navigation suffers from many of the same problems of intractability and sensitivity to error that other AI problems suffer from. In this chapter, I will discuss how Polly manages to navigate reasonably reliably and efficiently. Polly navigates in a particularly benign, but unmodified, environment. As with most systems in Polly, what's interesting is not it's precise navigation algorithm, but the basic structures of the environment which allow it to function. I will begin by discussing a fairly conventional formalization of the navigation problem and some of the reasons for its difficulty. Then I will add progressively more structure to the problem and discuss how that structure allows the use of simpler mechanisms. In doing so, I will derive a very simple, idealized, navigation algorithm for a class of worlds to which the MIT AI Laboratory belongs. Finally, I will give the details of Polly's navigation algorithm and describe how it does and does not implement the idealized algorithm.

9.1 Formal problem definition

Navigation is often formalized as the problem of choosing a path from one point in $\mathcal{R}^n$ to another given some restrictions on the intermediate positions and orientations which are allowed. This is clearly not the problem which humans solve. While humans definitely have some sense of "where they are", that sense is not a pair of Cartesian coordinates. Nor is a person's goal generally a pair of Cartesian coordinates—my goal is usually a room or a building, or, perhaps, a chair. Such places simply are not defined by a particular point in Cartesian space. At best, they can be put in correspondence with a region of Cartesian space. Polly's navigation problem is to get from wherever it is at the moment to some specified place (landmark) which can be perceptually discerned.

That said, I will proceed to formalize the problem in terms of $\mathcal{R}^n$ in the traditional way. The reasons for this are twofold: it allows clarity and precision, and it is a very general framework in which we can easily express the constraints and optimizations

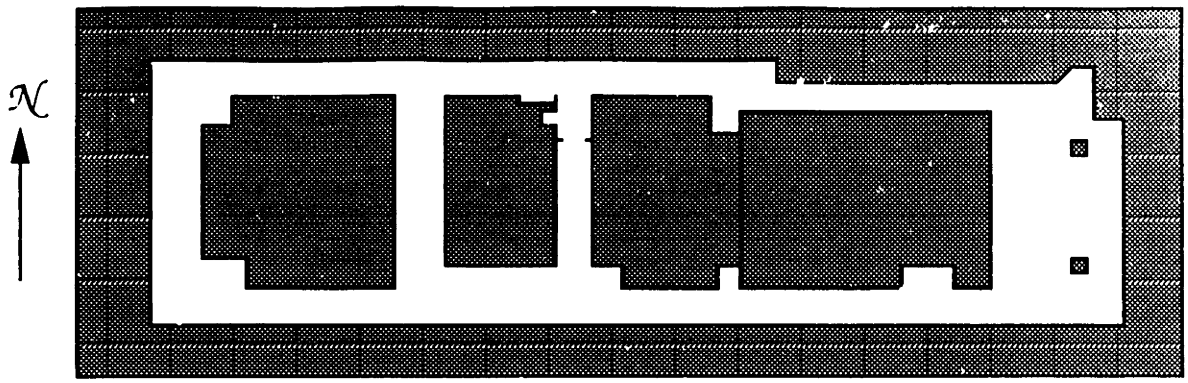


Figure 9-1: The freespace of the 7th floor.

which simplify Polly's navigation task.

One way of formalizing the navigation problem is to specify the free (unoccupied) space as a connected subset of $\mathcal{R}^n$. On first glance, the navigation problem would then simply be to find a path within that set from the start point to the goal point. However, this formulation neglects the fact that the robot occupies space itself. The freespace is not so important itself as the set of points on which we can safely place the center of the robot without any part of the robot hitting an obstacle. If we consider the case of robots which are not symmetrical, then we not only have to worry about the robot's location, but also it's orientation. The (position,orientation) pair is called a *configuration* of the robot and the space of possible robot configurations is the robot's *configuration space* (see Latombe [80]).[1] If we call the subset of the configuration space in which the robot can be placed without intersecting an obstacle the configuration freespace, then we can think of navigation as being the problem of finding a path in configuration freespace.

Since Polly is circular and omni-directional we can ignore its orientation and use $\mathcal{R}^2$ as its configuration space. This means that we can easily determine its configuration freespace by taking all the obstacles in the freespace and dilating them by 6 inches (the radius of the robot). The freespace for Polly in the 7AI environment is shown in figure 9-1. The direction "north" was arbitrarily assigned.

We will say that a navigation problem is a discrete control problem ($F \cup \{\perp\}$, A, G) in which the state space is the free configuration space, $F \in \mathcal{R}^2$, together with an extra element, $\perp$, which means that the robot has crashed (see section 6.1, p. 74). The actions, A, of the DCP are the motions which the robot can perform in one clock tick (1/15 second), and the goal is some region of freespace, $G \subseteq F$. This formalization is an oversimplification in that it does not take into account the dynamics of the robot, but they are unimportant for our purposes.

Navigation problems are most easily solved given perfect information about the space and one's position within it, although the solution may be computationally complex. Unfortunately, direct and accurate information about one's configuration

[1] Configuration space of robots can be arbitrarily complicated depending on their degrees of freedom.

is rarely available. We only have the sensor information, T, and so we need to find a T-policy (see section 6.1.1, p. 75) to solve the problem.

On paper, one can reduce the limited information case to the perfect information case, by recording the robot's position in a register and updating it as actions are performed. Formally, this amounts to first augmenting the problem (to provide the register, see section 6.1.2), and then using the T-policy:

$$p_T(x, r) = (p(r),\ (p(r))\,(r))$$

where $p: F \cup \{\perp\} \rightarrow A$ is some policy for the perfect information case. This is of course, much more confusing than the plain English version, so it deserves some explanation. The parameters of p_T are the sensory readings $x = T(s)$, and the value of the internal register, r. Notice that p_T doesn't use x. Instead, it assumes that r holds the correct position. The value of $p_T(x, r)$ is a pair, the first element of which is the action to be performed in the world, $p(r)$, and the second element of which is the new value for the register, $(p(r))(r)$. This latter value is simply the result of taking the action which is to be applied in the world and applying it computationally to get the new position.

This approach is known as dead-reckoning and it is well known not to work in practice because control actions are never perfectly predictable. Position estimates can be greatly improved by using perceptual data to close the loop. In the formalism, this amounts to using the T-policy:

$$p'_T(x, r) = (p(r),\ E(x, r, p(r)))$$

where E is some optimal estimator which computes a new estimated position given the old estimate, the current readings, and the action being taken. This is the strategy used by Moravec for the Stanford Cart [92] and used most recently by Kosaka and Kak in the FINALE system [71].

9.2 Simplifying navigation

Kosaka and Kak have had good experience with the position-estimation technique, but it is expensive and requires very detailed knowledge about the environment. We would like to bring other constraints to bear on the problem so that we can solve navigation problems using simple algorithms and realistic sensor suites. In particular, we would like to:

- avoid computing a complete path whenever possible,
- avoid the use of a detailed geometric model, and
- avoid estimating Cartesian position

We would like to avoid planning paths and computing Cartesian position, partly because they are expensive. We would like to avoid the model because the geometric

structure of the environment can change very rapidly, therefore both the model and any plans based on it can quickly become invalid without the robot's knowledge. In this section, I will use special properties of Polly's environment to simplify the navigation problem to the point where it can be solved using a set of reactive systems.

9.2.1 Reduction to the search of a small graph

The most important property of Polly's environment is that it is a network of corridors. If you can follow corridors, then you've solved much of the navigation problem. Another important property is that the intersections of the corridors are directly perceivable.

Let's assume that Polly always starts at an intersection and it's goal is always an intersection. We can then describe the configuration freespace as a graph whose vertices are corridor intersections and whose edges are corridor sections. Graph search is a well-understood problem, so it should be easy to find a path in the graph. If we can find a path in the graph, then we can execute the path using the corridor follower and the junction detector. Thus we can reduce the planning part of the Polly's navigation problem to the search of a small graph.

The reduction does not actually require that the graph consist of corridors and intersections. It just requires that the configuration freespace contain a network of landmarks and paths. We can think of a landmark L as a region of the configuration freespace such that the robot can determine when it is within L using perceptual data (see Kuipers and Byun [75]). We can think of a path as being not so much a thing in the world, as a control policy which will *somehow* reliably move the robot from one landmark to another. Formally, we require that:

1. the configuration freespace contain a number of regions, or landmarks, which are directly perceivable,

2. there exist policies which will reliably move the robot from one landmark to another, *i.e.* there are policies which reliably take the robot, in finite time, from any point in one landmark to some point in the other,

3. the landmarks are all connected by paths,

4. the goal region contains some landmark, and

5. the initial position is always within some landmark.

Condition 2 and 3 mean that the landmarks and their paths form a new DCP. Conditions 1, 2, 4, and 5 allow us to use lemma 7 (see p. 79) to reduce the navigation problem to the new DCP. In the case of Polly's environment, the landmarks are corridor intersections, and the control policies are "take the north corridor," "take the south corridor," *etc.* The reduced version of the environment is shown in figure 9-2.

Assume that we have a function L over sensor readings which when the robot is at a landmark returns the name of that landmark, and otherwise returns $\perp$, and that we have control policies p_{north}, p_{south}, p_{east}, and p_{west}, which follow the corridor in the

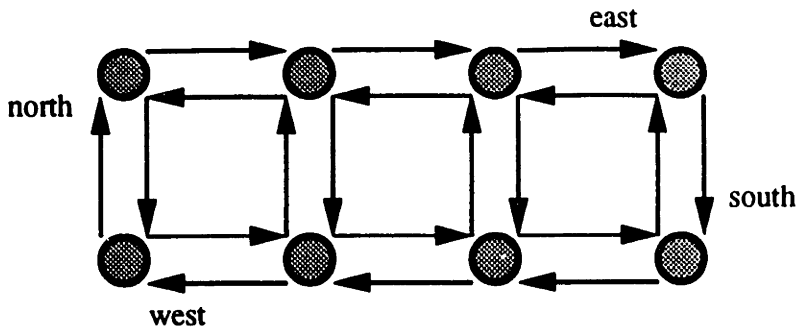


Figure 9-2: Graph structure of Polly's environment.

appropriate direction. Assume also that we have some policy, p', which solves the reduced DCP. Then by lemma 7, we can add a state variable to hold the name of the direction of the corridor the robot is presently following, and use the policy

$$p(s,d) = \begin{cases} (p_d(s),\ d) & \text{if } L(s) = \perp \\ (p_{p'(L(s))},\ p'(L(s))) & \text{otherwise} \end{cases}$$

Here the argument s is the current set of sensor readings, and d is the direction of the corridor the robot is presently following. The policy checks to see if the robot is at a landmark. If so, it computes a new direction and follows the corridor in that direction. If not, it continues in the direction it is presently heading.

