Imprinting in Robots

FINAL PROJECT
6.836 EMBODIED INTELLIGENCE

Nick Feamster
feamster@lcs.mit.edu

Jaeyeon Jung
jyjung@lcs.mit.edu

May 17, 2001

Contents

List of Figures

Chapter 1

Introduction

The main objective of our project is to design and build a robot that can emulate the filial imprinting process of an animal. Filial imprinting is the process by which young species learn to recognize their parents. It consists of seeking parents, followed by a decision-making process to link perceptual experience to one individual. After an animal undergoes filial imprinting, it follows the imprinted object and learns skills crucial to survival throughout its life.

Motivated by this, our project develops an artificial creature that focuses on the following sub-goals.

1. To understand how robots synthesize the information they receive from multiple sensors to follow imprinting objects and navigate obstacles in a dynamic environment.

2. To get a better intuition for how *sign stimuli* work in real animals. Sign stimuli are predilections toward the same species. They are used in the imprinting process to prevent the young from being imprinted onto the wrong object. In a robot, this can be implemented with internal templates stored in memory and applying primitive rules to determine whether or not a particular subject is a *match* for imprinting.

3. To investigate the use of environmental cues to identify and follow *mother*. Along with visual perception, there are numerous features that an animal uses to identify and follow its mother. This includes auditory, olfactory, and gustatory components and helps the young animal select the correct individual for visual imprinting. We emulate this behavior by using different kinds of sensors and various linking of functional units.

1.1 Imprinting Process

Imprinting is a well-known example of genetic influences on learning. It occurs during a sensitive period of development, when young animals learn rapidly to distinguish their parents from all other individuals. This process is essential for the survival as the young learn important skills from parents such as hunting for food and avoiding a predator.

Figure 1-1 shows the test done by Ramsey and Hess [7]. Ducklings were exposed to several models of a male duck. Imprinting was tested 5 to 7 hours later by offering the duckling a choice between a female duck and a male duck which was previously presented. Although the female duck was closer and louder, the duckling responded to the imprinted object, in this case the male duck.

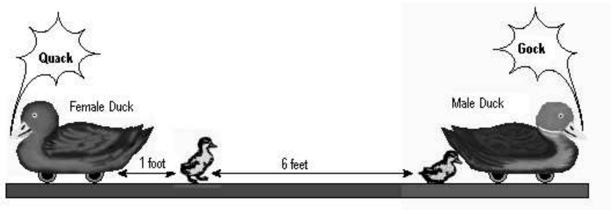

Figure 1-1: Filial Imprinting. A duck learns to follow the animal that it is imprinted on very early in life. Here, the baby duckling has been imprinted on the male duck, and thus will follow the male, even though the female duck is closer and louder.

To identify parents, animals utilize environmental cues such as visual, auditory, olfactory, and gustatory features of other objects. They also exhibit a preference to imprint on normal stimuli. Specifically, when given a choice, babies in the process of imprinting attend preferentially to images that more closely resemble members of their own species. This is based on genetically programmed preferences for simple, conspicuous features, and referred to as *sign stimuli* by ethologists [7].

Two distinct kinds of imprinting can be identified. Filial imprinting occurs soon after birth and ensures young birds remain close to their mother. On the other hand, sexual imprinting occurs later in development and determines the choice of a mate. It is different from filial imprinting in that the animal's preference is generalized to all individuals that are similar, rather than remaining specific to a single individual. It also involves special actions to entice a mate such as singing and dancing. In this project, we focus on the emulation of *filial imprinting*.

Due to limited resource and time constraint, we focus on the filial imprinting and the use of sign stimuli for recognition in this project.

1.2 Our Approach

To emulate filial imprinting, we use a subsumption architecture to control a robot. A robot should be able to detect an object and follow it based on sensory inputs. It is also equipped with bump sensors in order to avoid obstacles. The following outlines several behaviors to be implemented within the robot.

1. **Moving:** The robots should be able to move around. This includes moving forward and backward and changing an orientation.

2. **Avoiding obstacles:** When a robot runs into a wall or other objects, it should be able to back off and take another route.

3. **Seeking its mother:** A robot wanders around to find parents. This involves a pattern recognition and matching process with an internal state to decide if the object is a suitable parent.

4. **Following:** The robot should be able to follow a parent by keeping track of its motions. Our robot uses a combination of ultrasound and radio frequency (RF) sensors to estimate a distance between a target and itself and uses patterns in this distance information to devise a strategy for following its mother.

Chapter 2

Design

This chapter addresses software and hardware considerations we made in order to implement a robot described in Chapter 1. The control logic is developed based on a subsumption architecture that provides an incremental method for building an intelligent system. Considering that imprinting behavior depends heavily on the dynamics of the interaction of a creature with the environment, the perceptual system, central system, and actuation system are indeed closely intertwined in our robot. Hence, the subsumption architecture is a choice of our design as the decomposition can be easily done in terms of behavior generating module.

2.1 Subsumption Architecture

Our robot is programmed in the subsumption architecture. The figure 2-1 shows a network of 9 augmented finite state machines (AFSM), including 4 machines that take sensory inputs and one that controls actuators.

1. **Motor Control:** For the locomotion of a robot, we use a wheeled vehicle with two differential driven wheels and one caster wheel. This AFSM takes inputs for each motor and applies an appropriate voltage to the motors. To supply continuous locomotion, it is periodically triggered except when a robot should be stationary. If there is no input, it uses the last value assigned to each motor.

2. **Move:** The basic capability of a robot is to move around. Without any intervention, the *Move* AFSM simply directs *Motor Control* to move the robot forward at a fixed speed.

3. **Recognize:** The camera monitors and sends an activity message to *Follow* and *Detect Collision* AFSMs when it finds a matching target object. The robot stores a captured image as a template and uses this for pattern matching. The camera faces ahead and takes a snapshot

at regular intervals. As soon as the captured image sample matches the template, *Recognize* initiates the *Follow* and *Detect Collision* AFSMs and stops the recongnition procedure.

4. **Follow:** This controls the speed and the orientation of a robot based on the distance between a target and itself. A target object broadcast a signal so that a robot can determine the distance and the speed that the target is moving. It periodically takes a distance information and keeps last five values to estimate the velocity of a moving target and changes in relative distance between it and its target. To obtain an accurate value from the measurement and additionally to prevent unnecessary collisions, it inhibits *Move* while it is obtaining readings. The farther it is away from the target, the larger the output speed is (to enable Braitenberg-style following). It outputs a zero speed if it is closer than the certain distance to prevent from surpassing the target. One interesting module is *U-turn*. It may happen that a robot is moving in an opposite direction against a target, which it can determine by comparing a current distance with previously captured distances. It then issues a command to the *U-turn* AFSM so that a robot can reverse the orientation and move along with the target.

5. **Speed:** It takes inputs from *Follow* and passes appropriate instantaneous speed to *Motor Control*. This AFSM also suppresses *Move* when it needs to change the speed of a robot. This assumes that the speed is proportional to the amount of voltage applied to the motor and set the same amount to the each motor.

6. **Shaft Encoding:** The *Shaft Encoder* AFSM reads the left and right break-beam sensors and outputs running totals for each. The break-beam sensors used in the robots are level-triggered, so the software must keep track of the last sensed value to get an edge-triggered. Using an edge-triggered scheme is important for correctness of the shaft encoding algorithm. It provides counts and velocity of left and right wheels to *U-turn* and *Orientation* AFSMs.

7. **Orientation:** This AFSM enables the robot to travel in a straight line. Even though the same voltage is applied to each motors, the speed of each wheel will vary depending on the friction, different loads on each wheel, the distribution of weight on the robot's body, and obstacles that may encounter while the robot is underway. Therefore, it monitors motor speed by referring shaft encoder and adjusts the slower one by suppressing the other one till they will have an equal count. It is also called at regular interval.

8. **U-turn:** When *U-turn* gets a message from the *Follow* AFSM and it suppresses *Speed* and *Orientation* and makes a robot turn 180 degree around. It turns off both motors and resets counters for the right wheel. It issues a command to *Motor Control* so that a motor drives only the right wheel at a fixed speed, which makes a robot slowly turn around. When the counter reaches a value that corresponds to the number of counts for 180 degree, it returns a

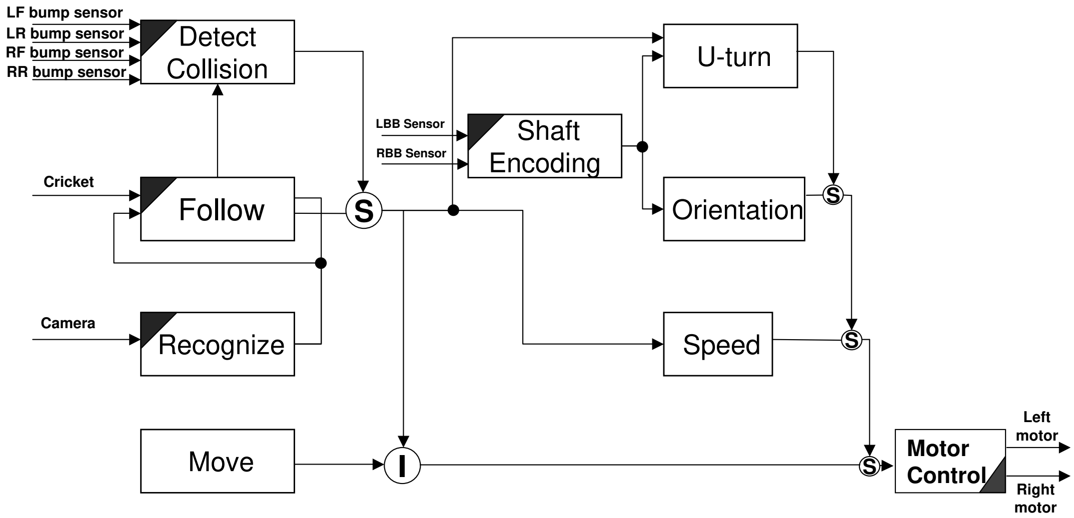

Figure 2-1: Subsumption Architecture. Our architecture has a network of 9 augmented finite state machines (AFSM), including 4 machines that take sensory inputs and one that controls actuators. Primitive locomotion is subsumed by various obstacle avoidance and following behaviors.

control back and waits for the next call.

