1 Autonomous Mobile Robots

To begin our study of behavior-based robotics, let's skip ahead to one of the fun parts. We will use BSim to demonstrate a robot executing a behavior-based program. Please visit the Web site www.behaviorbasedprogramming.com. There you will find a description of how to use BSim (a sample screen is shown in **Figure 1.1**). Select the Collection task, start the simulator, and watch as the simulation runs.

Example: Collection Task

How would you describe the actions of the simulated robot? From a top-level perspective, you could say that the robot is searching for a certain type of object (pucks). When the robot finds a puck, the robot then pushes the puck to the vicinity of the light source. The robot drops the puck off near the light source and then goes to look for another puck. If you are being especially precise, you might add to your description that while the robot finds and delivers pucks, the robot simultaneously avoids or escapes from encounters with other objects.

The sort of multifaceted, overall activity that the robot exhibits in this example we shall call a *task*.[1] A task is the

[1]A task is to a robot as an application is to a computer.

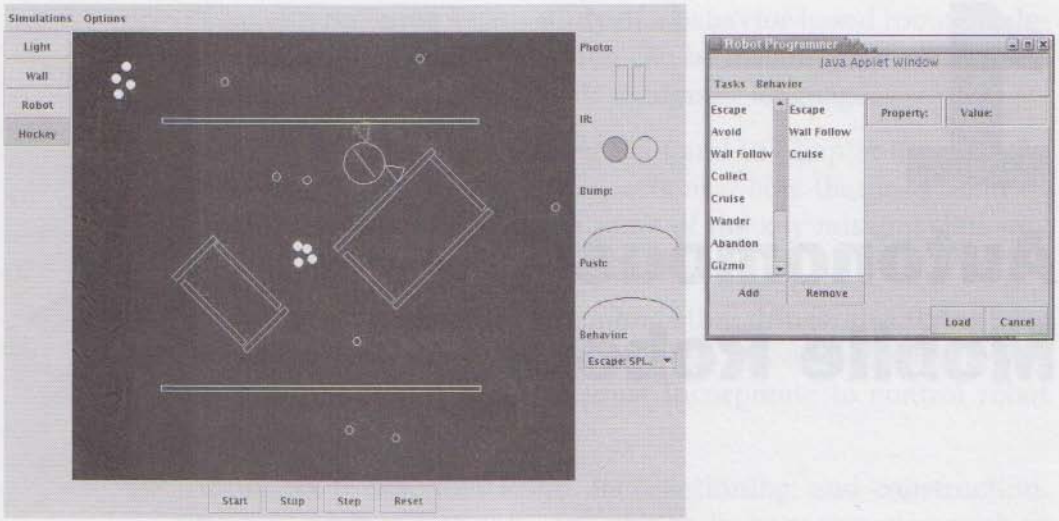

Figure 1.1

The BSim robot simulator enables users to build behavior-based programs for a virtual robot and view their operation in a simulate world. Here the robot executes a sample task. The simulated world is visible in the dark central area. Narrow rectangles represent walls, small open circles are pushable pucks, and closed circles represent light sources. The large circle is the robot. Wedge-shaped areas emanating from the robot show the field of view of short-range obstacle sensors. The status of the robot's sensors and operating program line the right side of the window. The smaller window on the right enables the user to program the robot.

kind of job you might ask another person to do. Tasks often have rather simple descriptions. If you wanted someone to perform the robot's job in this example you might say, "Please move all the pucks next to the light." But, as we shall later see, encoded in that deceptively simple phrase is a world of intricacy and subtlety.

Our ultimate purpose is to learn to design and program autonomous mobile robots. Starting with the description of a useful task that we wish the robot to perform, we must specify the physical features and write the appropriate software that will enable a robot to accomplish that task. One of the most powerful weapons we have for attacking complex problems of this sort is reductionism. Reductionism means that we try to reduce or decompose a large, difficult problem into a set of smaller, simpler problems that we can more easily understand. But what principles should we use to chop a big robot problem into small-

er, more manageable ones? For autonomous mobile robots, behavior-based approaches have proven effective.

Taking a behavior-based approach means that we try to devise a set of simple behaviors (algorithms connecting sensing to actuation) that, *when acting together*, produce the overall activity we desire. The Collection task is implemented in exactly this way. The number and nature of the simple (we will call them *primitive*) behaviors that implement the Collection task probably are not obvious to you. However, before you read further, please observe the simulation for a time while considering this issue. You will likely begin to form opinions about how the Collection task is accomplished; see if you can ferret out the elements, the primitive behaviors, that make it work.