9.2.2 Removal of search

The policy p has three advantages. It removes the need for estimating Cartesian position, at the cost of recognizing landmarks. It also removes the need for a detailed geometric map, although the robot still needs a topological map (the graph). Finally, the agent need only search a small graph rather than a large continuous space. This is a substantial savings, but we can do better. In fact, we can remove the need to do computational search entirely. Observe that the graph in figure 9-2 is really a grid. This means that the reduced version of the navigation problem is isomorphic to the serial product of two one-dimensional DCPs, one which can move north/south, and the other of which can move east/west. (It is the serial product because you can't move along an north corridor and an east corridor at the same time.) Each of these DCPs can be solved using difference reduction, *i.e.* move east when west of the goal and west when east of it. If we label the points in the grid with coordinates, then by lemma 11 (p. 82) we can use difference reduction to choose the corridors:

$$p'(l) = \begin{cases} \text{north}, & \text{if } y(l) < y(l_g) \\ \text{south}, & \text{if } y(l) > y(l_g) \\ \text{east}, & \text{if } y(l) = y(l_g), \text{ and } x(l) < x(l_g) \\ \text{west}, & \text{if } y(l) = y(l_g), \text{ and } x(l) > x(l_g) \end{cases}$$

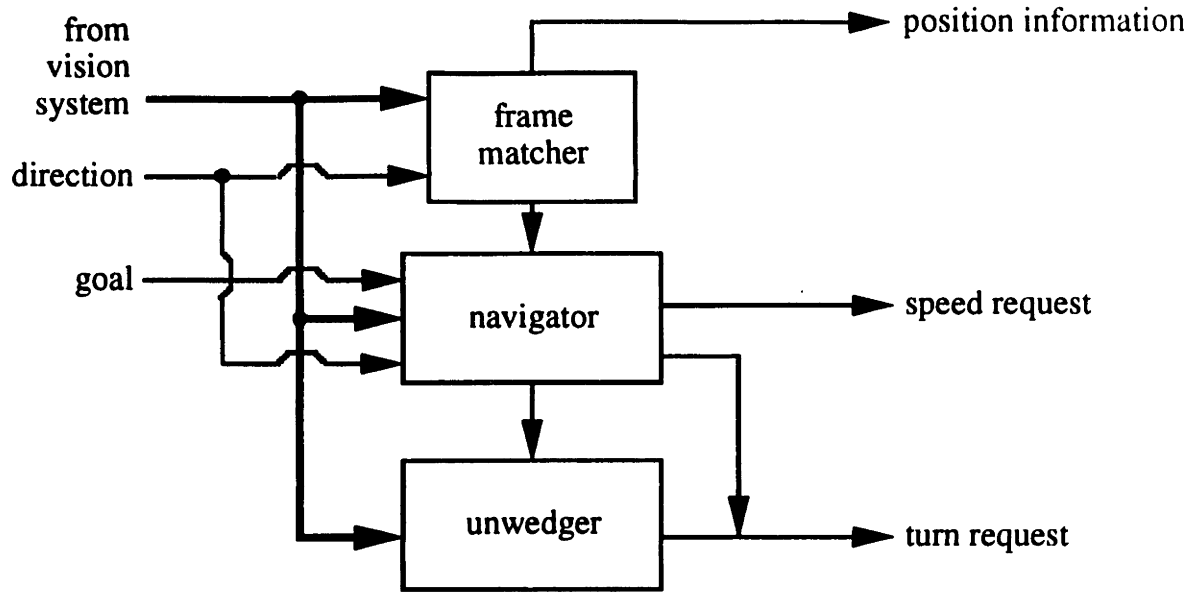


Figure 9-3: The high level navigation system.

where l is the landmark which the robot is presently at, $x(l)$ and $y(l)$ are its east/west and north/south coordinates, respectively, and l_g is the goal landmark. Note that the coordinates can be assigned arbitrarily, provided that they order the locations properly along the x and y axes.

9.2.3 Summary

By taking advantage of the special structure within the environment, we have removed the need for Polly to perform computational search during navigation. Effectively, we have gradually optimized a planning system into a reactive system. We start out with a geometric path planner/executer architecture. Since (1) the environment consists of a network of paths and recognizable landmarks (2) the initial position is always a landmark, and (3) the goal is always a landmark, we can replace the *geometric* path planner with a *symbolic* planner and the geometric path executive with a corridor follower. The symbolic planner has a much smaller search space (the graph), and so can be much more efficient. Since the graph is isomorphic to a grid, we can then replace the symbolic planner with a reactive system which performs difference reduction.

This analysis makes clear what additional assumptions the specialized system (the reactive system) is making, and what their computational value is. It also tells us that if we want to run the system in a world whose graph is not isomorphic to a grid, we do not have to throw out the hold system, we need only back off of the final optimization and return to using search within the graph.

9.3 Navigation in Polly

We have seen how Polly's navigation problem is easier than the general case of navigation because of specific properties of the environment. These properties allow an idealized policy, p, to solve the navigation problem without any planning, in fact, using only two bits of state information (the direction register telling whether to go north, south, east, or west). Polly implements a version of this policy using a network of parallel processes shown in figure 9-3.

One group of processes, the low-level navigation system which implements corridor following, was discussed in chapter 8. One difference between Polly and p is that p has four different corridor followers (p_{north}, p_{south}, p_{east} and p_{west}), and a register (the l parameter of the policy) which determines which one is used at any given time. Polly has only one corridor follower, which follows whatever corridor the robot is in, and has no register. Thus it might appear that Polly is implementing a qualitatively different policy. In fact this is not the case. Polly stores the direction of the desired corridor in its own orientation. In effect, Polly *does* have the direction register, it simply stores the direction information *in the world* rather than in RAM. Another way of looking at it is that it uses its orientation as a very slow RAM, one which is read by checking the shaft encoders, and which is written by turning. This externalization of state is possible because all the paths connecting all the landmarks in Polly's world can be followed using the same policy and are locally identifiable by their orientation. This need not be the case in an arbitrary environment.

Another process, the navigator, chooses new corridors when the robot comes to an intersection. The navigator then sends a signal to a third process, the ballistic turn controller, which aligns the robot with the new corridor. In effect, the low-level navigation system implements p_{north}, p_{south}, p_{east} and p_{west}, the navigator implements p', the orientation of the robot implements the internal state, and the ballistic turn controller implements the write circuitry for the internal state.

9.3.1 The navigator

The navigator is implemented by the Scheme procedure **navigator**, which receives messages on its inputs **goal-x** and **goal-y**, and sends messages on the wires **speed--request** and **turn-request**, which are handled by the speed controller and ballistic turn controller. The navigator also monitors the outputs **last-x**, **last-y**, and **frame-strobe?** from the place recognition system. The first two of these hold the coordinates of the last landmark which was recognized. **Frame-strobe?** is a signal which is asserted for one clock tick when a new landmark is recognized.

When the navigator's goal inputs are zero, there is no goal, and the navigator inhibits motion by setting **speed-request** to zero. When the goal inputs are non-zero, the navigator sets **speed-request** to full speed, and continually checks to see if a new landmark has been reached. When a landmark is reached, it checks whether the landmark is in fact an intersection (the place recognition system allows other types of landmarks), and if so, checks to see whether a left turn or right turn would point it in the direction of its goal. If so, it waits roughly 2 seconds, and issues a turn (by setting **turn-request**). The 2 second delay is needed because the vision system detects the turn before the robot is actually in the intersection. The navigator also

continuously informs the unwedger of what direction it would turn it it were to reach an intersection (see below).

While the robot is moving, the navigator continually checks whether it has reached its goal (`last-x=goal-x` and `last-y=goal-y`). When it reaches its goal, it clears `goal-x` and `goal-y` and stops. Thus the robot will stay there until some process sends a new goal to the navigator.

Because the place recognition system can fail, the navigator also continually checks whether it has overshot its goal. If so, then it will initiate a u-turn, even though it hasn't yet reached a landmark.

9.3.2 The unwedger

One problem with the local navigation strategy used by the corridor follower is that it can get stuck in local minima. This happens, for example, when the robot is perfectly perpendicular to a wall, in which case the difference between the space on the left and space on the right will be zero and the robot will not turn, even though it is blocked. The unwedger is used to take care of this problem. When the robot is blocked for more than two seconds, the unwedger initiates a 45 degree ballistic turn. The turn is in the same direction that the navigator would turn if the navigator were at an intersection. This is useful because the robot often *is* at intersections when it sees a wall in front of it, but it cannot tell because it is unable to turn its head. If the navigator has already aligned itself with the goal along one of the axes, then it will be traveling along the other axis and will not want to turn at an intersection. In this case, the unwedger is forced to turn in the direction of the last turn it saw. This is a useful strategy for getting out of *cul de sacs.*

In most cases the turn initiated by the unwedger will be sufficient to direct the corridor follower toward a new corridor. If the robot has driven into a dead end, then the unwedger will fire every two seconds until the robot has turned itself around, at which point it will leave the way it came. Occasionally, the robot will align itself with a corner and the unwedger and corridor follower will fight with one another—the unwedger turning away from the corner, and the corridor follower turning toward it. This problem was rare enough to be ignored.

9.4 Place recognition

Polly keeps track of its position by recognizing landmarks and larger-scale "districts." These places are given to it in advance. The lab and some of its landmarks are shown in figure 9-4. The job of the place recognition system is to determine, on each clock tick, whether it has entered the region corresponding to a landmark or district, and if so, what its qualitative coordinates are. The coordinates are only qualitative because the navigator only requires that they order the landmarks properly along the north/south and east/west axes. Therefore the coordinates can be warped by any monotonic, or rather strictly increasing, deformation. The coordinates of the different landmarks are shown in figure 9-4.

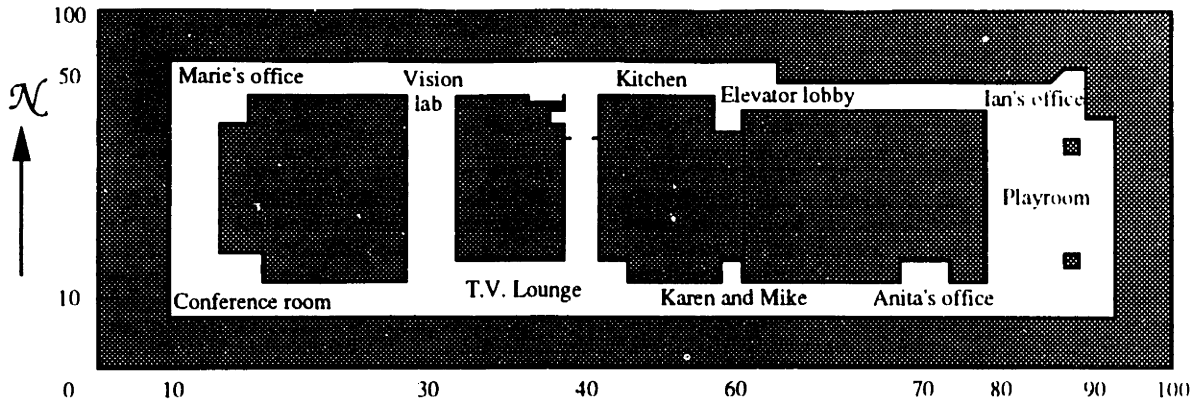


Figure 9-4: Landmarks in Polly's environment and their qualitative coordinates.

Kitchen	
Position	(50, 40)
Direction	west
Veer	0
Image	...

Corridor 1	
Position	(40, 40)
Direction	west
Veer	0
Features	left

Elevator lobby	
Position	(60, 40)
Direction	east
Veer	45
Features	right, wall

Figure 9-5: Example place frames.

Information about landmarks is stored in an associative memory which is exhaustively searched on every clock tick (66ms). The memory consists of a set of frame-like structures, one per possible view of each landmark (see figure 9-5). In some cases, more than one frame per view is given to compensate for variations in lighting or geometry over time. Each frame gives the expected appearance of a place from a particular direction (north/south/east/west). All frames contain a place name, qualitative coordinates, a direction and some specification of the landmark's appearance: either a 16 × 12 grey-scale image or a set of qualitative features (left-turn, right-turn, wall, dark-floor, light-floor). No explicit connectivity information is represented. Frames can also be tagged with a speech to give during a tour or an open-loop turn to perform. The latter is used to deal with the jog in the corridor at the elevator lobby. The complete set of frames describing the 7th floor is given in appendix A.

While at first glance this may seem to be an inefficient mechanism, it is in fact quite compact. The complete set of 32 frames for the 7th floor requires approximately 1KW of storage. The system can scan all the frames and find the best match at 15Hz using only a fraction of the CPU.

The code for the frame matcher is shown in figure 9-6. The matcher process is implemented by a scheme procedure `frame-matcher`. When a match is made, the matcher asserts the output `frame-strobe?` for one clock tick, latches the matched frame into the output `current-frame`, and latches its coordinates into the outputs `last-x` and `last-y`. Each frame is given a score and the frame with the best score is chosen from those frames with different coordinates than the previous position. If the best frame's score is below threshold (low scores are good), then that frame is taken to be the new position. The frame matcher disables itself if it is not in a corridor,

is not moving, has been blocked recently, or has recently matched another place, so as to prevent false matches. The disabling due to recent blockage or matching is implemented using the internal counter `frame-debounce`, which holds the number of clock ticks (15ths of a second) to wait before matching. The time to wait after matching is stored in a frame slot. Its default value is 2 seconds (30 ticks).

The matching of a given frame to sensory data is performed the Scheme procedure `match-frame`, whose source code is shown in figure 9-7. The basic scheme is to compute a score based on the sum of three penalty terms: one due to difference in position from the previous position, one due to difference in direction from the expected direction, and one due to the difference in appearance. The penalty due to position differences is dependent on direction. If the robot is moving east and the frame is west or north of the last position, then the penalty will be higher than if it were east of the last position.