9. **Detect Collision:** Whenever this AFSM notices that any bump sensor is pressed, it forces a robot to reverse its direction (turning slighly in one direction or another with a random perturbation) by suppressing the signal coming from the *Follow* AFSM.

2.2 Hardware Architecture

To implement a physical artificial creature, we first consider the hardware requirements and functionality that each component can provide. A robot, as a collection of these sensors, actuators and computational elements, must be organized in a way that it can intelligently accomplish tasks based on program and stimuli. In this section, we briefly review each subsystem and discuss design issues, taking into consideration how components can be integrated into a system to achieve perception, provide power, and achieve locomotion.

2.2.1 Computational Hardware

Computational hardware will be used as a platform to process data inputs, drive actuators and control the behavior of a robot. The computational elements include a microprocessor, memory, and programming language in which the control logic will be described. A microprocessor comes with the instruction set, which defines basic operations on data inputs. Each instruction is executed according to the orders specified in a program and results in some change in the state of microprocessor. The program along with the data is stored in the internal memory on the chip to be processed.

The hardware determines a robot's ultimate potential in a sense that the performance largely depends on the processing power of the microprocessor and the number of input and output ports are fixed once a platform is chosen. In addition, limited functionality can restrain simulating some behaviors. For this project, the capability to extend physical memory is crucial because a template image should be stored on the board and the image data typically requires 16KB at a minimal resolution (8 bit per pixel and 128×128 pixel resolution). The following list summarizes requirements for the hardware platform.

1. **Moderate performance:** A robot should be able to monitor sensors and process the input data in real-time. In particular, image processing is computationally intensive and a processor should be fast enough to handle this in a reasonable amount of time.

2. **Extensibility:** The number of input and output ports required depends on sensors and actuators. Since the type and the number of sensors and actuators can be changed during the project, the board should be able to accommodate these changes gracefully. The expansion slot for the extra input and output ports would be required. Unless the circuit board has a sufficient amount of memory, it should also support a way to extend a memory. The minimum requirement is 8Kb assuming that an image can be compressed without much degradation. In most cases, replacing on-chip memory is not recommended and could result in unstable operations.

3. **Availability:** The hardware elements should be able to obtain easily and user guide or technical specificaiton should be accessible. To complete the project on time, it is also critical to get the necessary items within the first quarter of project time-line.

4. **Ease of programmability:** Building a robot usually involves both some assembly language programming and some high-level language programming. As well as saving a time for coding, a high-level language is extremely useful when debugging a robot program. Using a high-level language implies the ability to convert the program into a set of instructions supported by a hardware. This can be done with the help of either a compiler and a linker or an an interpreter.

2.2.2 Sensors

A robot understands and is aware of its environment throughout the sensors. Sensors convert some physical phenomena into electrical signals that the microprocessor can read. This might be done by using an analog-to-digital converter on circuit board, by loading a value from an input/output port, or by using an external interrupt. In particular, we are concerning following sensors to implement filial imprinting behavior in a robot.

1. **Camera:** To enable the vision of our robot, we need a camera that takes a still image and outputs the image data according to a clock signal. Due to a limited size of a robot, a camera should be as small as it can be easily mounted on the robot. The another recommended feature is a low power consumption. As many devices compete for the common power supply, a camera with low power can lengthen the operation time of the robot.

2. **Light and sound sensors:** We are concerning light and sound sensors to measure a distance between a robot and a target object. The Cricket location support system [6] developed in the Network and Mobile Systems group at LCS provides a cost effective solution with a reasonable degree of accuracy. It uses a combination of radio frequency (RF) and ultrasound to estimate the distance between two points. With each RF advertisement, the beacon simultaneously transmits an ultrasonic pulse. Beacon receivers listen for RF signals, and upon receipt of the first few bits, listen for the corresponding ultrasonic pulse. When this pulse arrives, they obtain a distance estimate for the corresponding beacon based on the difference in arrival time between the RF and ultrasonic pulses. We attach a beacon transmitter to the target object and receiver to a robot so that a robot can determine distance information from the receiver whenever it is needed. The output information should be either output in serial or parallel digitized depending on the number of available input ports and board specification. While a parallel distance output is far simpler to read and thus reduces complexity, it also requires a very wide data bus. The range of operation can be an order of tens of feet and a distance unit can be as low as a couple of inches.

3. **Position and orientation:** A shaft encoder is a sensor that measures the position or rotation rate of a shaft. Typically, a shaft encoder is mounted on the output shaft of a drive motor or on an axle. The signal delivered by this sensor can be either a code that corresponds to a particular orientation of the shaft or it may be a pulse train. A robot can get feedback from a shaft encoder on how far its wheels have turned or on synchronizing two wheels' velocity [4]. As discussed earlier, a robot uses shaft encoder sensor inputs when it turns a fixed angle and also when it adjusts its orientation to move along a stright line. Though there are more accurate sensors available, such as gyros, tilt sensors and compasses, a simple shaft encoder provides sufficient functionality for our purposes.

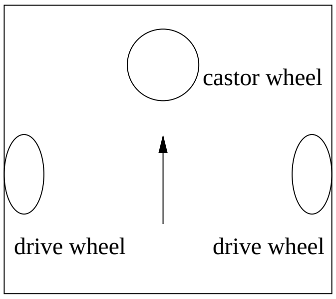

Figure 2-2: Bottom view of wheel arrangements. The differential drive wheel arrangement with one castor wheel enables our robot to turn on a very small radius and thus makes obstacle avoidance easier.

4. **Bump sensors:** To detect collisions, at least two bump sensors are required for both front and rear part of the robot chassis. As it is designed to move either forward or backward, bump sensors at the side can be optional. The sensors should be mounted in such a way that, when the robot contacts an object, one or two switches will close, thus indicating close proximity of an obstacle to the robot. The bump sensor should be able to trigger the switch ahead of *real collision* and absorb the impact of a collision until a robot responds that. The coverage of the sensor is also important. The sensors are usually triggered when the force is applied along with the lever. Therefore, the size and shape of the sensors matters in the design of a robot and the the dimensions of the chassis affect the choice of bump sensors, vice-versa.

2.2.3 Locomotion

As for a locomotion system, we designed a wheeled vehicle with two differential drives and one castor wheel. The differential drive scheme consists of two wheels on a common axis, with each wheel driven independently. This enables a robot to drive straight, to turn in place, and to move in an arc. Although differential drive is less complicated and easy to program, it has some important design problems, such as ensuring balance, moving in a straight line, etc.. Some additional supports, besides the two drive wheels are needed to maintain the stability. We use a castor wheel from a rotating stool for this purpose. The cast wheel is arranged in a triangle pattern as shown in figure 2-2. This design enables our robot to turn with incredibly small radius and thus makes obstacle avoidance easier.

Another design consideration for differentially driven robots is how to make the robot go straight. As described in the previous section, even when the same voltage is applied to motors, they will turn at different speeds and the robot will veer to one side or the order. To make the robot go stright, motor velocity must be controlled dynamically in the software and we implement this routine in the *Orientation* AFSM.

Chapter 3

Implementation

The implementation of our robot can be classified into several sections. The first is the implementation of the architecture of the robot itself: the parts we chose to use, etc. To build the physical embodiment of our robot, we use LEGO bricks; this gave us ample flexibility to change the architecture as we made design changes along the way.

The other, more interesting aspect of the implementation is the aspects that control the physical body of our robot. Specifically, the control aspects of our robot can be reduced to several main components:

- **Vision:** How the robot sees objects around it. The vision sensing of our robot was used for two main purposes: to develop the template that our robot then used as a basis for comparison in order to determine what an appropriate *mother* might be, and to capture views of its surrounding environment to determine whether or not what it sees at any given time might be its mother. For the vision component of our robot, we use the Mitsubishi artificial retina chip that is included as part of the Gameboy camera.

- **Environmental cues:** Once our robot detects its mother, it must use environmental cues that exist in order to develop a way to follow its mother using other sensations than vision (this is partly because our implementation of the vision sensor performs so slowly). To enable our robot to process environmental cues, we use the Cricket location support system as a distance detection mechanism.

- **Primitive survival:** At a minimum, the robot should be able to travel in a straight line, determine the direction that it is going, and change its direction accordingly. Additionally, it should be able to avoid obstacles. In order to enable primitive survival, we give our robot elementary bump sensing and shaft encoder inputs.