The major components of the simulated robot are shown in **Figure 1.2**. The bumper can determine if a collision has occurred and can sense whether the contacted item is sliding (like a puck) or is fixed (like a wall). BSim is implemented in such a way that the light sensors can see over the pucks, but walls block the light. The walls and pucks reflect infrared radiation (IR), so that

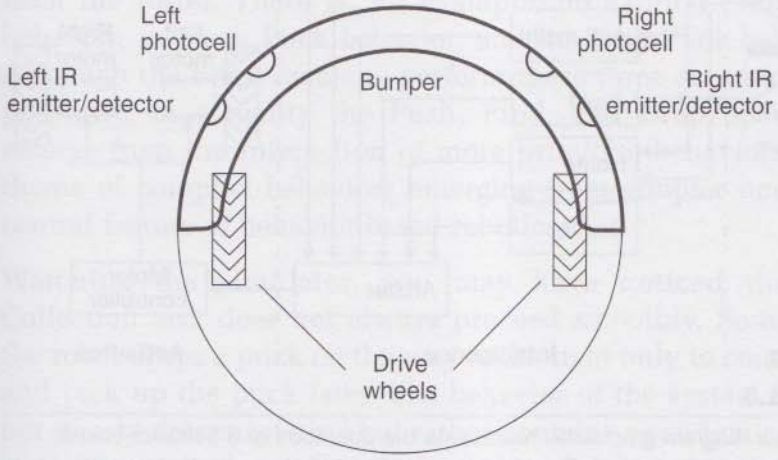

Figure 1.2

The BSim robot includes a simulated bumper that detects collisions with objects, IR proximity detectors that attempt to detect objects before a collision occurs, and dual photocells that can determine the direction and intensity of a light source. Robot motion is enabled using two independently controllable drive wheels visible through the robot shell.

the robot can detect them with its IR sensors. The light sources hang from the ceiling so that the robot can pass under them.

Figure 1.3 shows the *behavior diagram* of the program that implements the Collection task. A behavior diagram is a graphical tool that helps us understand what the robot program does and how it works. Most details in the figure are unimportant at this stage, but do take note of the major divisions. The robot can be thought of as having three parts devoted to sensing, actuation, and intelligence. The intelligence section takes input from the robot's sensors and sends its output to the robot's actuators. Internally, the intelligence section is composed of several primitive behaviors and an arbiter. It is the nature of these primitive behaviors that you have been trying to identify.

The Collection task is composed of only six primitive behaviors. They are called Escape, Dark-push, Anti-moth, Avoid, Home,

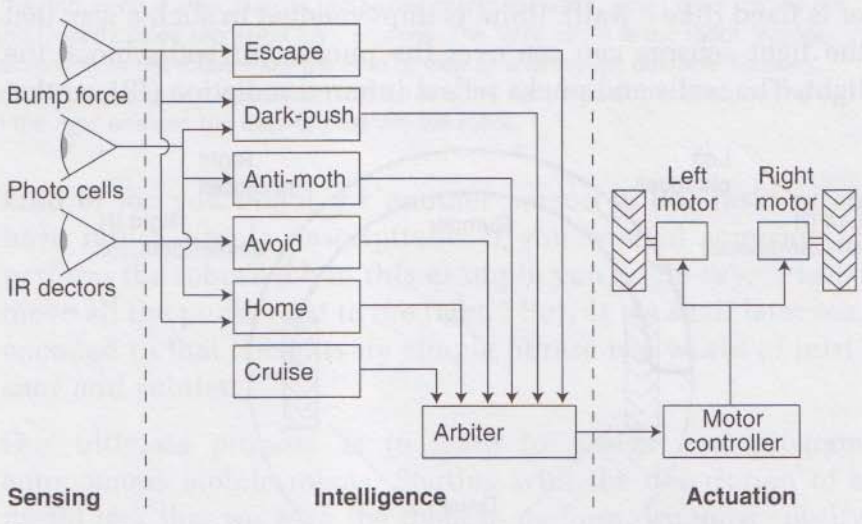

Figure 1.3

A behavior diagram graphically represents the operation of a behavior-based robot. At the highest level, the robot is composed of elements that provide sensing, actuation, and intelligence functions. Information about the robot's environment flows in through the robot's sensors to several primitive behaviors in the intelligence section. Behaviors named Escape, Dark-push, Anti-moth, Avoid, Home, and Cruise implement the Collection task by computing motion commands for the robot. A structure called an arbiter combines or selects the commands and sends a final command on to the robot's motors.