There are two types of frames corresponding to two types of feature sets. Image frames contain 16×12 grey-scale images of the landmarks taken from specific distances and directions. The penalty term for image frames is simply the sum of the squared differences of the corresponding pixels. Feature frames contain a bit vector of features (left turn, right turn, dark floor, wall ahead) computed by the visual system. The penalty term for feature frames is small if the current feature vector and the frame's feature vector are exactly the same, and large otherwise. `Match-frame` will refuse to match feature frames unless the feature bits have changed since the last clock tick. This is an additional mechanism for preventing the system from prematurely matching a frame which happens to appear exactly like the current frame. The requirement that the feature bits change amounts to a requirement that the robot leave one intersection or open space before matching the next one. Note that the wall feature is special cased because the wall detector can only detect walls at a fixed distance. As the robot approaches a corner of the lab, it will first see a turn, then a turn and a wall, and then just a turn again. The precise requirement for matching is that the robot be viewing a wall *or* that some other feature has changed. This prevents the robot from trying to match the turn a second time after the wall disappears.

The matching of both types of frames relies on the fact that the robot will only view the landmark from certain directions. The restricted viewing angle is due both to the geometry of the building (long narrow corridors), and the fact that the corridor follower tends to keep the robot in the center of the corridor looking forward. A static landmark is can be thought of as a very odd rigid object. Roughly speaking, the things which make the matching of rigid objects hard are variation in lighting, variation in viewpoint, and occlusion. Polly simply fails in the presence of occlusion, but handles the other two cases by relying on the fact that they don't actually vary very much in this domain. Lighting in office buildings is usually internally generated and so is kept at a constant level. Most of the viewing parameters are held fixed by the fact that the camera is bolted down—it cannot move up or down or rotate about its optical axis. The whole robot can move, but the fact that the robot is keeping itself aligned with the axis of the corridor means that it can only approach a landmark from one of a very few directions. Thus the only parameter of the viewpoint which can

```
(define (frame-matcher)
  (when (and corridor?
             (> speed #x1800))
    (when blocked?
      (set! frame-debounce 30))
    ;; FRAMES is the address of the list of frames to be matched.
    ;; FRAME is a pointer to the current frame.
    (let ((frame frames)
          (best frames)
          (best-value 99999999))
      (decf frame-debounce)
      (set! frame-strobe? false)
      (when (< frame-debounce 0)
        (find-districts)
        (dotimes (n frame-count)
          (let ((matchval (match-frame mini frame)))
            (when (< matchval best-value)
              (unless (and (= (frame-x frame) last-x)
                           (= (frame-y frame) last-y))
                (setf best frame)
                (setf best-value matchval))))
          ;; Advance FRAME to the start of the next frame.
          (setf frame (shift frame (if (image-frame? frame)
                                       image-frame-length
                                     feature-frame-length))))
        (when (< best-value 70000)
          (setf frame-debounce (frame-place-size best))
          (setf frame-strobe? true)
          (setf current-frame best)
          (setf last-x (frame-x best))
          (setf last-y (frame-y best))))
      (set! last-match-bits feature-bits))))
```

Figure 9-6: Source code for frame-matcher process. Code has been slightly simplified by removing compiler declarations.

```
(define (match-frame mini-image frame)
  (let ((fimage (frame-image frame))
        (sum 0)
        (bits-changed? (not (= (logand feature-bits #b11110)
                               (logand last-match-bits #b11110)))))
    (when wall?
      (set! bits-changed? true))
    ;; Compute penalties based on estimated position and direction of
    ;; motion.
    (let ((delta-x (- (frame-x frame) last-x))
          (delta-y (- (frame-y frame) last-y)))
      ;; Encoding of direction is north=0, east=1, etc.
      (incf sum (* (abs delta-x)
                   (vector-ref direction
                               (if (> delta-x 0)
                                   ;  N    E    S    W
                                   #(3000 100 3000 3000)
                                   ;  N    E    S    W
                                   #(3000 3000 3000 100)))))
      (incf sum (* (abs delta-y)
                   (vector-ref direction
                               (if (> delta-y 0)
                                   ;  N    E    S    W
                                   #(100 3000 3000 3000)
                                   ;  N    E    S    W
                                   #(3000 3000 100 3000))))))
    ;; Large penalty for getting the direction wrong.
    (unless (= (frame-direction frame)
               direction)
      (incf sum 100000))
    (if (image-frame? frame)
        ;; MINI-IMAGE is the 16x12 version of the current image.
        (incf sum (compute-difference fimage mini-image))
      (incf sum (if (and (= (frame-features frame)
                            feature-bits)
                         bits-changed?)
                    60000
                    10000000)))
    sum)))
```

Figure 9-7: Source code for matching process. Code has been slightly simplified by removing declarations and code for computing image differences.

vary in the normal case is the robot's distance. But this parameter is being scanned *mechanically* as the robot drives along the corridor. The result is that there is no need to do computational search in pose space. Matching of viewpoint-dependent templates is sufficient. All this is, again, dependent on the fact that the robot's path is tightly constrained by the environment. I will call this the constant viewpoint constraint.

The system can also recognize large-scale "districts" and correct its position estimate even if it cannot determine exactly where it is. There is evidence that humans use such information (see Lynch [84]). The robot presently recognizes the two long east/west corridors as districts. For example, when the robot is driving west and sees a left turn, it can only be in the southern east/west corridor, so its y coordinate must be 10, regardless of its x position. This allows the robot to quickly recover from getting lost. At present, the recognition of districts is implemented as a separate computation, but it ought to be folded into the frame system. The recognition of districts is implemented by the `find-districts` procedure, shown in figure 9-8. `Find-districts` is called by `frame-matcher`.

9.5 Patrolling

Patrolling is implemented by a separate process which gives alternate goals to the navigator. The process is implemented by the procedure `wander`, which sets the goal to the playroom when the robot gets as far west as the vision lab, and the vision lab when the robot gets as far east as the playroom.

The `wander` process is used to implement both the patrol pattern and the path for giving tours. `Wander` always controls the navigator's goals, unless the robot is in the process of starting up or going home.

9.6 Sequencers and the plan language

I have implemented a simple plan language which allows one to specify fixed sequences of actions for the robot to perform. Since parallelism is the default in Polly, a separate mechanism is necessary to reintroduce seriality. This is done using the macro `define-sequencer` which takes a sequence of condition/action pairs and produces a new process (Scheme procedure) along with a state variable, its program counter. The process checks its program counter on each clock tick. If it is a positive number n, then the process is enabled, and the process checks the nth condition. If the condition is true, then it executes the corresponding action, which is an arbitrary piece of Scheme code, and increments the program counter. The basic condition/action pair is given with the form:

(when *condition*
action)

Here the condition can be an arbitrary piece of scheme code. Various bits of syntactic sugar are available to make it more readable: **(wait** *condition*) and **(then** *action*)

```
(define (find-districts)
  ;; Use a stricter criterion for turns so that we don't get doorways.
  (let ((left-turn? (> left-distance 25))
        (right-turn? (> right-distance 25)))
    (when (and aligned?
               ns-corridor?
               wall?)
      (when (= direction north)
        (set! last-y 40))
      (when (= direction south)
        (set! last-y 10)))
    (when (and ew-corridor?
               (not dark-floor)
               aligned-long-time?
               (not blocked?))
      (when (and (= direction west)
                 open-right?
                 (not open-left?)
                 in-corridor?)
        (set! last-y 10))
      (when (and (= direction east)
                 (not open-right?)
                 open-left?)
        (set! last-y 10))
      (when (and (= direction west)
                 (not open-right?)
                 open-left?)
        (set! last-y 40))
      (when (and (= direction east)
                 (not open-left?)
                 (< last-x 70)
                 open-right?)
        (set! last-y 40)))))
```

Figure 9-8: Source code for find-districts. Some of the repeated tests could be collapsed into single disjunctions, but it would have required the author to have done a better job implementing boolean tests in his compiler.

```
(define-sequencer (leave-office :import (...))
  (first (pdisplay-goal "Leave office"))
  (when blocked?
    ;; robot should now be at far wall
    (set! global-mode 1)
    ;; turn toward door
    (turn! -40))
  (sleep 2)
  (when open-left?
    ;; robot should be out of office
    (pdisplay-goal "Align"))
  (sleep 2)
  ;; Now we want to make ourselves parallel with the corridor.
  ;; turn toward east wall of playroom
  (then (turn! -60))
  (sleep 1)
  (when (and (= direction east)
             blocked?)
    ;; Should be aligned with east wall; turn around
    (turn! 150))
  (wait ew-corridor?)
  (sleep 5)
  ,; Robot now thinks it's aligned; set position.
  (then
    (set! last-x 80)
    (set! last-y 40)))
```

Figure 9-9: Source code for the sequencer for leaving the author's office and entering the corridor. `pdisplay-goal` is a macro for displaying a string on the LCD display.

are ways of specifying pairs with null actions or conditions, respectively. (`Sleep` *seconds*) causes the process to be inactive for the specified period.

A sequencer is started, or "fired," when another process executes the form (`do!` *sequencer-name*). The sequencer then continues until it it finishes the last condition/action pair, at which point it waits to be fired again. Multiple firings (executions of `do!`) do not produce multiple copies of the sequencer. If a sequencer is fired while it is already executing, it simply resets to its first condition/action pair.

An example sequencer is shown in figure 9-9. It is a plan for leaving my office and entering the hallway. It is fired at startup time. The `:import` parameter is a compiler declaration.

```
(define-sequencer (offer-tour :import (...))
  (first (set! global-mode 3)
         (set! inhibit-all-motion? true))
  (when done-talking?
    (new-say "Hello.  I am Polly.  Would you like a tour?
              If so, wave your foot around."))
  (sleep 9)
  (then (if blocked?
            ;; The person's still there.
            (if (> hmotion 30)
              (do! give-tour)
              (begin
               (new-say "OK.  Have a nice day.")
               (set! global-mode 1)
               (set! inhibit-all-motion? false)))
            ;; The person's gone.
            (begin (set! global-mode 1)
                   (set! inhibit-all-motion? false)))))
```

Figure 9-10: Sequencer code for offering a tour.

9.7 Giving tours

Giving tours is implemented by a pair of sequencers, `offer-tour` and `give-tour`, shown in figures 9-10 and 9-11. `Offer-tour` is fired by the process `interact`, when the robot is in patrol mode, in an east/west corridor, is blocked by a person, and has not offered a tour in the last 5 seconds. `Give-tour` is fired by `offer-tour` when a visitor accepts the offer.

```
(define-sequencer (give-tour :import (...))
  (first (new-say "OK.  Please stand to one side.")
         (if (= last-y 40)
           (begin (set! tour-end-x 80)
                  (set! tour-end-y 10))
           (begin (set! tour-end-x last-x)
                  (set! tour-end-y last-y))))
  (when (not blocked?)
    (new-say "Thank you.  Please follow me.")
    (set! inhibit-all-motion? false)
    (set! global-mode 2))
  (wait frame-strobe?)
  (wait (at-place? tour-end-x tour-end-y))
  (sleep 1.0)
  (then
    (new-say "That's the end of the tour.
              Thank you and have a nice day.")
    (set! global-mode 1)))
```

Figure 9-11: Sequencer code for giving a tour.

Chapter 10

Experiments with Polly

Polly is one of the best tested vision-based robots to date. The low-level navigation system has seen hundreds of hours of testing in several different environments. The high-level navigation system and the tour-giving components are newer and so less well tested. To date, the robot has given around 50 tours, of which 20 involved the full pick-up protocol.

It is very difficult to test the performance of the robot in any direct and conclusive manner. It has been suggested to me that I measure the integral of the robot's deviation from the mid-line of the corridor as it drives through the corridor and use this to characterize the corridor follower's performance. Unfortunately, staying in the center of the corridor is not actually part of the robot's task. This is an important fact. It is neither necessary, nor particularly desirable for the robot to stay in the middle of the corridor. All that is necessary is that it safely get from one side to another. There are quantities which one might want to optimize, such as motor power consumption, but even these are hard to evaluate. Motor power consumption is actually dwarfed by computer power consumption on Polly. A robot which expends 5 watt-hours optimizing the last few hundredths of a watt-hour out of a motor plan which would only take 0.1 watt-hours anyhow, is not necessarily a good thing. On the other hand, total power consumption is dependent on so many irrelevant variables, that it is unclear how to make meaningful comparisons. Another problem is that it is hard to evaluate factors, such as sensitivity to local geometry, that involve very high dimensional spaces (*i.e.* the space of possible configuration spaces), in any kind of systematic manner.

For these reasons, much of the data presented here are, of necessity, anecdotal. Information tends to be more qualitative than quantitative, and much of it is, in the end, dependent on the particular environment in which I tested it. This is true of any complicated artifact interacting with a complicated environment. I see no real alternative to this.