- **Logical processing:** The robot must be able to process all of its sensory inputs and arbitrate

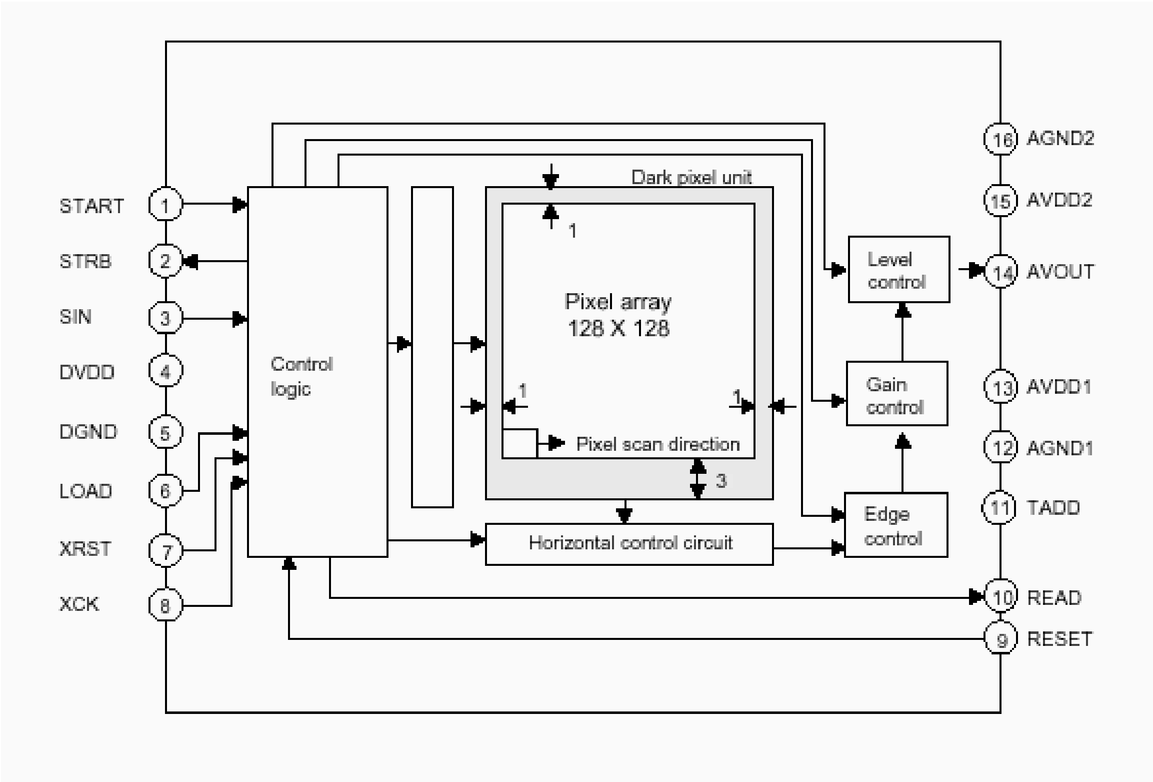

Figure 3-1: Block diagram of the Mitsubishi M64283FP artificial retina chip [5].

among them to make appropriate behavioral decisions. To enable the "brains" of our robot, we use the Handy Board, which includes a Motorola 6811 microprocessor that is programmable with Interactive C.

The following sections describe these components in further detail and describe the protocols we used to communicate with each sensory input.

3.1 Vision

To implement vision on our robot, we used the Misubishi M64283FP artificial retina chip. This chip is used in the Gameboy camera, and thus was an economical and fairly simple way of obtaining a vision sensor that is capable of being controlled with digital logic.

The M64283FP artificial retina chip is a CMOS image sensor of 128 $\times$ 128 pixels that supports various image processing functions and analog signal calibration. It also supports parallel processing, just like the human retina. Because of its low-power characteristics (around 15 milliwatts), it was also very attractive for use on a robot, where power supply is limited.

Figure 3-1 shows a block diagram of the M64283FP artificial retina chip that we use to capture 128×128 pixel greyscale images from the environment. Initially, the chip is reset by clocking pin 9,

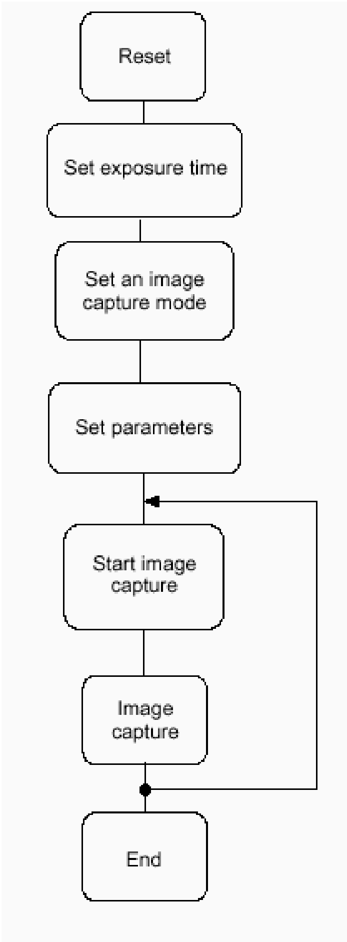

Figure 3-2: Operational flow diagram of the Mitsubishi M64283FP artificial retina chip [5].

the RESET pin, and ensuring that the START, SIN, LOAD, and XCK pins were all low. Subsequently, we set the registers accordingly to initialize image capture by setting appropriate digital values out on the serial input pin, SIN and loading them serially by clocking the LOAD pin.

Figure 3-2 shows a diagram of how the M64283FP artificial retina chip is operated. Initially, the camera is reset, and registers are set to the appropriate values to set the exposure time and the image capture mode for the chip. Once initial parameters are set up, the camera can operate in the image capture realm, repeatedly capturing images upon repeated requests. We initialize all relevant parameters upon the robot "waking up" (i.e., power on), and then have the robot operate in image capture mode to capture image templates and capture the present view of the world.

3.1.1 Image Capture

Once the camera is initialized, it can take many images sequentially on command. In order to take a picture, the START pin is set high, the XCK pin is clocked, and the START pin is cleared again. This begins the image capture process. Then, we clocked the retina chip in a tight loop until the

Figure 3-3: Sample picture taken by Gameboy Camera with Handy Board Interface.

READ pin went low (this pin is *active low*). When the READ pin goes low, this indicates that the chip can being reporting each pixel value. On each subsequent XCK clock signal, a new pixel value becomes available on the AVOUT pin. Note that this is an analog reading between 0 and 5 Volts; we use the built in A/D converter on the Handy Board to convert each pixel to an 8-bit value. Thus, each image could be accurately represented in 16 KB of memory.

We used the freely-available Acarv [2] program to read in values from the serial port from the Handy Board to verify that our image capture routines were working properly.

The speed of the image capture was limited by the speed of the Handy Board. The Handy Board controller reportedly runs at 1 MHz, but the overhead of the Interactive C interpreter seemed to slow the performance considerable. The average time to capture one 128×128 pixel image was approximately 1 minute (far too slow to perform dynamic operations using image processing routines). We have included a few sample images that were taken with this retina chip as controlled by the Handy Board. Appendix A.1 has the code that we used to perform image capture with the artificial retina.

3.1.2 Image Recognition

We were faced with having to perform image recognition with very limited memory and processing power. Initially, we thought that we might perform recognition by doing some sort of edge detection on the images captured by the camera (incidentally, the artificial retina chip has some built-in edge enhancement techniques which would have made this easier than usual).

However, to perform matching via edge detection requires us to account for scaling, etc., and potentially requires a large amount of storage for the image, assuming that the filter is not causal. Additionally, we did not have enough memory to store the template *and* the captured image, and we did not wish to have to keep imprinting the robot with a new template after each new match attempt.

To solve this problem, we decided to use a simple image detection technique, a sum of absolute differences. Specifically, we performed image subtraction of the stored 8K template and the image as follows:

$$SAD = \sum_{i=0}^{127} \sum_{j=0,\text{even}j}^{127} (|I(i,j) - T(i,j)| >> 5)$$

We ignored the low-order bits of the differences in order to prevent problems with overflow. Finally, we found that, because the signal from the camera is an analog signal, we considered any signal differences less than 64 (i.e., in the lower 5 bits) to be contributed by analog noise.

We acknowledge that this is overly conservative and perhaps ignores some signal differences that can actually be attributed to the capture of a different signal. However, we have found that, by using a distinctive mother object (e.g., a distinctive black and white-colored penguin) the absence or presence of such a mother object creates distortions in the captured image of more than 64 reconstruction levels per pixel. Similarly, most, if not all, signal noise can be accounted for by ignoring the lower 5 bits.

3.2 Memory

Our robot required some sort of memory in order to store a template of an image that could be used for comparison with its mother. Because the storage on the Handy Board itself is limited to 32 KB, including program memory, we felt it was necessary to store our template off-board in a separate memory. Anatomically, one might almost consider this as a sort of long-term memory for a creature – once imprinted with an appropriate template (prior to "birth"), the template is persistent and independent of what the robot sees at any instance in time.

The easiest way to implement off-board memory for the creature was to use an EEPROM that interfaces with a serial-peripheral interface (SPI). An SPI port is a way of interfacing to memory so that values can be stored serially, thus limiting the width of the bus. The Handy Board also has an SPI interface, which makes storage to SPI EEPROM relatively easy. However, the largest SPI-interface EEPROM that was available as of April 2001 was $8K \times 8$, meaning that we could only store 8 KB of data into the EEPROM at any given time.