and Cruise. If the robot collides with an object that the robot cannot push, the Escape behavior directs the robot to back up and spin, thereby choosing a new direction. Dark-push tries to avoid pushing pucks when the light is not visible to the robot; this is the case when the robot faces away from the light. Refusing to push pucks in the dark usually prevents the robot from pushing pucks away from the light. The Anti-moth behavior causes the robot to turn away when it gets too close to the light. This lets the robot drop off a puck in the vicinity of the light. The Avoid behavior monitors the robot's IR proximity detectors. Normally an avoid behavior would cause the robot to turn away from obstacles, but the Collection task configures Avoid in the opposite way. When the robot senses an object, the robot turns *toward* the object. This helps the robot find pucks. If the object happens not to be pushable, the Escape behavior takes over to drive the robot away from the obstacle. The Home behavior allows the robot to orient itself toward the light source and move in the direction of the light. The Cruise behavior has the robot drive straight when the robot otherwise doesn't know what to do.

Perhaps some behaviors that you expected to find are missing from the roster. There is, for example, no explicit Find Puck behavior, no Push Puck behavior, and no Drop Puck behavior. Although the robot seems to perform these three activities purposefully, in actuality the Push, Find, and Drop operations *emerge* from the interaction of more primitive behaviors. This theme of complex behaviors emerging from simpler ones is a central feature of behavior-based robotics.

Watching the simulator, you may have noticed that the Collection task does not always proceed smoothly. Sometimes the robot drops a puck on the way to the light only to come back and pick up the puck later. The behavior of the system is thus not purely deterministic,[2] but rather contains a random component. You never know exactly how the robot is going to get the pucks to the light. But at the same time, the robot's overall

[2]In a deterministic program, events follow each other in a completely predictable way. Given a particular initial configuration and a particular sequence of sensor readings, the robot always responds in exactly the same way.

behavior is very robust—no matter the bumps and missteps along the way, ultimately the robot collects the pucks.

These are basic characteristics of behavior-based systems. But before we plunge into a sea of details, let's step back to view the big picture and acquire a better grounding in the fundamentals.

Robot Defined

There are many definitions of the word robot, but I think of a robot[3] simply as: *A device that connects sensing to actuation in an intelligent way.* This minimalist characterization takes for granted a couple of assumptions. First, the thing the robot senses is the external world, not just the robot's own internal state. Second, at least some of the actuation is applied to the robot itself in such a way that the robot moves (see **Figure 1.4**). (Without these assumptions, a sedentary lawn watering system, whose only sensor is a timer and whose only actuator is a servo valve on a water line, would qualify as a robot. Most unsatisfying!) Making intelligence part of the definition of robot seems to

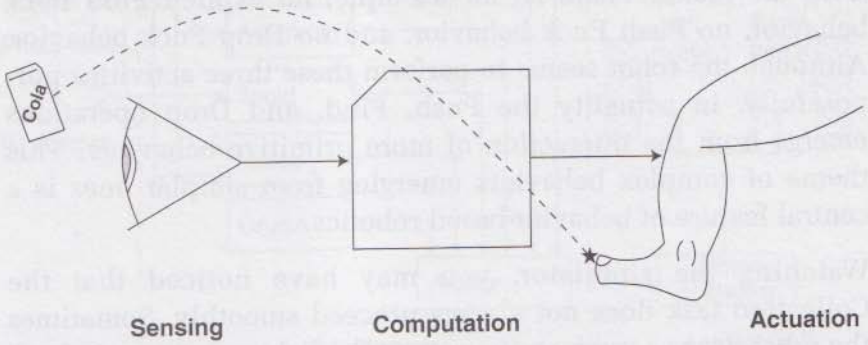

Figure 1.4

In its simplest form, a robot senses the world, performs some computations on what it has sensed, and then acts. The robot's actions can affect both the robot and the world.

[3]Teleoperated mobile devices, platforms controlled exclusively by a remote human operator, are popularly referred to as robots. However, for purposes of this text, a robot must be autonomous to qualify as a robot. Thus, in general, robots mentioned here will be assumed to be autonomous.

add a subjective component. But all that I mean by intelligence in this context is that the robot processes sensor information into actuator outputs with some minimum level of complexity.[4]

A robot is not a computer. Computer scientists agree on the abstract definition of a computer. And although one computer may be faster or slower than another or may have more memory than another, in a fundamental sense, all computers are equivalent.[5] The same high-level program that solves a problem on a multimillion-dollar supercomputer can, in principle, solve the problem on an 8-bit microprocessor—eventually.