Since most of the time the robot works fine, I have focused mostly on the errors which do occur. I have tried to catalog the different types of failure modes that I have observed in the system as it has moved outside of its design envelope. I have also tried to document the ways in which it recovers from problems, and the cases in which it cannot. While this information is less definitive than numerical tests of

System	Frames/sec	MIPS	Speed	Obstacle detection?
Polly	15	16	1m/s	Yes
Stanford Cart (Moravec)	0.001	.25 (?)	0.003	Yes
FINALE (Kosaka & Kak)	0.03	16	0.2	No
Mobi	0.1	1	0.1	Yes
Thorpe *et al*	0.1	1	0.4	No
VTIS	< 1 ?	?	1–5.5m/s	No
VaMoRs-87 (Dickmanns)	25	1.32 (?)	26.4	No
VaMoRs-91 (Dickmanns)	25	?	14	Yes
SCARF (Crisman)	0.1	16	?	No
ALVINN (Pomerleau)	?	?	55 m/h	No
AuRa path follower	0.18	0.5	0.18	No

Table 10.1: Processing and driving speeds of various visual navigation systems, and the problems they solve. The first group are indoor navigation systems intended for office buildings, although the Stanford Cart was also run outdoors. The second group are outdoor path followers, mostly road followers.

speed and accuracy, I feel that it is more edifying. In the end, I feel that it gives a more accurate picture of the performance of the system. None of this is to say that the robot is unreliable. Quite the contrary: the robot works well enough in the normal cases for which it is designed that we can begin to focus on its performance in pathological situations. Many of the failure modes discussed here, such as darkened rooms, the presence of specular reflection or the presence of isoluminant edges, would cause problems for most of the vision systems that have been built to date.

10.1 Speed

Polly is the fastest indoor vision-based robot to date. It is fastest both in terms of processing speed and rate of motion. Table 10.1 shows the processing and driving speeds for a number of robots in the literature. Since the different systems perform very different tasks, it is difficult to draw firm conclusions from the numbers. They are sufficient to show that Polly is a fast system by any standard, however.

10.1.1 Processing speed

Because all visual processing was done at low resolution, it could be done at a very high frame rate. A summary of the processing time for different levels of capability is given in table 10.2. The reader should note that the elapsed time does not decrease as the amount of processing decreases, thus the system is I/O bound. The I/O time was spent waiting for the serial link to the base, waiting for VMEbus transfers to and from the frame grabber, and waiting for the frame grabber to finish grabbing a frame. Although it is hard to test this directly, I believe that the principle limitation was that the frame grabber could not grab consecutive frames without double-buffering, which

Test	Time (sec)	Frames/sec
Full system	67	15
Corridor follower	67	15
I/O only	67	15
No I/O	15	67
No VP	10	100

Table 10.2: Execution times for 1000 frames. "Full system" is all code presently implemented, including the person detector. "No I/O" is the corridor follower without any frame grabbing or output to the base (a single frame is grabbed at the beginning and processed repeatedly). "No VP" is the collision avoidance system run without I/O or the vanishing-point box. All execution times are for a Texas Instruments TMS-320C30-based DSP board (a Pentek 4283) running with no wait states. The processor has a 60ns instruction time. The first three lines are the same because the system cannot digitize frames faster than 15 FPS.

could not be implemented without sacrificing video output for debugging. Even if this restriction were removed, it would only have shifted the bottleneck to the base. At 9600 baud, there is only time to transmit 32 bytes per video frame time to the base. Since most base commands require at least 11 bytes of I/O, between input characters and output prompts, there would not be sufficient time to perform all the necessary transactions with the base at video rate. At 15fps, which is one half video rate, there is just barely enough time to reset the velocities as acceleration caps, and to poll the base for the odometry.

There is room for a great deal of improvement. A better compiler might be able to halve the processing time. Since the system was I/O bound in any case, it was not worth implementing the compiler improvements.

10.1.2 Driving speed

Driving speed was limited by the dynamics of the robot base, not by processing speeds. Although the robot's center of gravity is well below its midpoint, it can still fall over when decelerating, since all torques are applied at the very bottom of the robot. The result is that the wheels stop moving but the robot's card cage keeps moving and the robot falls over. This can be controlled by imposing acceleration caps at the cost of greatly increasing the stopping distance of the robot. The other problem is that internal control system of the base is calibrated for unloaded conditions. When the base tries to go from a stop to its maximum speed, the motors cannot provide sufficient force to accelerate the base at the rate the control system wants. When the control system notices that it isn't moving fast enough, it signals an error and shuts down. Therefore speeds over 1m/s require pushing the robot to help it accelerate. The control system can be disabled, allowing the DSP to directly specify motor currents, but then the robot tends to pop "wheelies," and so again, human intervention is required to keep it from falling over. I believe the system has run as fast as 2.5m/s in this mode, but I have not calibrated velocity measurements. Apart from these

stability and low-level control considerations, the navigation system appears to be able to pilot the robot as fast it is physically capable of moving.

Polly has been run on two different bases. The robot was generally run at 1m/s on the old base. Unfortunately, the base was damaged during the testing of the bump sensors and so a new based was substituted. The new base, while otherwise functionally equivalent, had a different gear ratio and internal control system. This new control system was grossly underdamped. The new base regularly accelerated to 2m/s regardless of the velocity it was given, at which point it would brake and fall over. It was necessary to add extra damping within the DSP's control system to compensate for this. Even so, the new base can only safely run at 0.75m/s.

10.2 Complete test runs

The night the pick-up system was first implemented, I videotaped four test runs. Traces of the test runs are shown in figures 10-2 through 10-7. The robot successfully gave the tour in three of the four tests.

In the first run, the visitor misunderstood the instructions given by the robot and thought the robot said to move out of the way. The visitor moved part-way out of the way, and the robot misinterpreted the motion as a wave of the foot. It correctly believed that the visitor wanted a tour, but for the wrong reasons. When the robot started the tour, it was too close to the elevator lobby to recognize it as a landmark. It had not recognized it previously because it was occluded by the visitor. The robot should have recognized the area and veered to the right, but instead continued forward until it was blocked by the wall. Fortunately, the unwedger and the navigator were able to successfully pilot the robot into the next corridor, in spite of the failure, see figure 10-1. Once the robot reached the kitchen, the place recognition system resynced itself, and the robot continued without incident.

In the second test, a place recognition error caused it to make an inappropriate turn from which it took a long time to recover. The run was terminated, and the robot was allowed to run, with the navigator turned off, until the place recognition system resynced itself. The robot made no errors after that point. The other test runs were performed properly.

10.3 Other environments

Much of Polly's efficiency is due to its being specialized to its environment. Of course, the robot is not specialize to *that particular environment*, but to a class of environments, which we can call its "habitat." One of the aims of the analytic part of this work has been to understand what features of the environment are necessary to the proper operation of the robot, or alternative, what the robot's habitat truly is. According to the theory, the low-level navigation code should work in any environment which consists of carpeted corridors whose carpets are textureless, and in which all obstacles rest on the ground. I have conjectured that these properties are true of many office buildings (office environments). I have tried to test this conjecture directly, as

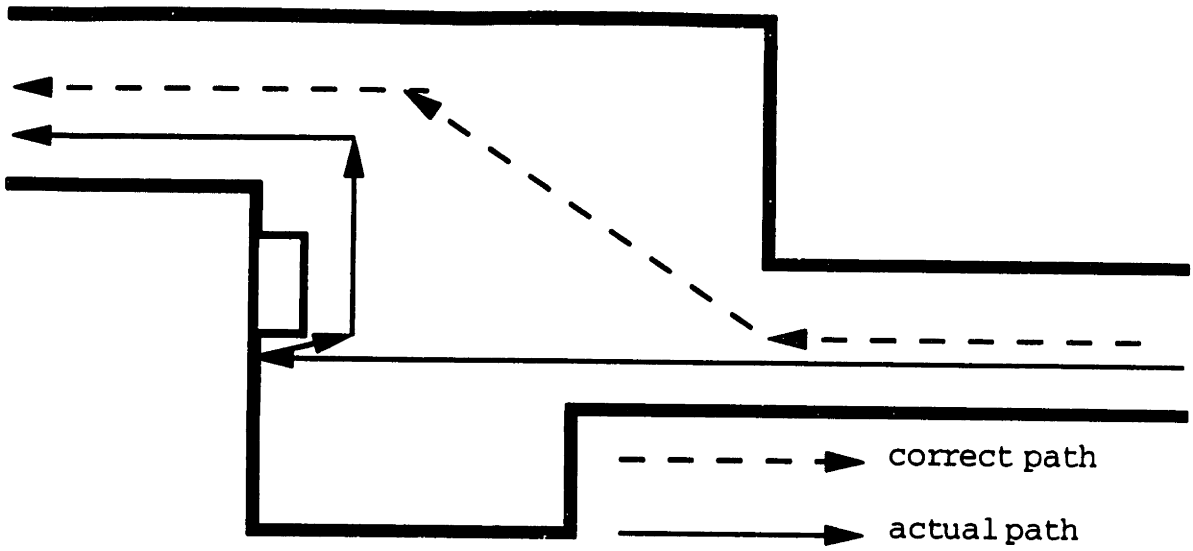


Figure 10-1: Detail of the area near the elevator lobby with the robot's intended path and actual path during the first test run..

much as possible, and so have tested the robot in environments other than the AI lab.

Note that *no parameters were changed for any of these tests.* The only difference between the code run for these tests and the code run on the 7th floor is that the navigator was disabled (because the robot had no map for the other floors).

10.3.1 Tech Square

Tech Square is the common term for the building which houses both the AI lab and the Laboratory for Computer Science. The first test of Polly was to take it to different floors of Tech Square. The building has nine floors plus a basement. The corridor follower and obstacle avoidance work on all floors except the ninth floor and the basement. These floors have shiny linoleum tile rather than carpet. The tile is shiny enough that it acts like a dirty mirror when viewed from two feet above the ground. The result is that images of the overhead lights appear in the floor. Worse yet, they move with the robot so that if one happens to be close enough to the robot to make it back up, the robot will back into a wall. The roughness of the floor serves to diffuse the light somewhat, so it is possible that the edge detector could be tuned to a high enough frequency band to ignore the lights.

Many floors have carpet boundaries. The robot had no problems with the boundaries which appeared horizontal in its field of view because the carpet boundary detector was specifically tuned to horizontal boundaries, but some boundaries paralleled the walls. Such boundaries restricted the robot to the channel between the wall and the boundary. If the robot had turned 90 degrees at the right point, it probably could have crossed the boundary, but it did not do so. A more intelligent algorithm for detecting carpet boundaries would be a great asset.

Place	Action	Comments
corridor	drive west	
	stop	detected person
	"Hello..."	introduction speech
	pause	confused visitor moves out of the way
	"Please stand to one side"	Polly makes mistake visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
elevator lobby	drives west	robot misses landmark
west wall	stop	
	turn right	unwedger takes over
	drive north	
north wall	stop	
	turn left	unwedger takes over robot now on course
	"I can avoid obstacles..."	chatter speech
kitchen	"...follow corridors..."	robot still making previous speech
intersection	"On the left here..."	kitchen place speech
	"My vision system runs..."	chatter speech
vision lab	turn left (south)	
	"On the right here is..."	Vision lab speech
	drives south	
T.V. lounge wall	stop	
	"By the way..."	chatter speech
	turn left (east)	unwedger takes over
	drives east	
couch	drives east	
end of couch	"This the T.V. lounge..."	place speech
end of lounge	"God, this place..."	chatter speech
	drives east	
Karen's office	"This is..."	place speech
	drives east	
Anita's office	"This is..."	place speech
	drives east	
playroom	turns left	
	"This is..."	place speech
	drives north	
near chair	stops	
	turns left	obstacle avoidance
	"This is the end..."	Goodbye speech
	drives	cruise mode

Figure 10-2: Transcript of first test run.

Place	Action	Comments
west wall	stops	
	turns right	obstacle avoidance
	drives north	
north wall	stops	
	turns left	unwedger takes over
	drives west	

Figure 10-3: Transcript of first test run (contd.)

Place	Action	Comments
corridor	drive east	
	stop	detected person
	"Hello..."	introduction speech
	pause	visitor waves foot
	"Please stand to..."	visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
Anita's office	"This is the playroom..."	place recognition error
	turns left	
	drives north	drives into printer alcove

Figure 10-4: Transcript of second test run.