Given that the size of the pictures that we were trying to capture were 16 KB, some downsampling was necessary in order to store some representation of an image as a template. We found that downsampling the image by a factor of 2 horizontally and storing this in memory resulted in a relatively accurate representation of the image. Figures 3-4 and 3-5 show a 16 KB image and the corresponding downsampled 8 KB image, respectively. Given the low resolution of the picture to begin with, we find that actually even the human visual system can perform the appropriate low-pass

Figure 3-4: 128×128 image taken with the Gameboy camera. The picture shows an image that we took in our lab; clearly visible is the power supply in the rear of the picture, the soldering iron on top of it, and some pliers in the foreground.

Figure 3-5: 64×128 image taken with the Gameboy camera and stored in SPI EEPROM. The displayed picture is 128×128, with even columns repeated. The degradation of the image, while noticeable, is by no means catastrophic.

filtering to mask the data loss that has occurred by horizontal downsampling. As a result, we find that this operation is relatively reasonable for a creature to do. Such a downsampling operation is also perfectly reasonable given what we learned about the human visual system—specifically, humans do not store *entire images*, either, but rather focus on general high-level representations (such as in the experiment where small details in flashing images were not perceived by humans).

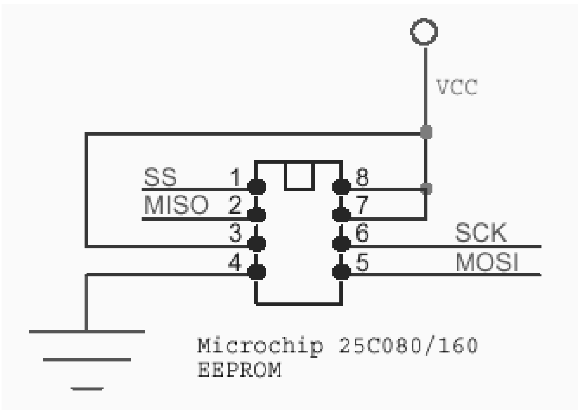

Figure 3-6: Pinout for the Microchip 25160 SPI $8K \times 8$ EEPROM [3].

Distance (clock ticks)	Motor Speed
100	75
200	77
300	80
400	82
500	85
600	87
700	90
800	92
900	95
1000	98

Table 3.1: Mapping for distance values to corresponding pursuit speeds. Our robot behaves in a Braitenberg-style fashion that allows it to follow its target more aggressively when it is farther away from the target and more slowly when the vehicle is closer to the target.

3.3 Environmental Cues

Once it has used vision to recognize its mother, the robot must make use of appropriate environmental cues to maintain a close distance to its mother. Specifically, our robot uses the Cricket location support system to enable it to determine its distance from its mother at any given instance. With this information and a small amount of additional state, our robot can determine the two necessary pieces of information that aid it in keeping a close distance too its mother:

- **Absolute distance sensing:** The robot can determine exactly how far away its mother is, so that when the mother becomes a certain distance away, it begins moving closer to its mother once again.
- **Relative distance sensing:** Even though Cricket gives the robot a non-directional distance reading, the robot can determine whether or not it is moving closer to or farther from its mother based on successive distance measurements. Specifically, if the robot notices that the distance from its mother is becoming progressively farther away, it knows within a reasonable probability that it is headed in the wrong direction (we assume, of course, that the mother will not move away from its children faster than they are capable of moving.

Additionally, we used a *Braitenberg* style model for our sensor/actuator routines, so that the farther the mother was from our robot, the faster the robot moved in an attempt to catch up to the mother. The table of the speeds we used to map measured distance to motor speeds is given in Table 3.1.

To enable these distance measurements, we use the Cricket location support system, which consists of a *beacon* and a *listener*. the Cricket beacon simultaneously emits an RF and ultrasound pulse; by measuring the distance in arrival time of these two signals at the listener, the listener can

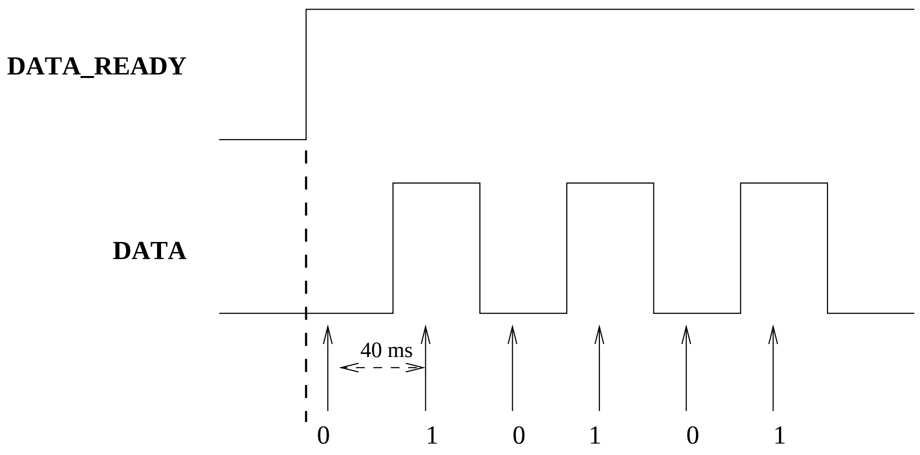

Figure 3-7: Waveform diagram for input data from Cricket listener.

determine to a reasonably accurate degree its distance from the beacon. We found the precision of the system to be within a couple of inches, although accuracy sometimes varied depending on the opportunity for multipath reflection (e.g., if many obstacles were in the way). Cricket uses *beacons* to disseminate information about a geographic space to *listeners*. A beacon is a small device attached to some location within the geographic space it advertises. Cricket beacons are inexpensive and more than one of them can be used in any space for fault-tolerance and better coverage.

To increase the accuracy of the measurements we received from Cricket, we took the median of 5 readings at any point in time. This process was also quite slow, since our Handy Board implementation of an implementation to the Cricket listener could only allow input rates of up to 25 bits per second (meaning that reading one 16-bit value could take almost a second!). Once we received 5 readings, we sorted these readings in a non-optimized way (i.e., using bubble sort) and took the median value as our distance reading. While this method was slow, it managed to give us reasonably reliable readings.

The interface to the Cricket location support system is outlined in Table 3.2. In order to request a reading from the Cricket listener, the digital input DATA_REQ was pulsed. The control mechanism then polled the DATA_READY output until this pin went high. As soon as this pin goes high, the Cricket listener begins outputting the calculated distance value as a 16-bit serial output on the DATA line at 25 Hz. Reading of the actual distance value thus consisted of carefully-timed sampling of this waveform.

Figure 3-7 shows the relation of the DATA_READY and DATA pins, and the arrows show roughly how we sampled the serial data line. Specifically, since the data was being transmitted as a waveform at 25 Hz, we would want to sample roughly at this rate in the middle of each time cycle. Thus, a logical implementation would sleep for 20 ms once the DATA_READY pin went high and subsequently sample each bit, sleeping for 40 ms in between readings. However, we found that the overhead of the Interactive C implementation precluded using such a simple implementation. Because of the

Pin	I/O	Description
DATA_REQ	1-bit Digital Input	Request a reading from the listener.
DATA_READY	1-bit Digital Output	Cricket notification that a distance reading is ready
DATA	16-bit Serial Digital Output	Distance reading (in clock ticks) of the distance between the beacon and the listener.

Table 3.2: Interface to the Cricket Listener module. DATA_REQ is pulsed high, indicating that the Cricket Listener should take a distance reading. When the Handy Board hears DATA_READY go high, it begins reading the serial data sent on the DATA line.

overhead of the machine, we found that it took approximately 2 ms to read each bit in a tight loop; thus, we allocated 2 ms for reading and subsequently slept for 38 ms, thus allowing us to sample roughly every 40 ms (25 Hz sampling rate). Additionally, to allow for drift in sampling rate, we slept only 5 ms initially, rather than 20 ms, to limit the possibility of the Handy Board falling behind and becoming desynchronized.

3.4 Primitive Sensing

To enable our robot to perform primitive operations such as obstacle avoidance, we use left and right bump sensors on the front and rear of the robot, for a total of four bump sensors. Any of the bump sensors are hit, the robot reverses its direction, but applies a mild perturbation to the motors to allow the reversal to have a slight turn. We use this method in an attempt to prevent the robot from reaching some "deadlock" state, whereby it becomes trapped and continually backs up and moves forward to hit the same wall.

The bump sensors were designed based on the fact that the robot was generally moving either in the forward or reverse directions, and, as such, only front and rear bump sensors were necessary. We designed the bump sensors based on the fact that we did not particularly care which side of the front or back the obstacle was hitting, but certainly we wanted to detect that obstacles that were close to our robot on either side. As such, we used bump sensors that extended wider than the horizontal width of the robot, and where touching these whiskers *from any angle* would result in the appropriate bump sensor being triggered.

3.5 Logical Processing

In this section, we give an overview of the decision making our robot needs to do given various sensor inputs. We then proceed to discuss how we implemented the sensor processing and related issues that arose due to the complexity of sensors.

3.5.1 Elements

We implemented the logical layer of our robot using the Handy Board, which provides an Interactive C interface to the Motorola HC6811 microcontroller. Specifically, the logic of our program needed to arbitrate among the following inputs:

- **Camera:** The robot must process visual input and determine at any given point in time whether it is looking at its mother.
- **Cricket:** The robot needs to use inputs from Cricket to determine whether it is too far from its mother, and additionally, if it is moving in the completely wrong direction from its mother.
- **Bump Sensors:** The robot must use its bump sensors to determine whether or not it is encountering an obstacle at any given point in time.
- **Shaft Encoders:** Finally, the robot must use shaft encoders and break-beam sensors to determine whether or not it is moving in a straight line at any point in time, and also to control the degrees of its turns (e.g., to make a 180° U-turn).