The equivalence of all computers makes it far easier for computer scientists to analyze computer programs than it is for them to analyze robot programs. Using the notion of equivalence and the definition of computability, computer scientists can prove theorems about computation. Often scientists can decide in advance if a computer is able to solve a particular sort of problem and, if so, how long that solution might take.

The practical effect of the equivalence of computers is that when you sit down to write a program in a high-level language, you need give almost no thought to which computer will run your program. Is the hard drive interfaced to the computer via a FireWire or a USB connection? You don't care. Will the computer be powered by 120-volt AC and run in the United States or will the computer be plugged into 240-volt supply in a European country? It makes no difference. Will the computer use an optical mouse or a mechanical mouse? Such issues simply have no relevance as you compose your program.

---

[4]People disagree about what properly constitutes a robot; my definition does not resolve those disputes.

[5]Remarkably, the equivalence of all computers was proven in 1936—a time when the first electronic computer was still a decade away. Alan Turing, a British mathematician, is responsible for the proof; he invented a concept called the universal Turing machine. All computers are Turing machines and because of this, all computers share certain fundamental properties. See *Computer Power and Human Reason* by Joseph Weizenbaum; MIT Press, 1976 for a readable explanation of Turing machines. Or review Turing's original paper, "On Computable Numbers, with an Application to the Entscheidungsproblem," Alan Turing, *Proc. London Math. Soc.* Ser. 2–42 (Nov. 17, 1936), pp. 230–265.

But the logic that proves the equivalence of computers does not hold for robots. In general, robots are decidedly *not* equivalent to each other. Rarely can one devise a theorem proving that a robot will be able to perform a particular task. Different robots have different sets of sensors and different sets of actuators. Robots come in different shapes and have sensors positioned in different places. In contrast to computers, such differences are not academic—when it comes to robots, these differences are critical. Two robots constructed or equipped differently but designed to accomplish the same task may require not just different code, but different approaches to the problem. Even robots that are identical except for where sensors are positioned on the robot may require radically different programs. With robots, we don't get to ignore the details.

Still, we can make some statements that hold true for all robots. Here are a few: Robots have sensors that measure aspects of the external world. Robots have actuators that can act on the robot and on the world. The output of a robot's sensors always includes noise and other errors. The commands given to a mobile robot's actuators are never executed faithfully. Not all these statements are happy ones, but buck up! It's the challenge that makes robotics interesting.

Let's consider a robot's components in a bit more detail.

Sensing

To act effectively in an unknown and changeable world, a robot must collect information about the world. Sensors provide that information. A sensor is a transducer that converts a physical quantity into an electrical signal. Once this signal is digitized, the robot's computing elements can use the signal to reason about the external world.

I find it helpful to think of sensor-provided signals as answers to questions that the robot asks about the external world. Perhaps hundreds of times per second a robot asks questions like: "Is the left bump switch triggered?" "Does the right-pointing IR sensor

detect a reflection?" "What is the numeric difference of the signals from the left and right photocells?" Together, answers to questions such as these tell the robot its current situation; knowing the situation, the robot's program can then choose an appropriate response. When sensors answer the robot's questions correctly and unambiguously, the robot accomplishes its task. But if the robot can't get good answers or, worse, if it asks poor questions, success becomes elusive.

Take another look at the Collection task running on BSim. BSim's sensor window shows the instantaneous values of the robot's sensors. Observe in particular the display of the photocells while the robot is homing on the light source. (See the online help information if you have trouble identifying the correct display.) The left and right photocells point diagonally forward. The left photocell points more directly toward the light than does the right photocell when the robot points to the right of the light. In this configuration, because of the more direct pointing, the left photocell detects more light than the right. When the robot points to the left of the light source, the right photocell sees more light than the left. The robot's program uses this difference to move the robot in such a way as to equalize the light intensity measured by each photocell and thus home on the collection point. And this is where the conceptual trouble with sensors begins.

In our example, the question we want to answer is, "Which way is the collection point?" But "collection point" is a human concept. The robot knows nothing about such things. All the robot has to work with are numbers—the digitized electrical signals produced by sensors. In our example, we have arranged things so that when the robot points toward what we think of as the collection point, the photocell light level difference is zero. Thus the only question the robot needs to ask (or is able to ask) is, "What is the difference between the left and right photocell readings?" The relationship between the human concepts with which we reason and the numbers the robot computes is in the programmer's mind, not the robot's mind.

Our first duty as robot programmers is to make sure this relationship, between the questions we would like the robot to ask and the ones it is able to ask, is as sound as possible. We must always consider such things as: Under what circumstances does the difference between the photocell readings *not* indicate the direction to the collection point? Could there be another light source that will fool the robot? Do the two photocells report the same numeric value when exposed to the same intensity of light? And so on. Being always mindful of the shortcomings of our sensors and our concepts will help us program more effectively.