10.3.2 The Brown CS department

Polly was also tested at the Computer Science Department at Brown University. Again, the system worked well: it successfully followed corridors, avoided obstacles, including people, and moved through open spaces. There were problems, however. The robot's major problem was lack of light. The lights were dark enough to seriously inhibit the response of the video camera. Since many of the edges between the walls and the carpet were very low contrast to begin with, the edge detector would often miss them when running at low light levels. If robot saw one wall, but not the other, it would sometimes drive into the invisible walls. Again, this situation would be a problem for nearly any vision system.

There were two aspects of the environment's geometry which were problematic. One was the presence of a downward staircase. It is possible that the robot would have seen an edge at the boundary of the staircase and stopped, but I did not want to risk having my robot fall down the stairs. The other problem was the presence of some very narrow corridors. These corridors were narrow enough that the robot could not turn around in them without backing into the wall. The robot could perform a 180 degree turn in place if it knew to do so, but if the robot reached a dead end and relied on the unwedger to turn it around 45 degrees at a time, then it would back into the wall between turns. This problem would have been solved with a rear bumper.

Place	Action	Comments
corridor	drive east	
	stop	detected person
	"Hello..."	introduction speech
	pause	visitor waves foot
	"Please stand to..."	visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
Anita's office	"This is..."	place speech
	drives east	
	"I can avoid obstacles..."	chatter speech
playroom	turns left	
	drives north	
	"This is..."	place speech
	drives north	
near chair	stops	
	"My vision system..."	chatter speech
	turns left	obstacle avoidance
	drives	
west wall	stops	
	turns right	obstacle avoidance
	drives north	
north wall	stops	
	turns left	unwedger takes over
	drives west	
	"By the way..."	chatter speech
	drives west	
elevator lobby	veers right	
north wall	stops	
	turns left	obstacle avoidance
	drives west	
kitchen	"On the left..."	place speech
	drives west	
	"God,..."	chatter speech
vision lab	turns left	
	"On the right..."	place speech
	drives south	
T.V. lounge wall	stop	
	turn left (east)	unwedger takes over
	drives east	

Figure 10-5: Transcript of third test run.

Place	Action	Comments
couch	drives east	
end of couch	"This the T.V. lounge..."	place speech
	drives east	
Karen's office	"This is..."	place speech
	drives east	
	"This is the end..."	goodbye speech

Figure 10-6: Transcript of third test run (contd.).

10.4 Burn-in tests

I have conducted a number of "burn-in tests" on the robot. In these tests, I have let the robot run laps for 45 minutes or more. As of this writing, I have performed five burn-in tests. The longest test was 90 minutes. All runs worked well, running without incident for extended periods. Three of the five runs ran flawlessly until they were terminated by me. The other two are described below.

In the first test, the robot ran for 45 minutes without incident. The test was terminates by a series of mishaps. The base, being underdamped, started up too fast and then broke, almost tipping. This generated an internal error in the firmware control system of the base, which caused the base to halt and limp itself, ignoring the commands coming from the DSP. I reset the base, but forgot that resetting the error also resets the odometer, so that the robot would think it was pointing south, which it was not. The robot then performed most of a lap in spite of having completely incorrect odometry information. When I realized the base was reporting bad odometry, I reset it again to fix the odometer. In the process I accidentally triggered the go-home function which, at the time, could not be turned off and which was not interlocked with the wanderer, so the go-home sequencer and the wanderer fought for control of the navigator. Surprisingly, the wanderer won and the robot continued to do laps, but eventually the wanderer brought the robot near my office and that caused the go-home routine to forcibly park the base. The whole series of mishaps took about three laps to complete. The series of mishaps was quite extreme, but the robot nevertheless succeeded in performing several more laps and parking safely.

The most recent test, the 90 minute test, was terminated by the failure of the motor battery. The battery appears to have been weakened by the racing in the control system of the new base. When the robot maneuvers through tight passages or oddly shaped areas, it starts and stops frequently, this causes the based control system to race: it first accelerates to roughly 2m/s and then brakes to slow down to the speed specified by the DSP (usually around 0.5m/s in these situations). Thus, the control system pours a huge amount of energy into the motors to accelerate 30 pounds of metal, only to immediately pour more energy into the motors to slow it down again. This had clearly weakened the base's batteries, which are separate from the battery which runs the electronics. To make matters worse, the playroom had just been rearranged by workmen and so the freespace channel which the robot had

Place	Action	Comments
corridor	drive west	
	stop	detected person
	"Hello..."	introduction speech
	pause	visitor waves foot
	"Please stand to one side"	visitor moves
	drives forward	
	"Thank you, please follow me"	visitor follows
elevator lobby	veers right	
north wall	stop	
	turn left	avoid obstacles
	"I can avoid obstacles..."	chatter speech
kitchen	"...follow corridors..."	robot still making previous speech
intersection	"On the left here..."	kitchen place speech
	"My vision system runs..."	chatter speech
vision lab	turn left (south)	
	"On the right here is..."	Vision lab speech
	drives south	
T.V. lounge wall	stop	
	"By the way..."	chatter speech
	turn left (east)	unwedger takes over
	drives east	
couch	drives east	
end of couch	"This the T.V. lounge..."	place speech
end of lounge	"God, this place..."	chatter speech
	drives east	
Karen's office	"This is..."	place speech
	drives east	
Anita's office	"This is..."	place speech
	drives east	
playroom	turns left	
	"This is..."	place speech
	drives north	
near chair	stops	
	turns left	obstacle avoidance
	"This is the end..."	Goodbye speech
	drives	cruise mode

Figure 10-7: Transcript of the last test run.

Place	Action	Comments
west wall	stops	
	turns right	obstacle avoidance
	drives north	
north wall	stops	
	turns left	unwedger takes over
	drives west	

Figure 10-8: Transcript of the last test run (contd.).

to follow through the playroom was more complicated than usual (see figure 10-9). The complexity meant more starting and stopping. The starting and stopping made the base race, which drained the batteries further, which appeared to make the base race more. Toward the end of the run, the base was unable to accelerate as quickly as it wanted and so would regularly halt and shut down when traversing the playroom.

Apart from these problems, the robot functioned well, running roughly 30 laps. The robot made three errors. The first was due to the overhead lights having been replaced with dimmer ones. The space between two lights had a strong enough shadow on the right side to drive the robot way (the robot avoids places which are dark enough to make the edge detector fail). The robot veered to the left in a sufficiently gradual manner that it still thought it was in a straight corridor, see figure 10-10. When it finally past the shadow, if was turned far enough to the left for the right wall to be out of view. Since the right wall was out of view and it believed that it was aligned with a straight corridor, it believed that it had come a landmark and turned accordingly. Once the problem was diagnosed, it could be compensated for by placing an obstacle across from the shadow to prevent the robot from veering. This solved the problem. Unfortunately, it is hard to compensate for the new lights. Dropping the darkness threshold did in fact prevent it from making the error, but also prevented the robot from halting when it approached dark obstacles in dark areas. It appears that either brighter lights or a more sensitive camera are the only answers.

The other two problems were due to shafts of bright morning sunlight coming out of office doors. The shafts of sunlight saturate the camera pixels and appear to be obstacles. The robot halted and tried to find a way around the sunlight in vain. In one case, it tried to enter an office and grazed a doorway on its way in. Both problems were solved by removing the shafts of sunlight. A complete log of the run is given in appendix B.

10.5 Limitations, failure modes, and useful extensions

10.5.1 Low-level navigation

In general, all low-level navigation problems are obstacle detection problems. Fortunately, most of these are false positives rather than false negatives so the system is

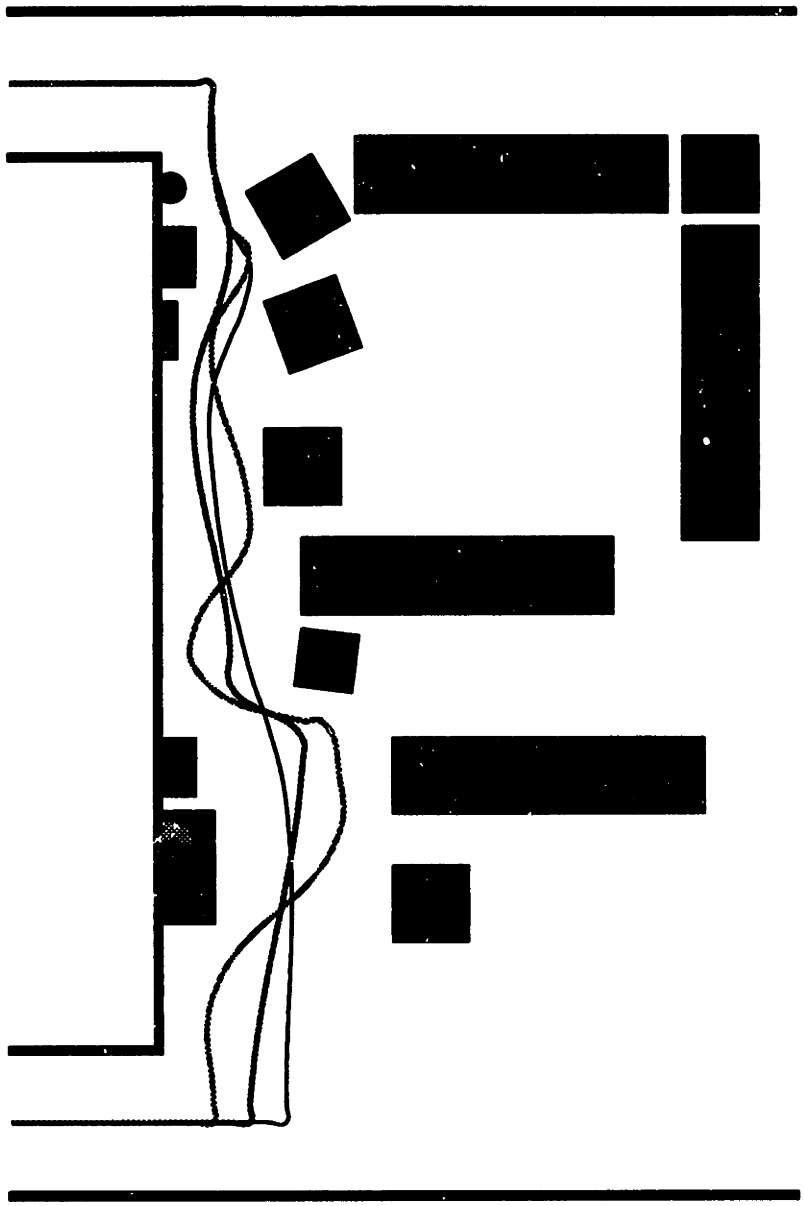


Figure 10-9: The layout of the playroom after begin rearranged by workmen, and the typical paths made by the robot. Note that NEITHER LAYOUT NOR PATHS ARE METRICALLY ACCURATE (neither could be measured accurately). Both are qualitatively accurate however, and the paths accurately represent which pieces of furniture redirected the robot's path.

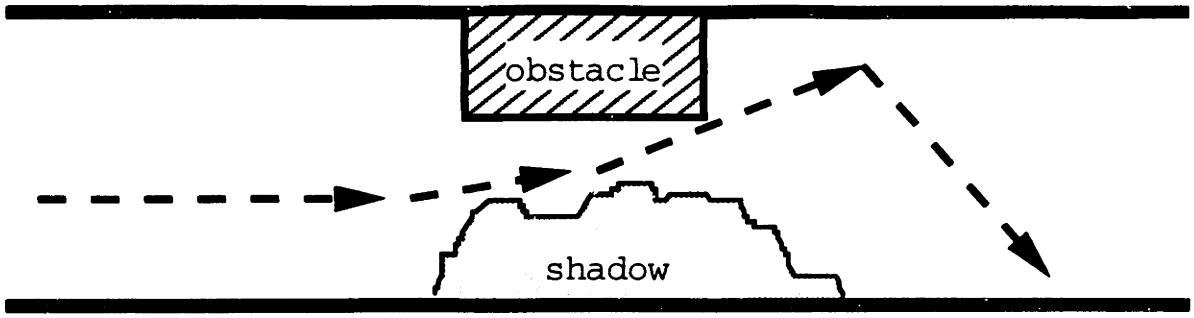


Figure 10-10: The shadow problem in the last burn-in run. When the overhead lights were replaced, one area of the corridor was dark enough that the robot could not see within it. The robot avoided the area, but lost sight of the right wall when it passed it and thought it had reached the elevator lobby. The robot then veered to its right as it should in the elevator lobby.

very conservative. The system's major failure mode is braking for shafts of sunlight. If sunlight coming through office doors in into the hallway is sufficiently strong it causes the robot to brake when there is in fact no obstacle. Shadows are an equivalent problem, but are less of a problem than one would expect because shadows are generally quite diffuse and so will not necessarily trigger the edge detector.