3.5.2 Control Flow

One of the most challenging aspects of the logical processing of our robot were figuring out how to arbitrate among different sensed signals in a manner that allowed the robot to appear fairly interactive, yet prevent any catastrophes from occurring that might result if the robot failed to sense any inputs quickly enough. Much of the problems which arose during the implementation of our robot can be attributed to the processing of sensory information.

In particular, the speed of many of our sensors is particularly slow and require a fair amount of processing. For example, the input from the Cricket listener sensor took 5+ seconds, mostly due to the fact that the Handy Board required a particularly slow serial data input rate and, due to noise considerations, we sometimes took up to 5 readings per input reading reported to the Handy Board. For this reason, the Handy Board spends a large percentage of its operating time polling for inputs from the Cricket listener. For this reason, we felt that it might be wise to put sensor reading functionality into threads. However, we found that, because sensor/actuator connections were closely tied, having many threads responsible for modifying the speed of the motors results in a lot of chaos without very strict locking discipline. Additionally, we found that the context switching and additional memory overhead of running multiple threads on the Handy Board was particularly high, and that certain threads were rarely scheduled CPU time, resulting in extremely sporadic and infrequent sensor readings. As such, we decided to accept a performance slowdown in exchange for more predictable (not to mention more efficient!) behavior from the Handy Board.

Because we decided to use an event loop architecture, our tradeoff was simply to poll the Cricket listener for readings much less often. We polled the Cricket listener only once every 20 iterations through the loop. Additionally, because of the slow performance of the Cricket readings, we required our robot to *stand still while taking distance readings*. This averts the possibility that our robot would run into a wall while taking distance readings and fail to have the bump sensors detect this collision.

However, we called other higher priority functions (such as obstacle avoidance and orientation control mechanisms) every time through the loop. In this fashion, we were able to predictably control the priority of arbitration between the multiple sensor inputs.

Figure 3-8 shows how the robot behaves throughout the course of its life. Initially, the robot is in the imprinting stage. Once a template is captured, it remains in the stage where it is looking for its mother (i.e., a matching template). Once a matching template has been found, it identifies whatever Cricket beacon it hears as its mother and attempts to start following that beacon. If the robot thinks that the beacon is within a fixed distance of the robot, it remains stationary. Otherwise, the robot begins moving forward, keeping track of the following inputs:

- **Distance to mother:** If the distance is increasing, the robot assumes that it is headed in the wrong direction and turns 180 degrees (a "U-turn"). Additionally, if the robot "catches up" to the mother, it stops moving, and the robot periodically adjusts its speed depending on how far from the mother it thinks it is.

- **Obstacles:** If the robot is hitting an obstacle, it reverses its current direction by reversing the polarity of the motors, and randomly perturbing the voltages so that the reversal is not directly backwards but allows a slight turn.

Given the physical embodiment of our robot, a U-turn could be performed in two distinct directions (e.g., a left or right U-turn), and each of these U-turns can be performed in two ways (e.g., by rotating one wheel forward or the other wheel in reverse). We implemented our U-turn to default to a left-forward U-turn. If, however, the robot encounters an obstacle on its left, it reverses the direction of the U-turn by driving forward with the opposing wheel, turning 180 degrees, *plus* the number of degrees that the robot had already turned in the counter-clockwise direction.

3.6 Robot Embodiment

This section describes the embodiment of our robot, specifically, the placement of physical components, and the design of construction of the physical body, motors, etc. of our robot.

3.6.1 Physical Embodiment

To build the physical embodiment of our robot, we used LEGO bricks. This was because the design of our robot was effectively *ad hoc*. Being software engineers, we had no idea regarding what would be the best placement for various sensors, power supplies, etc., and we found that the optimal placement of each component became obvious as we were building (i.e., placement of sensors in optimal places for readings, and placement of other components to balance the weight of the robot and to minimize wire length).

Using LEGOs allowed us to prototype and design as we went along, without having to tediously rebuild portions of our robot as we found that initial portions of our design were faulty.

3.6.2 Locomotion

To enable motion of our robot, we used two Polaroid motors, which run at 19,000 RPM with no load at 6.9 V. We used a $5 \times 5 \times 5 \times 3 : 1$ (375:1) reduction ratio for these motors, which gave us enough torque to drive each of the wheels and additionally allowed us enough granularity in speeds.

Because each motor was controlled independently, the robot was not guaranteed to move in a straight line simply by setting the power of each motor to an equal value. We found that when each motor was set to an equal power, the robot veered slightly to the right. To ensure that, over a short range, our robot maintained a relatively straight path, we set the differentials on the motors to slightly favor the right motor (i.e., we drove the right motor with slightly more power) to counteract this tendency.

To counteract long-term deviations from a straight-line course and to control error propagation we used shaft encoders and break-beam sensors to allow the robot to keep track of how many rotations both the left and right wheels had made, and occasionally correct its orientation to keep it moving along a relatively straight path.

3.7 Power Considerations

Powering both of the motors, the Handy Board, the Gameboy camera, and the Cricket listener resulted in the Handy Board getting reset when we tried to start up the motors. This is due to the fact that when the motors start, they draw a large amount of current, thus drawing power from the Handy Board and causing a reset due to lack of power to the Handy Board. To solve this problem, we powered the motors separately from the Handy Board. Additionally, providing power to the Cricket listener with the same power can result in a noisy transmission from the listener to the Handy Board and thus a corrupted signal. However, we only witnessed this effect sporadically and thus decided that, as long as the motors were driven from a separate power supply, the robot could function acceptably.

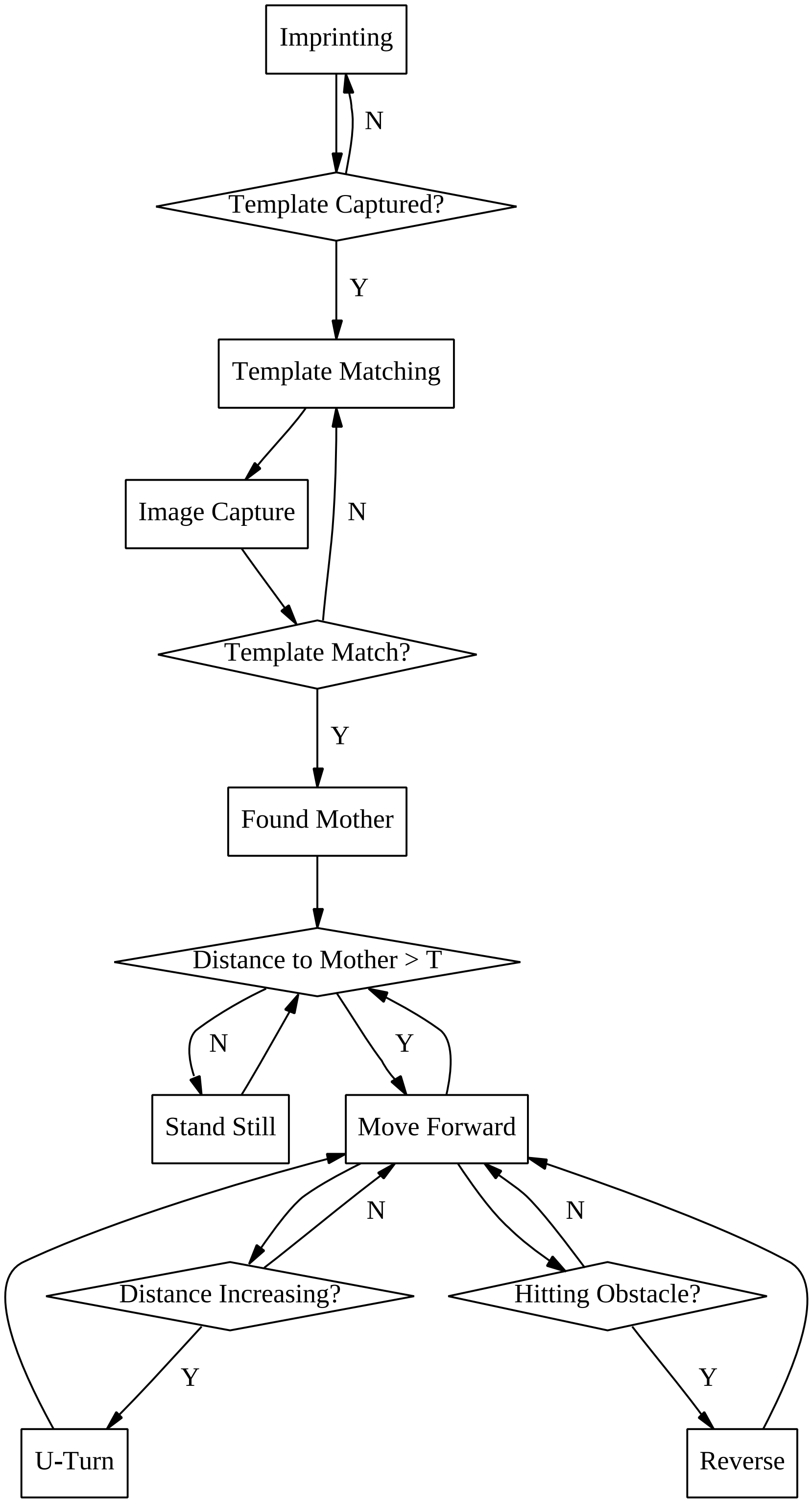

Figure 3-8: Flow diagram for robot behavior.