False negatives can be caused by a number of less common conditions. The present system also has no memory and so cannot brake for an object unless it is actually within the camera's field of view. Some of the objects in the lab have the same surface reflectance as the carpet on which they rest, so they can only be distinguished in color. Since the robot only has a black and white camera, it cannot distinguish these isoluminant edges. The edge detector can also fail in low light levels. Of course, most vision systems are likely to miss an object if they cannot even find its edges so this failure mode should not be surprising.

The other major collision avoidance problem is that the robot cannot see backward and so can sometimes back into a wall. This could be solved by adding a rear bumper.

10.5.2 The navigator

High-level navigation performance is determined by the accuracy of place recognition. In general, the system works flawlessly unless the robot gets lost. When the robot gets lost, the navigator will generally overshoot and turn around. If the robot gets severely lost, the navigator will flail around until the place recognition system gets reoriented. The worst case is when the place recognition system thinks it is east of its goal when it is actually at the western edge of the building (or west of the goal when it is east). In this case, the navigator unit and the unwedger continually make opposite course corrections. The navigator should probably be modified to give up in these situations. In general, the system would benefit from being able to explicitly notice that it's lost.

10.5.3 The unwedger

In general, the unwedger works very well. Its one failure mode appears when the robot gets into a corner. The corridor follower then tries to point the robot into the corner. When the unwedger turns the robot 45 degrees, it is insufficient to move it out of the attractor basin of the corner. Thus the robot loops, alternately turning toward and away from the corner. This could be fixed by increasing the turn angle, but that can cause problems in other situations. Perhaps the best solution would be to explicitly note the looping behavior and force a u-turn.

10.5.4 Place recognition

While recognition by matching images is quite general, it is fragile. It is particularly sensitive to changes in the world. If a chair is in view when a landmark template is made, then it must continue to be in view, and in the same place and orientation, forever. If the chair moves, then the landmark becomes unrecognizable until a new template is made. Another problem is that the robot's camera is pointed at the floor and there isn't very much interesting to be seen there. For these reasons, feature frames are preferred over image frames. At present, the only landmark which is recognized using image frames is the kitchen. In ten trials, the robot recognized the kitchen eight times while going west, and ten times while going east. Westward recognition of the kitchen fails completely when the contents of the kitchen are rearranged.

Both methods consistently miss landmarks when there is a person standing in the way. They also fail if the robot is in the process of readjusting its course after driving around an obstacle or if the corridor is very wide and has a large amount of junk in it. Both these conditions cause the constant-viewpoint constraint to fail. The former can sometimes cause the robot to hallucinate a turn because one of the walls is invisible.

Recognition of districts is very reliable, although it can sometimes become confused if the robot is driven in a cluttered open space rather than a corridor.

10.5.5 Camera limitations

Dynamic range is a major problem for vision systems. Polly suffers greatly from the fact that its camera and digitizer only have a dynamic range of 10:1 or 20:1, which means that any scene illuminated by both office lights and sunlight will necessarily contain pixels which are lost on one end of the scale. Such regions become, in a sense, invisible.

Polly also suffers greatly from its limited field of view. In spite of the fact that it has a wide field of view by most standards, it can still easily miss an obstacle which is too near it. The limited field of view is also a problem when the robot is trying to determine if it is driving through an intersection. Without being able to turn its head, it cannot look to see if it is really passing another corridor.[1] A wider field of view and/or the ability to steer the camera would be a great asset.

[1]The robot could stop dead, turn, look, and turn again, but this would both waste time, and greatly increase the risk of its getting rear-ended by a human.

Finally, I would have liked very much to have been able to use color or stereo information. I was not able to do so because of hardware limitations in the frame grabber. I believe that both of these cues could greatly improve the performance of the robot.

10.5.6 Multi-modal sensing

Given the current state of vision technology, it is a bad idea to rely exclusively on vision for obstacle avoidance, particularly vision based on a camera with the limitations discussed above. I avoided other sensing modalities because of limited time, but the use of other modalities would be essential to the development of an industrial strength version of Polly. I have found that bump switches are extremely useful. Papering one's robot with bump switches seems like a very good idea. Unfortunately, bump switches are of little or no use at high speeds. A bump switch which extends 10cm forward only gives the robot 10ms of warning of a collision when the robot is moving at 1m/s. Most robots simply cannot decelerate from 1m/s to a stop in 10ms.

It would also be good to have used translational odometry to disambiguate similar landmarks. Polly's hardware is physically capable of it, but it would have required much complicated device drivers.

Chapter 11

Summary and conclusions

11.1 Why Polly worked

Polly's efficiency and reliability are due to a number of factors. Specialization to a task allows the robot to compute only the information it needs. Specialization to the environment allows the robot to substitute simple computations for more general ones.

Polly is an existence proof that a robust system with a large behavioral repertoire can be built using simple components which are specialized to their task and environment. It also demonstrates how we can analyze specialization so that we may better understand a system and transfer the insights gained in its design to the design of other systems.

There are also a number of architectural features in Polly's favor. Polly uses multiple strategies in parallel to reduces the likelihood of catastrophic failure. If the different strategies have independent failure modes then the combination of strategies can be very reliable. When the vanishing point computation generates bad data, the depth-balancing strategy compensates for it and the distance control system prevents collisions until the vanishing point is corrected.

Polly's control and perception loops run very fast (all visual percepts and motor commands are recomputed every 66ms) so it can rapidly recover from errors. We can think of Polly's control problem as being the problem of generating the next 66ms of safe path. Even at 1m/s, 66ms is only 6.6cm or about 3 inches. In contrast, many other navigation systems, which can only process an image and make a control decision every few seconds or minutes, must compute path segments which are on the order of meters. Verifying that a 6cm path is safe is simply a lot easier than verifying that a 1m path is safe, particularly if the size of the freespace around the robot is on the order of a meter. Before committing to the 1m path, a robot needs to make very precise and reliable measurements to insure that that path will in fact be safe. Those measurements take a great deal of computing time however. But if it takes the control system ten seconds to make a control decision, then the control system would have to commit to a path at least 10m long to maintain a speed of 1m/s. But that, in turn, might require even more precise measurements, and so on.

The robot's small size significantly simplifies the navigation problem which it

faces. Office buildings are designed for creatures which take up the same footprint on the floor as normal human. The width of a doorway is made to conform to the width of a person's shoulders. My office door is 3 feet wide and my shoulders are half that. That means that I have roughly 9 inches of clearance on each side as I walk though. Even a robot which is only 2 feet in diameter has only 6 inches of clearance. That is less than the minimum operating distance for standard Polaroid sonar sensors. At distances of less than 6 inches, a robot has to start relying on very short range sensors, such as infrareds, or on tactile feedback. Tactile feedback is usually a very bad idea for a robot weighing 100 pounds or more. Polly, being only 1 foot wide, can survive a fair amount of slop in its navigation. Its small size also means that it is light enough that it can mostly ignore the issues of vehicle dynamics.

11.2 Lightweight vision

There are few things which one can really prove about vision as a whole. I do not claim that lightweight systems are a replacement for reconstruction systems. Nor do I claim to have proven the long-term efficacy of task-based vision, active vision, or qualitative vision as research strategies. This report is best taken as a reminder that there are a number of resources available to the designer of vision systems that may have been forgotten.

The most important such resource in this work has been the structure of the environment. The presence of simplifying structures in the environment often allows simple computations to be substituted for expensive ones. This simplification can be done in a principled manner by making the structures explicit in the form of habitat constraints and describing the simplification in the form of a general lemma. The advantage of the lemma is that it allows other people to perform the same optimization in the future. It also shows what is important about the simplification and what is not. The particular edge detector used by my robot for figure ground separation is unimportant. What is important is the fact that any (thresholded) linear filter restricted to the right band will do.

The structure of the task is a complementary resource. It tells the designer what information is needed and what performance is required. More importantly, it tells the design what information is *not* needed and what performance is *not* required. Computing more information means computing more representations, which is obviously more expensive, or squeezing more information into the existing representations, which is often even worse. The first principle taught in most undergraduate AI classes is that a good representation makes the important information explicit, and nothing else. As one squeezes more and more information into a representation, the information tends to become less and less explicit, meaning that further, non-trivial processing is required simply to look at the representation and extract the information which was needed in the first place. Improve performance parameters on the other hand, generally means making trade-offs, usually either trading cost for performance, or trading one performance characteristic for another. Improving unimportant performance parameters is therefore not only wasted effort, it is sometimes a waste of

money and important performance parameters

Resolution is a useful case in point. Many researchers I have talked to have taken it for granted that images below 128×128 are mostly useless, whereas Polly uses resolutions as low at 16×12. Polly demonstrates that surprisingly good results can be obtained with surprisingly low resolutions (see Horswill and Brooks [64] and Pomerleau [100] for other examples). Obviously, some tasks and environments require higher resolution, but many do not. Nor does a system need to sample everything at the same resolution. If a smooth, finely textured object moves past the camera, then the intensity field will vary rapidly in both space and time and so will have to be finely sampled to estimate the motion field. The motion field itself however, will vary much more slowly and so may not need to be sampled nearly so finely.

While it can be difficult to get people to admit to it in writing, there is a common attitude in both the AI and robotics communities that vision is not even worth considering as a sensor because of its expense and unreliability. I will refer to this as "fear of vision." Vision is, of course, often expensive of necessity. If one is faced with the task of constructing digital terrain maps from high resolution satellite images, then the only option is to build a full stereo or shape from shading system and use whatever computational resources are necessary to do the job. If one only has to make a robot drive down the corridor however, then one has many more options. A vision researcher might wish to use an expensive technique because of its intrinsic research interest. A reactive planning researcher however, who only wants a way of giving his reactive planner something to execute, will presumably be content with anything which does the job. Many such researchers have put up with sensors which give them less information than they need, simply because they felt that vision was impractical. If this work succeeds in convincing any such people that it is safe to try vision, then I will be happy.

11.3 Studying the world

I have argued that we need to study not only the structure of agents, but the structure of the environment and the relationships between agent structures and environment structures. In short, that we need to study the world. The approach I have used has been to define a formal semantics on a set of possible agents and a set of transformations over agents which preserve those semantics when some constraint holds. This approach allows the assumptions made by an agent to be separated and their computational significance drawn out. The assumptions (constraints) can then be cataloged for use in future designs and checked against different domains to make computational profiles of those domains. There are doubtless other possible approaches which should be explored.

A reasonable objection is that this project is applicable only to specialized systems, not to truly general systems. While this is true, such truly general systems are uncommon in AI. Most general vision systems assume, either implicitly or explicitly, that the world has no specular (shiny) surfaces or that surfaces are almost completely smooth. Others assume that it is piecewise planar. Most general planners assume that

the world is completely stable. That requires, among other things, that the planner be the only agent in the world. Most also assume that the world is deterministic or only boundedly unpredictable, and that the possible effects of an action are easily determined in advance. I must hasten to add that none of these are *necessarily* faults. They are simply habitat constraints like those discussed here. Such constraints can only be judged in terms of their usefulness and their match to the the agent's actual physical environment. Our choice is not whether to use constraints, but which constraints to use.

However one decides, it is important that constraints be stated explicitly and that their computational significance be made explicit. This is true whether the agents are simple reactive robots built from toy cars or complex reasoning systems implemented on supercomputers. It is also important that we understand the real underlying difficulties of the domains in which we test our systems. This can only be done by examining the world and the agent together. If we do not analyze our test domains, then we run the risk of concluding that our systems are more effective than they truly are. In short, we need to learn to be intelligent consumers of specialization.

Studying the world also has positive value for the design of general-purpose intelligent systems in that it helps make clear what tasks are hard and what tasks are easy. As the analysis of wall following demonstrates, the requirements of a task may grossly underdetermine the actual choice of mechanism. There may be a wide variety of very simple strategies which can perform the task. This is not meant to deny that problems are often hard in the general case, but rather to point out that if the vast majority of actual *instances* of a problem which are encountered by an agent in its daily life are of one or another very simple variety then the agent may thrive by using simple "hacks" when possible and saving its cognitive resources for the truly hard instances.

Perhaps the most important point is this: *AI is and must be a natural science* in the sense that it must continually make and test hypotheses about the nature of the external world. Our algorithms, representations, and formalizations of the world must eventually be compared with external reality. Doing so early reduces the risk of wasted effort. If we wish to understand intelligence then we must study not only ourselves but the world in which we live.

Appendix A

The frame database

The following is the database of frames presently in use by the robot. Each frame is defined by a **defframe** form which takes the name of the landmark, its coordinates, a set of features, and the direction the robot would be pointing if it were at that landmark and saw those features. The features can be an image, or they can be bits. The possible bits are left, right, wall, and dark (meaning there is a dark floor here. If the appearance of a place is variable, multiple frames can be used to specify the possible appearances. The **:veer** parameter is the amount to veer left or right (in degrees, right is positive), when reaching that landmark. The default veer is zero. The **speed** parameter is used to specify a string to send to the voice synthesizer when giving tours. Note that the spelling required to get the voice synthesizer to say the right thing can be rather odd. The **:place-size** parameter is the amount of time it should take the robot to drive through the landmark. The default is 2 seconds. The **:passage?** parameter means that the landmark is a grid point, and so the navigator should consider making turns at the landmark.

```
(defframe "Ian's office" ; starting point;  Ian's office.
  :x 90 :y 40
  :direction south
  :left? t)

(defframe "Ian's office"
  :x 90 :y 40
  :direction east
  :speech "This is eean's ǫffice."
  :place-size 20
  :left? t :wall? t :dark? t)

(defframe "T.K.'s office"
  :x 80 :y 40
  :direction west
  :place-size 10
  :passage? t
  :left? t :dark? t)
```

```
(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 35
  :place-size 6
  :right? t :wall? t :dark? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 45
  :place-size 6
  :right? t :wall? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 45
  :place-size 6
  :right? t :dark? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction west
  :veer 45
  :place-size 6
  :right? t)

(defframe "Elevator lobby"
  :x 60 :y 40
  :direction east
  :veer 45
  :place-size 6
  :right? t :wall? t)

(defframe "Kitchen"
  :x 50 :y 40
  :direction west
  :speech "On the left here, we have the copier room and the kitchen."
  :image ...)

(defframe "Kitchen"
  :x 50 :y 40
  :direction east
```

```
  :image ...)

;; The control algorithms loose at this junction so the
;; place size is large to give them a chance to
;; stabilize.
(defframe "Corridor 1"
  :x 40 :y 40
  :direction west
  :passage? t
  :place-size 5
  :left? t)

(defframe "Corridor 2"
  :x 30 :y 40
  :direction west
  :passage? t
  :speech "On the right here is the vizon lab."
  :left? t)

;;; Kluge: this is to deal with the effects of having
;;; Marc Raibert's door open at the end of the hall on
;;; sunny days.  It overwhelms the AGC on the camera
;;; and makes the whole corridor appear to be dark like
;;; the playroom.
(defframe "Corridor 2"
  :x 30 :y 40
  :direction west
  :passage? t
  :speech "On the right here is the vizon lab."
  :left? t :dark? t)

(defframe "Corridor 1"
  :x 40 :y 40
  :direction east
  :passage? t
  :right? t)

(defframe "Corridor 2"
  :x 30 :y 40
  :direction east
  :passage? t
  :speech "On the right here is the vizon lab."
  :right? t)

(defframe "Corridor 2"
```

```
  :x 30 :y 40
  :direction east
  :passage? t
  :speech "On the right here is the vizon lab."
  :right? t :dark? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction west
  :veer -30
  :left? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction west
  :veer -40
  :left? t :right? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction north
  :right? t)

(defframe "Marie's office"
  :x 10 :y 40
  :direction north
  :right? t :left? t)

(defframe "Conference room"
  :x 10 :y 10
  :direction west
  :right? t)

(defframe "T.V. Lounge"
  :x 30 :y 10
  :speech "This is the T.V. Lounj."
  :direction east
  :passage? t
  :left? t)

(defframe "T.V. Lounge"
  :x 40 :y 10
  :speech "This is the T.V. lounj.  We waste a lot of time here."
  :direction east
  :lefT? t :place-size 10)
```

```
(defframe "T. V. Lounge"
  :x 40 :y 10
  :direction west
  :speech "This is the T.V. lounj.  We waste a lot of time here."
  :place-size 10
  :right? t)

(defframe "T.V. Lounge"
  :x 30 :y 10
  :passage? t
  :direction west :right? t)

(defframe "Anita's office."
  :x 70 :y 10
  :direction east
  :speech "This is the office of Anneeta Flin,
           superwoman of the Nineteez."
  :left? t :place-size 3)

(defframe "Anita's office"
  :x 70 :y 10
  :direction west
  :place-size 3
  :right? t)

(defframe "Robert's office"
  :x 71 :y 10
  :direction east
  :place-size 0
  :right? t :dark? t)

(defframe "Play room"
  :x 80 :y 10
  :speech "This is the Playroom."
  :direction east
  :passage? t
  :left? t :dark? t)

(defframe "Play room"
  :x 80 :y 10
  :direction east
  :passage? t
  :speech "This is the Playroom."
  :left? t :right? t :dark? t)
```

```
(defframe "Hi Karen"
  :x 60 :y 10 :direction west
  :right? t)

(defframe "Hi Karen"
  :x 60 :y 10 :direction east
  :speech "This is Karen and Mike's office."
  :left? t)
```

Appendix B

Log of the last burn-in run

7:50 AM Make freespace channel in playroom.

Started robot.

Robot consistently veers to left by room 739, loses the wall, and mistakes the room for the elevator lobby.

Experimented with changing lighting conditions in room 739. No effect.

8:10 Have determined that the problem is due to a shadow cast by the new lights. The shadow is dark enough to force the robot to avoid it.

Problem is solved by placing a new obstacle across from the shadow to balance it. Will try dropping the shadow threshold later.

8:30 Door to room 793 is opened. Morning sun casts a band of light strong enough to be a barrier into the hallway. Robot enters office, skims doorway, and halts. I reorient and reset the base and the orbot continues. The robot believes it is at $(40, 40)$, when it is at $(30, 10)$. The place recognition system resets at the playroom and continues normally.

8:40 Uneven geometry of the playroom is making the robot do a lot of obstacle avoidance. The control system of the base is racing whenever it starts up after being blocked. No collisions so far. Damping in the DSP is compensating. Robot's path through playroom is quite variable.

8:45 Base control system seems worse. It "popped a wheelie" while accelerating from a stop in the playroom.

9:00 Another wheelie.

9:05 Base does another wheelie and shuts down. I manually reset the base and the robot continues normally.

9:07 Door to room 711 is opened. Sunlight swamps the video camera, and the robot stops and thrashes (trying to get past the sunlight). I disable the base.

9:09 Occupant of 711 leaves and closes the door. Robot restarted.

9:15 Base halts twice in a row. Possible battery problems. I reconfigure the playroom to be more corridor-like so the robot will not have to stop as much.

9:17 The robot runs the playroom without stopping.

9:20 Base halts. Restarted.

9:23 Base halts. Run terminated.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] Philip Agre and Ian Horswill. Cultural support for improvisation. In *Tenth National Conference on Artificial Intelligence*, Cambridge, MA, 1992. American Assoiciation for Artificial Intelligence, MIT Press.

[3] Philip E. Agre. The dynamic structure of everyday life. Technical Report 1085, Massachusetts Institute of Technology, Artificial Intelligence Lab, October 1988.

[4] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, 1987.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[6] John Aloimonos. Purposive and qualitative active vision. In *DARPA Image Understanding Workshop*, 1990.

[7] Yiannis Aloimonos and Azriel Rosenfeld. Computer vision. *Science*, 253:1249–1254, September 1991.

[8] Ronald C. Arkin. Motor schema based navigation for a mobile robot. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 264–271. IEEE, March 87.

[9] N. Ayache and O. D. Faugeras. Maintaining representations of the environment of a mobile robot. *IEEE Transactions on Robotics and Automation*, 5(6):804–819, 1989.

[10] Ruzena Bajcsy. Active perception vs. passive perception. In *Proc. Third IEEE Workshop on Computer Vision: Representation and Control*, pages 55–59. IEEE, October 1985.

[11] Dana H. Ballard. Animate vision. *Artificial Intelligence*, 48(1):57–86, 1991.

[12] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice Hall, 1982.

[13] A. Bandopadhay, B. Chandra, and D.H. Ballard. Egomotion using active vision. In *Proceedings, CVPR '86 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Miami Beach, FL, June 22–26, 1986)*, IEEE Publ.86CH2290-5, pages 498–503. IEEE, 1986.

[14] Stephen T. Barnard and Martin A. Fischler. Computational and biological models of stereo vision. In *Proc. DARPA Image Understanding Workshop*, September 1990.

[15] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence.* William Kaufmann, Inc., 1981.

[16] Fred Bauer and John A. Nohel. *Ordinary Differential Equations: a First Course.* W. A. Benjamin, Inc., New York, 1967.

[17] King Beach. Becoming a bartender: The role of external memory cues in a work-directed educational activity. *Journal of Applied Cognitive Psychology*, 1992.

[18] Randall Beer. A dynamical systems perspective on autonomous agents. CES 92-11, Case Western Reserve University, Cleveland, Ohio, 1992.

[19] P. Bellutta, G. Collini, A. Verri, and V. Torre. Navigation by tracking vanishing points. In *AAAI Spring Symposium on Robot Navigation*, pages 6–10, Stanford University, March 1989. AAAI.

[20] Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *Computer*, 23(12):25–37, December 1990.

[21] Andrew A. Berlin. A compilation strategy for numerical programs based on partial evaluation. TR 1144, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1989.

[22] Andrew Blake and Alan Yuille, editors. *Active Vision.* MIT Press, Cambridge, MA, 1992.

[23] David J. Braunegg. Marvel: A system for recognizing world locations with stereo vision. Technical report, MIT Artificial Intelligence Laboratory, 1990.

[24] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automoation*, 2(1):14–23, March 1986.

[25] Rodney A. Brooks. The behavior language; user's guide. AI Lab Memo 1227, MITAI, Apr 1990.

[26] Christopher Brown, David Coombs, and John Soong. Real-time smooth pursuit tracking. In Blake and Yuille [22], pages 126–136.

[27] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[28] David Chapman. Intermediate vision: Architecture, implementation, and use. TR 90-06, Teleos Research, 1990.

[29] David Chapman. Vision, instruction, and action. Technical Report 1204, Massachusetts Institute of Technology, Artificial Intelligence Lab, April 1990.

[30] David Chapman. *Vision, Instruction, and Action.* MIT Press, 1992.

[31] Noam Chomsky. *Aspects of the Theory of Syntax.* MIT Pres, Cambridge, MA, 1965.

[32] Jonathan H. Connell. *Minimalist Mobile Robotics.* Academic Press, 1990.

[33] David Coombs and Karen Roberts. "bee-bot": using peripheral optical flow to avoid obstacles. In *Proc. of the SPIE Conf. on Intelligent Robots and Computer Vision XI: Algorithms, Techniques, and Active Vision, (Boston, MA, November 15–20, 1992)*, 1992.

[34] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[35] Jill D. Crisman. Color region tracking for vehicle guidance. In Blake and Yuille [22], chapter 7.

[36] James E. Cutting. *Perception with an Eye for Motion.* MIT Press, 1986.

[37] DARPA SISTO. *Proceedings of the 1993 DARPA Image Understanding Workshop*, Washington, D.C., 1993. Morgan Kaufman.

[38] D. DeMenthon. A zero-bank algorithm for inverse perspective of a road from a single image. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 258–263. IEEE, March 1987.

[39] Ernst D. Dickmanns. Expectation-based dynamic scene understanding. In Blake and Yuille [22], chapter 18.

[40] Michael Dixon. Embedded computation and the semantics of programs. TR SSL-91-1, Xerox Palo Alto Research Center, Palo Alto, CA, September 1991.

[41] Bruce Randall Donald and James Jennings. Constructive recognizability for task-directed robot programming. *Robotics and Autonomous Systems*, 9:41–74, 1992.

[42] Jon Doyle. A truth maintenance system. In Webber and Nilsson [137], pages 496–516.

[43] Sean P. Engelson and Drew McDermott. Image signatures for place recognition and map construction. In *Proceedings SPIE Symposium on Intelligent Robotic Systems, Sensor Fusion IV*, November 1991.