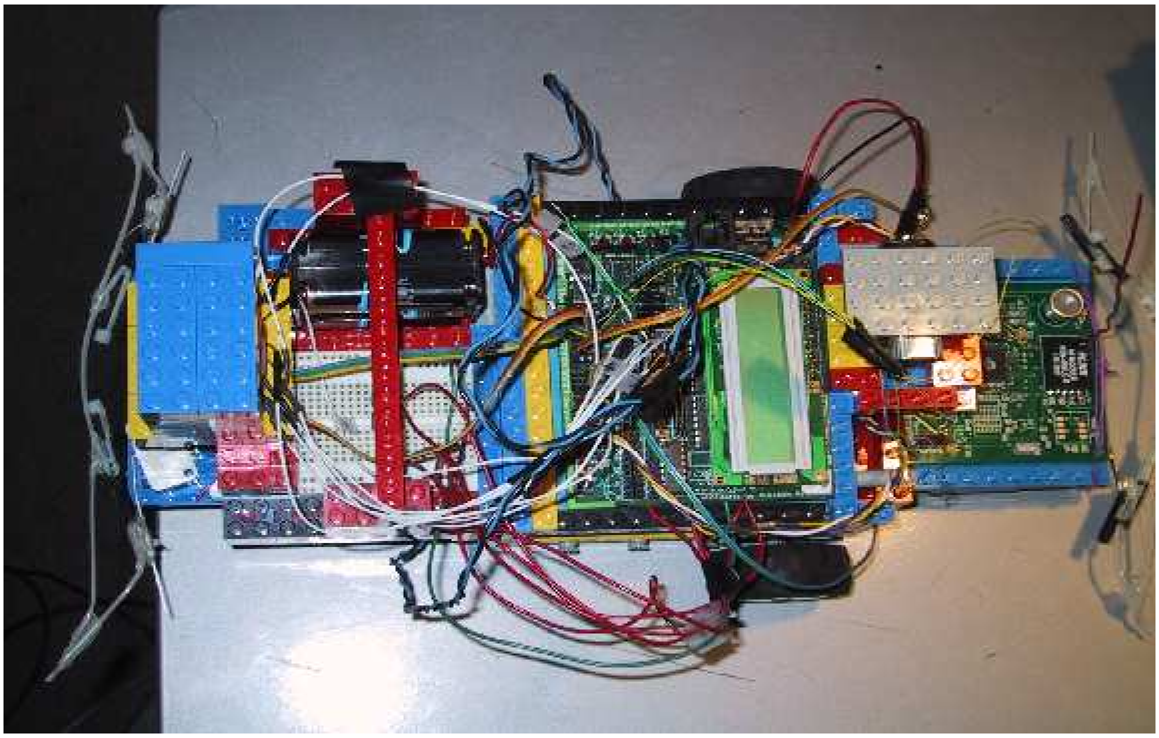

Figure 3-9: Physical embodiment of the robot.

Chapter 4

Results and Lessons

This section describes the measurements we took to determine the accuracy and precision of our sensors and detection algorithms. We additionally present the test scenarios we used to verify the correct behavior of our robot, and include general commentary and advice on robot design.

4.1 Measurements

In this section, we briefly describe measurements taken to test the image recognition capabilities of our robot, as well as measurements we took to evaluate the effectiveness of the Cricket location support system as a distance sensor.

4.1.1 Imprinting Test

The setup for our imprinting test is given in Figure 4-1. We first took a picture of the penguin against the wall to imprint our robot with the template. We took five successive pictures of the penguin against the wall and measured the pixel differences that each picture had against the template. Note that this will be nonzero because the pixel values reported from the camera are actually analog and are subject to corruption by noise. We also took several pictures of a completely different object (i.e., a stuffed alligator). The results of our tests in Table 4.1 show that our pixel differencing technique can be effective in recognizing a template object. The numbers reported are the sum of absolute pixel differences, divided by 32 to avoid overflow.

4.1.2 Cricket Test

To test the precision and accuracy of the Cricket location support system as a distance sensor. We took roughly 40 distance samples for each scenario and report the mean and variance for each experiment.

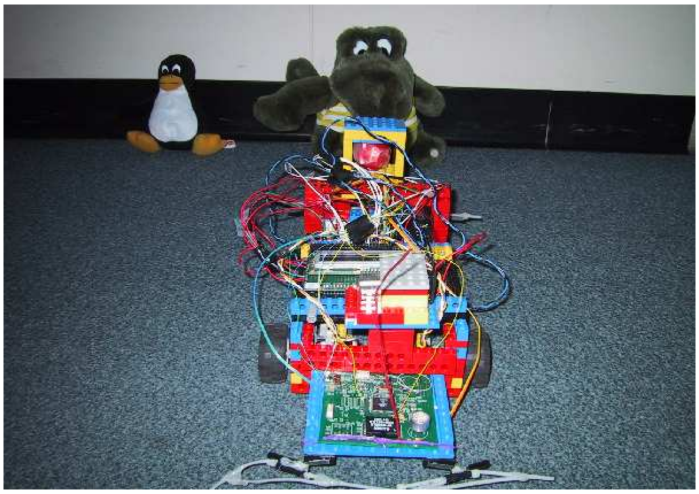

Figure 4-1: Picture of the setup we used to test the effectiveness of our imprinting algorithm. We first imprinted our robot on the penguin. We took several images of the penguin and calculated pixel differences, then took several images of the alligator and calculated pixel differences again.

Case	**Average**	**Standard deviation**
Penguin	92	16.5
Alligator	423	50.3

Table 4.1: Comparing SAD values for penguin and alligator

Our three experiments are as follows:

1. Beacon approximately 70 distance units away from robot, robot facing away from beacon (Cricket listener facing beacon).

2. Same experiment as 1., with the robot now facing the other direction (Cricket listener facing away from beacon).

3. Same experiment as 2., with human obstruction in between beacon and listener.

Our experiments show that placing the Cricket listener on the tail of our robot slightly increased the variance of measurements, which suggests that a better design would have been to place the Cricket listener on the *front* of the robot, where readings would have been more accurate. Furthermore, we find that introducing obstructions between the robot and its mother can result in particularly sporadic readings and behavior. We did the equivalent of median filtering on the readings to obtain more reliable values. Nevertheless, it is clear that our robot performs less reliably in a dynamic environment. Table 4.2 shows these results.

Case	**Average**	**Standard deviation**
1	70.89	1.04
2	74.31	1.75
3	84.83	4.77

Table 4.2: The average and standard deviation of 40 distance samples for each case

4.2 Test Scenarios

To test the functionality of our robot, we did both testing of individual modules and behavior testing for our robot. In this section, we describe the tests that we performed to ensure the working operation of our robot. For the behavior tests, we have posted the corresponding movie files to `http://nms.lcs.mit.edu/~feamster/6.836/`.

4.2.1 Camera Test

In order to test our camera, we used the serial I/O library available from the contributions page on the official Handy Board Web page [1] to output pixel values from the captured image to the serial port. We used a program that is freely available on the Web called *Acarv* to read in 8-bit pixel values from the serial port and print them to the display using horizontal scan lines [2]. Note that the source code for this program is in Delphi and is not publicly available.

We slightly modified the behavior of our program to call `serial_putchar()` to output the captured pixel values to the serial port rather than store them in the eeprom. This allowed us to view the images with *Acarv* and save the captured images as bitmap files. The trickiest part of this was setting the baud rate appropriately in Windows and matching this baud rate accordingly with what *Acarv* expects. We found that the program expects serial data at 9600 baud. Figure 4-2 shows a screenshot of the image being captured from the Gameboy camera and being sent to *Acarv*.

4.2.2 Cricket Test

In order to test that the Cricket listener was functioning properly, we programmed the Cricket listener to output a 16-bit distance value of `0x5555` regardless of the true distance, because this value corresponds to a binary value of `0b0101010101010101`, and, if we could correctly read this value, we could be guaranteed that our sampling rate of the digital waveform was correct.

Once we were guaranteed that we could get accurate distance readings from the Cricket listener, we reprogrammed the listener so that it returned the actual distance values.

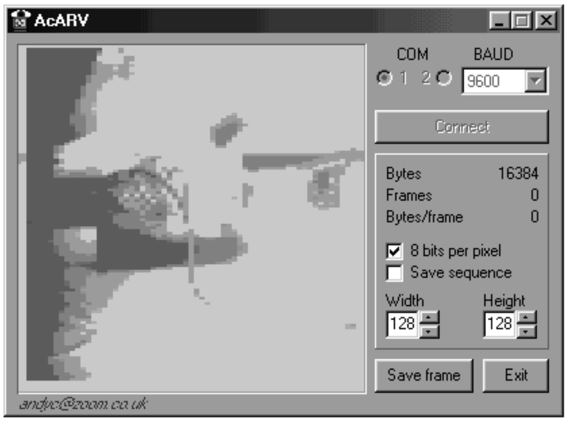

Figure 4-2: Screenshot of the *Acarv* program we used to test the Gameboy camera functionality.

4.2.3 Locomotion Tests

We subjected our robot to two locomotion tests: a test to see if it could move in roughly a straight line, and a test to ensure proper operation of the U-turn on command.

These tests were fairly simple. The straight line test consisted simply of giving our robot a command to begin moving forward, and observation to verify that the robot moved roughly in a straight line and additionally that distance correction (via equalization of shaft encoder counts) occurred on a regular basis to ensure that the robot was still moving in a straight direction.

For the U-turn test, we simply wrote a function called `U-turn()`, which turned the robot a fixed number of shaft encoder counts on one wheel. We verified that this changed the orientation of the robot roughly 180°. Additionally, we tested the combination of U-turn and obstacle avoidance, by making sure that, if the robot began to make a U-turn in one direction and then was forced to abort the U-turn in that direction due to an obstacle, that it would reverse its direction and continue turning until its orientation was 180° from its *original* orientation.

4.2.4 Behavioral Tests

Because our robot had several basic functionalities, we tested our robot with several independent tests. We used independent and combined tests to verify the correct behavior for obstacle avoidance, following, and U-turn functionalities.

To test the following behavior, we started our robot within the threshold distance of the beacon to ensure that it remained stationary. Then, we slowly moved the beacon farther away from the robot and verified that the distance readings our robot was receiving were increasing (this was in fact the case). We continued to move the beacon farther from the robot, until the distance between the beacon and the robot was greater than a certain threshold. We verified that, at this point, the robot began to move forward. Once the robot started moving forward, we verified that it continued to move towards the beacon (assuming a relatively stationary beacon), and that the robot would stop moving once the distance between it and the beacon was less than the specified threshold distance.

We tested simple obstacle avoidance by setting the robot on a course for its mother (by moving the beacon far away from the robot) and then directed it towards a wall to cause the robot to move forward and suddenly hit a wall while in pursuit of its mother. We then verified that our robot would reverse its direction and turn slightly, so that it could stand a chance of turning to avoid the wall the next time it moved forward.

Finally, we tested the U-turn functionality by setting the robot in motion (again, by placing the beacon farther away than the threshold distance from the robot), but originally orienting it so that it was moving away from the beacon. Our test results show that, when Cricket gave us reliable distance readings, our robot was able to recognize that it was moving farther away from the beacon and perform a U-turn.

Chapter 5

Conclusion

5.1 Emergent Behaviors

From our implementation, we showed that we can build a robot that develops *emergent behaviors* from simple sensor/actuator relationships. Without the use of any global control or explicit communication between the robot and the surrounding objects in its environment (e.g., its mother), we have created a robot that appears to recognize an object as its mother and follow it closely as a young animal might do to its mother.

Additionally, we have incorporated internal states so that the robot can keep a short history of its relative distance to its mother and bahave in a certain fashion according to how the distance changes over a short time period. The robot appears to have some knowledge about the direction in which it is travelling, apparently noticing that it is travelling in the wrong direction and reversing its direction if it thinks it is travelling away from its mother.

However, this behavior is merely a product of its environment. Specifically, this only work reliably if the robot is headed in *exactly* the wrong direction. Otherwise, the robot will turn 180 degrees and head in a different direction. But one could imagine the pathological case where the robot begins travelling at a 90 degree angle with respect to the mother's motion. In this case, the robot will never converge to find its mother, but merely always travel back and forth along a perpendicular with respect to the mother's line of motion. However, in most cases, the simple behavior that we have given to the robot makes it appear as though the robot actually "knows" where its mother is.

5.2 Image Processing

Our attempts to perform image recognition given limited memory and processing were successful enough for the very constrained environment in which we placed our robot. However, given the limited computational and memory resources, it is very difficult to account for various anomalies that make image processing difficult, including scaling, analog noise, and lighting changes.

Given our ability to only store half of one picture, we used an ad hoc technique for image recognition that works under very constrained conditions. While the fact that animals only use partial templates to perform matches of current conditions against a template is likely true (as shown in lecture), it is more likely that animals have much more sophisticated image recognition than we implemented in our robot.

With ample resources (or simulation), it might be interesting to evolve "baby robots" with different image recognition techniques in order to see what sort of techniques are most effective given computational constraints.

One particular enhancement that could be made to our robot is to increase the speed at which the robot captured images. The capture rate was extremely slow, in particular because our code was not optimized for speed and the overhead of the Handy Board was particularly slow. We recommend that the clocking signals for the artificial retina chip be made by an independent circuit.

5.3 Environmental Cues

We successfully used the Cricket location support system to provide our robot environmental cues sufficient to follow a moving target. By attaching a Cricket beacon to the moving target (i.e., the *mother*), our robot could use a Cricket listener to determine absolute distance to the beacon.

With additional minimal storage, our robot could determine relative distance changes over time to determine whether or not it was moving closer to or farther from the target.

One aspect of our robot was that it was not capable of determining directionality. Specifically, the robot could not determine whether or not the target was in front of it, behind it, or to one side of it. Thus, we relied extensively on the fact that the mother would move relatively slowly and in a predictable fashion in order to allow the robot to follow it.

Several changes could be made to our integration of the Cricket location support system to improve its performance. First, the placement of the Cricket listener should be placed in a slighly more prominent position on the robot. Presently, the listener is positioned on the "tail" of the robot and is relatively close to the ground. This means that, as the robot attempts to follow its mother, the Cricket listener is placed in a position on the robot that tends to obstruct readings if the beacon is not placed high enough off the ground.

Second, the Cricket listener seems to draw a lot of power from the battery. It might be advisable to power the Cricket listener from a separate power supply from the Handy Board power supply to extend the life of the batteries, and to reduce the amount of noise experienced by the Cricket listener. The tradeoff here is, of course, that the robot becomes heavier with the addition of more batteries.

5.4 Acknowledgements

We thank Hari Balakrishnan and NTT for funding this project, which made it financially more feasible to build a physical robot. Additionally, we thank Bodhi Priyantha for his assistance with Cricket, and his willingness to tailor the listener protocol to our design specifications, and Allen Miu for providing us with robot building supplies and advice on building robots from lessons he had learned in the past.

Appendix A

Code

A.1 Vision Code

```
int START=0;
int SIN=1;
int LOAD=2;
int RESET=3;
int Xck=4;
int READ=9; /* digital input */
int Vout=4; /* analog input */

int gb_reg[8];
int i, j;
int val, mask;

void clock() {
set_digital_out(Xck);
/*   sleep((float)0.2); */
clear_digital_out(Xck);
}
/*
   This code is based on Gameboy Camera Interface IC Ver 3.2 developed by
   Drew Weng.  http://www.geocities.com/drew01833/robots.html has his
   original code.
*/


void main()
{
  int diff=500, last_diff=0;

  printf( "Gameboy Camera   Press Start\n");
  start_press();


  gb_reg[0] = 0x80;
  gb_reg[1] = 0x04;
```

```
  gb_reg[2] = 0x00;
  gb_reg[3] = 0x10;
  gb_reg[4] = 0x01;
  gb_reg[5] = 0x00;
  gb_reg[6] = 0x01;
  gb_reg[7] = 0x04;

  printf( "Initializing...\n" );
  reset_camera();
  printf("After reset_camera...\n");
  set_registers();
  init_spi();
  while( 1 )
  {
    printf("Press [START]   to send image\n" );
    start_press();
    serial_putchar(255);
    printf( "Sending Image..." );
    get_image();
    beep();
    printf("\nDone!\n");
    print_image();
    while (1 && diff > 500) {
      diff=get_diff();
      printf("difference is %d\n", diff);
      beep();
      start_press();
    }
    begin_follow();
  }

}

void reset_camera()
{
  clear_digital_out(START);
  clear_digital_out(SIN);
  clear_digital_out(LOAD);
  clear_digital_out(Xck);

  clear_digital_out( RESET );
  clock();
  set_digital_out( RESET );
}

void set_registers()
{
  for( i = 0; i < 8; i++ )
  {
    val = i << 8;       /* Put the register address in the hi byte */
    val += gb_reg[i];               /* Put the register value in the low byte */
    for( j = 10; j >= 0 ; j-- )
    {
      clear_digital_out( Xck );
```

```
      mask = 1 << j; /* mask the next bit */
      if( val & mask )
        set_digital_out( SIN );
      else
        clear_digital_out( SIN );

      set_digital_out( Xck );
    }
    set_digital_out( LOAD ); /* assert load */
    clear_digital_out( Xck );
    clear_digital_out( LOAD );
  }
}

void get_image()
{

  int k;

  set_digital_out( START ); /* Start of capture */
  clock();
  clear_digital_out( START );

  while( !digital( READ ) )
  {
    clock();
  }

  i = 0;
  while( i < 16384 ) /* get next pixel value and send it out serial port */
  {
    if (!(i%2))
      eeprom_write(i/2, analog(Vout));
    clock();
    i++;
  }
}

int get_diff() {

  int diff=0;

  set_digital_out( START ); /* Start of capture */
  clock();
  clear_digital_out( START );

  while( !digital( READ ) )
  {
    clock();
  }

  i = 0;
  while( i < 16384 )
  {
```

```
    if (!(i%2)) {
      diff += (abs(analog(Vout)-eeprom_read(i/2)))>>5;
    }
    clock();
    i++;
  }
  return diff;
}



void print_image() {
  int i,j;

  for (i=0; i<8192; i++) {
    j=eeprom_read(i);
    printf("Pixel %d: %d\n", i, j);
    serial_putchar((j<<8)|j);
    serial_putchar((j<<8)|j);
    sleep((float)0.005);
  }
}
```