[44] Marc H. Raibert *et al.* Dynamically stable legged locomotion. TR 1179, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, September 1989.

[45] Jerome A. Feldman. Four frames suffice: A provisionary model of vision and space. TR 99, Computer Science Department, University of Rochester, Rochester, NY 14627, September 1982.

[46] Richard Fikes, Peter HArt, and Nils Nilsson. Learning and executing generalized robot plans. In Webber and Nilsson [137], pages 231–249.

[47] R. James Firby. Adaptive execution in complex dynamic worlds. YALEU/CSD/RR 672, Computer Science Department, Yale University, 1989.

[48] Margaret M. Fleck. Boundaries and topological algorithms. TR 1065, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[49] Erann Gat. Bdl: A language for robotic control systems. Jet Propulsion Laboratory, 1990.

[50] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings, AAAI-92*, 1992.

[51] Andrew S. Gavin and Masaki Yamamoto. A fast, cheap, and easy system for outside vision on mars. In *to appear in Intelligent Robots and Computer Vision XI*, Cambridge, MA, September 1993. SPIE.

[52] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682, 1987.

[53] J. J. Gibson. *The Senses Considered as Perceptual Systems.* Houghton-Mifflin, Boston, 1966.

[54] J. J. Gibson. *The Ecological Approach to Perception.* Houghton-Mifflin, Boston, 1979.

[55] Richard W. Hamming. *Coding and Information Theory.* Prentice Hall, Englewood Cliffs, N.J. 07632, 1980.

[56] Kristian J. Hammond and Timothy M. Converse. Stabilizing environments to facilitate planning and activity: An engineering argument. In *Ninth National Conference on Artificial Intelligence*, pages 787–793, Menlo Park, CA, July 1991. American Association for Artificial Intelligence, AAAI Press.

[57] Barbara Hayes-Roth. Selective perception in an intelligent monitoring agent. In Simmons [113], pages 66–69.

[58] James A. Hendler and James Sandborn. A model of reaction for planning in complex environments. In *Proceedings of the Knowledge-Based Planning Workshop*, Austin, TX, December 1987. DARPA.

[59] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[60] B. K. P. Horn. *Robot Vision.* MIT Press, 1986.

[61] Ian Horswill. The senselisp programmer's manual. Unpublished technical note, MIT Artificial Intelligence Laboratory, 1989.

[62] Ian Horswill. How to hack yourself senselisp. Unpublished technical note, MIT Artificial Intelligence Laboratory, March 1990.

[63] Ian Horswill. Proximity detection using a spatial filter tuned in three-space. In *Proceedings of the 1991 AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, 1991.

[64] Ian Horswill and Rodney Brooks. Situated vision in a dynamic environment: Chasing objects. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, August 1988.

[65] Ian D. Horswill. Reactive navigation for mobile robots. Master's thesis, Massachusetts Institute of Technology, June 1988.

[66] Katsushi Ikeuchi and Martial Herbert. Task oriented vision. In *DARPA Image Understanding Workshop*, 1990.

[67] Leslie P. Kaelbling. Goals as parallel program spcifications. In *Proceedings, AAAI-88*, pages 60–65, St. Paul, MN, 1988.

[68] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[69] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. A spectrum of abstraction hierarchies for planning. In *Proceedings of AAAI-90*, 1990.

[70] David Kortencamp. Applying computational theories of cognitive mapping to mobile robots. In Marc Slack and Erann Gat, editors, *Working notes of the AAAI Spring Symposium on Control of Selective Perception*, pages 83–89. AAAI, Cambridge, Massachusetts, 1992.

[71] A. Kosaka and A. C. Kak. Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties. *Computer Vision, Graphics, and Image Processing*, 56(3), September 1992.

[72] David J. Kriegman and Ernst Triendl. Stereo vision and navigation within buildings. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 402–408. IEEE, March 87.

[73] David J. Kriegman, Ernst Triendl, and Tomas O. Binford. A mobile robot: Sensing, planning and locomotion. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 402–408. IEEE, March 87.

[74] B. Kuipers. Representing knowledge of large-scale space. Technical Report AI-TR-418, Massachusetts Institute of Technology, Artificial Intelligence Lab, 1977.

[75] Benjamin J. Kuipers and Yung-Tai Byun. A robust, qualitative approach to a spatial learning mobile robot. In *SPIE Advances in Intelligent Robotics Systems*, November 1988.

[76] Christopher G. Langton. Artificial life. In *Artificial Life* [77], chapter 1.

[77] Christopher G. Langton, editor. *Artificial Life*, volume VI of *Santa Fe Institute Studies in the Sciences of Complexity.* Addison Wesley, New York, 1989.

[78] Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors. *Artificial Life II*, volume X of *Santa Fe Institute Studies in the Sciences of Complexity.* Addison Wesley, New York, 1992.

[79] John E. Larid, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[80] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.

[81] Hector J. Levesque and Ronald J. Brachman. A fundamental tradeoff in knowledge representation and reasoning (revised edition). In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 42–70. Morgan Kaufman, Los Altos, CA, 1985.

[82] T. Lozano-Pérez, M. Mason, and R. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1), 1984.

[83] David G. Luenberger. *Introduction to Dynamic Systems: Theory, Models, and Applications.* John Wiley and Sons, 1979.

[84] Kevin Lynch. *The Image of the City.* MIT Press, 1960.

[85] D. M. Lyons and M. A. Arbib. A formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation*, 5(3):280–293, 1989.

[86] Pattie Maes. Behavior-based artificial intelligence. In Meyer and Wilson [90].

[87] David Marr. *Vision.* W. H. Freeman and Co., 1982.

[88] Maja J. Mataric. Minimizing complexity in controlling a collection of mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 830–835, Nice, France, May 1992.

[89] Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, Massachusetts, 1991.

[90] Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats: The Second International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, Massachusetts, 1993.

[91] George A. Miller, Eugene Galanter, and Karl H. Pribram. *Plans and the Structure of Behavior.* Henry Holt and Company, 1960.

[92] Hans P. Moravec. The stanford cart and cmu rover. Technical report, Robotics Institute, Carnegie-Mellon University, February 1983.

[93] Randal C. Nelson. Visual homing using an associative memory. In *Proceedings of the DARPA Image Understanding Workshop*, pages 245–262, 1989.

[94] Allen Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, MA, 1990.

[95] ed. Nils J. Nilsson. Shakey the robot. Technical Report 323, SRI International, April 1984.

[96] H. Keith Nishihara. Minimal meaningful measurement tools. TR 91-01, Teleos Research, 1991.

[97] Romedi Passini. *Wayfinding in Architecture*, volume 4 of *Environmental Design Series.* Van Norstrand Reinhold, New York, 1984.

[98] Roger Penrose. *The Emperor's New Mind.* Oxford University Press, 1989.

[99] T. Poggio and V. Torre. Ill-posed problems and regularization analysis in early vision. In Lee S. Baumann, editor, *Image Understanding Workshop (New Orleans, LA, October 3-4, 1984)*, pages 257–263. Defense Advanced Research Projects Agency, Science Applications International Corp., 1984.

[100] Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1), 1991.

[101] Louise Pryor and Gregg Collins. Planning to perceive: a utilitarian approach. In Simmons [113], pages 113–122.

[102] W.V. Quine. *Philosophy of Logic.* Harvard University Press, 2 edition, 1970.

[103] Douglas A. Reece and Steven Shafer. Active vision at the system level for robot driving. In Simmons [113], pages 70–77.

[104] Daniel Reisfeld, Haim Wolfson, and Yehezkel Yeshurun. Detection of interest points using symmetry. In *Proceedings of the Third International Conference on Computer Vision*, pages 62–65, Osaka, Japan, December 1990. IEEE Computer Society.

[105] Chirstopher K. Riesbeck and Roger C. Schank. *Inside Case-Based Reasoning.* Erlbaum, Hillsdale, N.J., 1989.

[106] Paul S. Rosenbloom. *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 1983.

[107] Stanley J. Rosenschein. Formal theories of knowledge in ai and robotics. report CSLI-87-84, Center for the Study of Language and Information, Stanford, CA, 1987.

[108] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of machines with provable epistemic properties. In Joseph Halpern, editor, *Proc. Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98. Morgan Kaufmann, 1986.

[109] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2), 1974.

[110] David A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., Newton, Massachusetts, 1986.

[111] Uwe Schnepf. Robot ethology: A proposal for the research into intelligent autonomous systems. In Meyer and Wilson [89], pages 465–474.

[112] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings 10th IJCAI*, pages 1039–1046, 1987.

[113] Reid Simmons, editor. *Working notes of the AAAI Spring Symposium on Control of Selective Perception*, Stanford, California, 1992. American Association for Artificial Intelligence.

[114] Reid Simmons, Long Ji Lin, and Chris Fedor. Autonomous task cotrol for mobile robots. In *Proc. IEEE Symposium on Intelligent Control*, Philadelphia, PA, September 1990.

[115] Herbert A. Simon. *Sciences of the Artificial.* MIT Press, Cambridge, Massachusetts, 1970.

[116] Lee Spector and James Hendler. The supervenience architecture. In Avi Kak, editor, *Working notes of the AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, pages 93–100. AAAI Press, Asilomar, California, 1991.

[117] Anselm Spoerri. The early detection of motion boundaries. Technical Report 1275, MIT Artificial Intelligence Laboratory, 1991.

[118] Guy L. Steele Jr. RABBIT: A compiler for SCHEME (a study in compiler optimization). TR 474, MIT AI Lab, May 1978.

[119] K. Storjohann, T. Zeilke, H. A. Mallot, and W. von Seelen. Visual obstacle detection for automatically guided vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 761–766, May 1990.

[120] Lucy Suchman. *Plans and Situated Action.* Cambridge University Press, 1987.

[121] Michael J. Swain. Color indexing. Technical Report 390, University of Rochester Computer Science Department, November 1990.

[122] Michael J. Swain. Active visual routines. In Simmons [113], pages 147–149.

[123] W.B. Thompson and J.K. Kearney. Inexact vision. In *Workshop on Motion: Representation and Analysis*, 1986.

[124] W.B. Thompson, K.M. Mutch, and V.A. Berzins. Dynamic occlusion analysis in optical flow fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7:374–383, 1985.

[125] C. E. Thorpe. *FIDO: Vision and Navigation for a Robot Rover.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, December 1984.

[126] Learning to Act using Real-Time Dynamic Programming. Andrew g. barto and steven j. bradtke and satinder p. singh. Unpublished.

[127] Where to Look Next Using a Bayes Net: The TEA-1 System and Future Directions. Raymond d. rimey. In Simmons [113], pages 118–122.

[128] Peng-Seng Toh and Andrew K. Forrest. Occlusion detection in early vision. In *Proceedings of the International Conference on Computer Vision*, 1990.

[129] John K. Tsotsos. Analyzing vision at the complexity level. *Behavioral and Brain Sciences*, 13(3):423–469, 1990.

[130] Matthew A. Turk, David G. Morgenthaler, Keith Gremban, and Martin Marra. Video road following for the autonomous land vehicle. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 273–280. IEEE, March 1987.

[131] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

[132] F. J. Varela and P. Bourgine, editors. *Toward a Practice of Autonomous Systems: the Proceedings of the First European Conference on Artificial Life.* MIT Press, Cambridge, MA, 1992.

[133] R. Wallace. Robot road following by adaptive color classification and shape tracking. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 258–263. IEEE, March 1987.

[134] R. Wallace, K. Matsuzaki, Y. Goto, J. Crisman, J. Webb, and T. Kanade. Progress in robot road-following. In *1986 IEEE Internation Conference on Robotics and Automation*, pages 1615–1621, April 1986.

[135] R. Wallace, A. Stenz, C. Thorpe, H. Moravec, W. Whittaker, and T. Kanade. First results in robot road-following. In *IJCAI-85*, 1985.

[136] A. M. Waxman, J. LeMoinge, and B. Srinivasan. Visual navigation of roadways. In *1985 IEEE Internation Conference on Robotics and Automation*, April 1985.

[137] Bonnie Lynn Webber and Nils J. Nilsson, editors. *Readings in Artificial Intelligence.* Morgan Kaufman, Los Altos, CA, 1981.

[138] Norbert Wiener. *Cybernetics.* MIT Press, Cambridge, 1961.

[139] Lambert E. Wixson. Detecting occluding edges without computing dense correspondence. In IU93 [37], pages 933–938.

[140] John Woodfill and Ramin Zabih. Using motion vision for a simple robotic task. In *AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, 1991.