A.2 Environment Sensing Code

```
int DATA_REQ = 7;          /* digital out */
int DATA_READY = 12;       /* digital in */
int DATA = 15;             /* digital in */

int motors_on=0;
int speeds[10] = { 75, 77, 80, 82, 85, 87, 90, 92, 95, 98 };
int MAX_DISTANCE=1000;
int DISTANCE_THRESHOLD=80;

int get_distance() {

  int i;
  int j;
  int data[16];
  int distance;
  clear_digital_out(DATA_REQ);

  distance=0;

  set_digital_out(DATA_REQ);
  while (digital(DATA_READY));
  sleep((float)0.005);

  for (i=0;i<16;i++) {
    data[i] = !digital(DATA);
```

```
    sleep((float)0.038);
  }
  clear_digital_out(DATA_REQ);

  for (i=0,j=1;i<16;i++,j*=2) {
    distance+=(data[i]*j);
  }
  return ((distance>>1)&0x7fff);
}


void distance_test() {
  int distance;

  while(1) {
    distance=get_distance();
    printf("\n");
    printf("distance = %d\n", (distance>>1)&0x7fff);
    sleep((float)2);
  }
}

void begin_follow() {
  int distances[5];
  int median_distances[2];

  int i, j, k=0, l=0, temp, distance, speed, last_speed=0;
  int pid;

  for (i=0;i<2;i++)
    median_distances[i] = MAX_DISTANCE;

  for (i=0;i<5;i++) {
      distances[i]=get_distance();
  }

  reset_encoders();
  while(1) {
    if (!(l%20)) {

      set_motors(0,0);
      distances[k%5]=get_distance();
      distances[(k+1)%5]=get_distance();
      k=(k+2)%5;

      hog_processor();

      for (i=0; i<5; i++)
for (j=i; j<5; j++){
  if (distances[i]<distances[j]) {
    temp = distances[j];
    distances[j]=distances[i];
    distances[i]=temp;
  }
```

```
}
      distance=distances[2];

      printf("distance=%d, d1=%d\n", distance,  median_distances[1]);

      if (distance > median_distances[1]+10 &&
  median_distances[1] > median_distances[0]+10 && motors_on) {
ao();
tone(880.0, 1.0);
uturn(speed);
      }


      median_distances[0] = median_distances[1];
      median_distances[1] = distance;

    }

    l = (l+1)%20;
    /**********************************************/

    check_bump(speed-5,speed);

    if (distance > DISTANCE_THRESHOLD && !motors_on) {
      speed=50;
      run_motors(speed-5,speed);
      motors_on=1;
    } else if (distance > DISTANCE_THRESHOLD && motors_on) {
      if (distance > MAX_DISTANCE)
speed=100;
      else
speed = speeds[distance/100];

      run_motors(speed-5,speed);
    }
    last_speed=speed;

    if (distance < DISTANCE_THRESHOLD-10) {
      motors_on=0;
      ao();
    }
  }
}
```

A.3 Motor Control Code

```
int RIGHT_FRONT = 13;
int LEFT_FRONT = 14;
int RIGHT_BACK = 11;
int LEFT_BACK = 10;
```

```
int stop_flag=0;

/* 0 is the right motor, 1 is the left motor */
void
set_motors(int l, int r) {
  motor(0, r);
  motor(1, l);
}

/* 2 is the left motor, 6 is the right motor */
void
reset_encoders() {
  encoder2_counts=0;
  encoder6_counts=0;
  encoder2_high_threshold = 125;
  encoder6_high_threshold = 125;

}

void
print_encoders() {
  printf("2: %d, 6: %d\n", encoder2_counts, encoder6_counts);
}

void
check_bump(int left_speed, int right_speed) {

  while(1) {

    if (digital(RIGHT_BACK) || digital(LEFT_BACK)) {
      set_motors(-right_speed, -left_speed);
      sleep((float)5);
      reset_encoders();
      set_motors(0,0);
    }

    if (digital(RIGHT_FRONT) || digital(LEFT_FRONT)) {
      set_motors(-right_speed-random(100-right_speed),
 -left_speed-random(100-left_speed));
      sleep((float)5);
      reset_encoders();
      set_motors(0,0);
    }
  }
}


void
run_motors(int left_speed, int right_speed) {

    set_motors(left_speed,right_speed);

    if (encoder2_counts-2 > encoder6_counts) {
```

```
      while (encoder2_counts > encoder6_counts) {
set_motors(0,right_speed);
sleep((float)0.25);
      }
      set_motors(0,0);
    }
    if (encoder2_counts < encoder6_counts-2) {
      while (encoder2_counts < encoder6_counts) {
set_motors(left_speed,0);
sleep((float)0.25);
      }
      set_motors(0,0);
    }
}

void
uturn(int speed) {

  reset_encoders();
  set_motors(speed,0);
  while(encoder2_counts < 42) {
    sleep((float)0.25);
    if (digital(LEFT_FRONT) || digital(RIGHT_FRONT) ||
digital(LEFT_BACK) || digital(RIGHT_BACK))
      break;
  }

  if (encoder2_counts < 42) {
    set_motors(0,speed);
    while(encoder6_counts < 42+encoder2_counts) {
      sleep((float)0.25);
      if (digital(LEFT_FRONT) || digital(RIGHT_FRONT) ||
  digital(LEFT_BACK) || digital(RIGHT_BACK))
break;
    }
  }

  set_motors(0,0);
  reset_encoders();

}
```

A.4 Memory Code

```
/* C subprograms to access a Microchip 25Cxxx EEPROM via the SPI port.
Interactive C Version 3.1
January 23, 1998
Hook up diagram, 25Cxxx EEPROM to Finger Board SPI port
25Cxxx Finger Board SPI port JP4
Pin 1 Vcc +5V
Pin 2 MISO
Pin 3 Vcc
```

```
Pin 4 Vss ground
Pin 5 MOSI
Pin 6 SCK
Pin 7 Vcc
Pin 8 Vcc
*/

void init_spi()
     /* init_spi will initialize the SPI port for 1MHz SCK rate and set PD5 (SS) as
an output bit.*/
{
  /* set PORTD ($1008) SS* high, SCK-Lo, MOSI-Hi when DDRD bit 5 is set
     (xx10111)*/
  poke(0x1008,0x2f);
  /* set DDRD ($1009) register (xx1110xx) */
  poke(0x1009,0x38);
  /* set SPCR ($1028) register (01010000)
     SPIE = 0 , no interrupt
     SPE = 1 , set SPI on
     DWOM = 0, port D outputs are push-pull
     MSTR = 1 , SPI is configured as master
     CPOL = 0 , active high clocks selected
     CPHA = 0, select phase
     SPR1, SPR1 = 0 , select E/2 clock
  */
  poke(0x1028,0x50);
}
void out_spi(int outchar)
     /* out _spi will write an ASCII character to the SPI port.*/
{
  int test=0;
  poke(0x102A, outchar);
  while (test == 0) /*loop until data is sent*/
    {
      test = peek(0x1029);
      test = test & 0x80;
    }
}


void eeprom_write_enable()
     /*This routine will set the write enable latch of the 25Cxxx eeprom*/
{
  bit_clear(0x1008, 0b00100000); /*set PD5 CS* low*/
  out_spi(0x6); /*send out write enable instruction*/
  bit_set(0x1008, 0b00100000); /*set PD5 CS* hi*/
}
int eeprom_read(int address)
     /*This routine will read a eeprom memory cell at location address
25C80 EEPROM, address is 0-1027
25C160 EEPROM, address is 0-2047
*/
{
  int HiByte, LoByte,mem_data;
```

```
  HiByte=address>>8; /*mask off lower 8 bits*/
  LoByte=address & 0b0000000011111111; /*mask off 8 higher bits*/
  bit_clear(0x1008, 0b00100000); /*set PD5 CS* low*/
  out_spi(0x3); /*send out read instruction*/
  out_spi(HiByte); /*send out first 8 bits of address*/
  out_spi(LoByte); /*send out last 4 bits of address*/
  out_spi(0x0); /*clock out the memory data*/
  mem_data=peek(0x102A); /*read in memory data*/
  bit_set(0x1008, 0b00100000); /*set PD5 CS* hi*/
  return mem_data;
}
void eeprom_write(int address, int mem_data)
     /*This routine will write mem_data to location address
25C80 EEPROM, address is 0-1027
25C160 EEPROM, address is 0-2047
*/
{
  int HiByte, LoByte;
  HiByte=address>>8; /*mask off lower 8 bits*/
  LoByte=address & 0b0000000011111111; /*mask off 8 higher bits*/
  eeprom_write_enable(); /*set eeprom so can write to it*/
  bit_clear(0x1008, 0b00100000); /*set PD5 CS* low*/
  out_spi(0x2); /*send out write instruction*/
  out_spi(HiByte); /*send out first 8 bits of address*/
  out_spi(LoByte); /*send out last 4 bits of address*/
  out_spi(mem_data); /*clock out the memory data*/
  bit_set(0x1008, 0b00100000); /*set PD5 CS* hi*/
}
```

Bibliography

[1] Handy board. `http://www.handyboard.com`.

[2] Andy Clark. Acarv: image capturing program. `http://pages.zoom.co.uk/andyc/camera.htm`.

[3] Jim Fong. Using SPI EEPROM memory; application note: EAS-AN001, January 1998.

[4] Joseph L. Jones, Anita M. Flynn, and Bruce A. Seiger, editors. *Mobile Robots*. A K Peters, 1998.

[5] Mitsubishi. Semiconductor technical data M64283FP version 2.2.3.

[6] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proceedings of the 6th ACM MOBICOM*, August 2000.

[7] Michael J. Zigmond, Floyd E. Bloom, Story C. Landis, James L. Roberts, and Larry R. Squire, editors. *Fundamental Neuroscience*. Acamemic press, 1998.