Actuation

Having decided what to do, the robot makes it so by sending commands to the actuators. Actuators are transducers that perform an operation inverse to that of sensors—an actuator converts an electrical signal into a physical quantity. Actuators can include devices like speakers that convert electricity to sound. Sometimes, even LEDs that convert electricity to light are thought of as actuators. But in the context of mobile robots, an actuator usually consists of one or more electric motors connected via a gear train to a wheel, leg, arm, or gripper.

The virtual robot in BSim has two actuators, the left and right wheel motors. Actuators are less philosophically challenging than sensors, as they raise fewer issues of concept or perception. Send an electric current to an actuator and the actuator does something. Still, like sensors, actuators have their own ways of confounding us when we try to use actuators to maneuver a robot. In particular, real-world actuators don't always do as they are told. Send, say, 500 milliamperes of current to both drive motors for one second and, in one situation, the robot will move one foot forward. But send exactly the same current for the same time to the motors when the robot is in some other circumstance (for example, straddling the divide between a tile floor and a shag carpet) and the robot will translate half a foot forward while spinning 30 degrees to the left.

Responding to the Challenge

Mobile robots don't know what they will find in their environment (the environment is unstructured); the position of nearby objects can change without warning (the environment is dynamic); and mobile robots usually don't know where they are (position uncertainty grows as the robot moves). Despite these fundamental issues, until the mid-1980s most researchers tried to force mobile robots into the same mold as manipulator robots. That is, researchers tried to account for every object, build a world model, plan a series of motions, and then execute the plan. But that approach didn't work very well. The amount of computation involved forced mobile robots to move and

[8]See *Mind Children* by Hans Moravec. Harvard University Press, 1998.

respond very slowly, and their slow response made them vulnerable to unanticipated changes in their environment.[9] Often, by the time a planning robot got around to executing its plan, the dynamic environment had changed to the point that the plan was rendered obsolete.

A new approach was needed.[10] Rather than relying on an omniscient programmer to tell the robot where everything is, let the robot find out for itself. That is, give the robot sensors that can detect objects in the robot's environment. Because objects can move, don't just sense the objects once and remember where they were, but instead sense the objects continuously and react immediately when motion occurs. If you want to make progress in robotics even though robots don't usually know where they are, then tackle problems that don't require knowledge of absolute position.[11] And most important, build a robot control system designed to handle unstructured, dynamic environments. This is exactly what behavior-based robotics attempts to do.

Robot's World View

To program a robot effectively, we must see the world as the robot sees it. This requires that we largely abandon the familiar

[9]Researchers weren't being dense in the days before behavior-based robotics when, as my Handey-project group did, they insisted on using world (and robot) models; they were trying to be rigorous. It *is* possible to analyze robots in a general way, analogous to the way that computers are analyzed *if* we use the formalism of a world model. Unfortunately, the baggage that world models force robots to carry is so cumbersome that in real-world environments, robots collapse under the strain.

[10]In some ways the "new approach" is a case of back to the future. In the late 1940s and early 1950s, a Kansas City-born, British neurological researcher, W. Grey Walter, built some autonomous robots that rivaled the best of the early artificial intelligence robots. Walter's robots, Elmer and Elsie, could wander about without getting stuck, could find their way into their hutches by following a light, and could recharge themselves. Each robot had a brain consisting of two vacuum tubes. Some consider Walter's creations to be the world's first behavior-based robots. You'll find more details on the Web and in Walter's fascinating book *The Living Brain*, W. W. Norton & Company, Inc., 1953.

[11]It should be noted that behavior-based robots have no trouble incorporating absolute position information and using it effectively. Rather, it is the case that such information is rarely available in real-world situations and when it is, the cost is typically high and/or the resolution is low.

views and concepts we use to understand events and to communicate with others. Looking around a room, any person can easily identify chairs, tables, and other people. We know where the walls are, we recognize places we have been before, and we can reason from a rich set of concepts. People have mechanisms, evolved over millions of years, that enable us to interpret the chaotic sensory information we take in. People have hardwired facilities for noticing subtle motions, recognizing faces, and processing sounds. We understand the person speaking to us even when we are surrounded by a dozen other conversations, all taking place at the same time.

Robots start from zero. A robot with a sophisticated sonar sensor does not sense, say, a chair leg, a wastebasket, or an umbrella stand. As indicated in **Figure 1.9**, no matter how complex and interesting its environment, the robot's view of the world col-

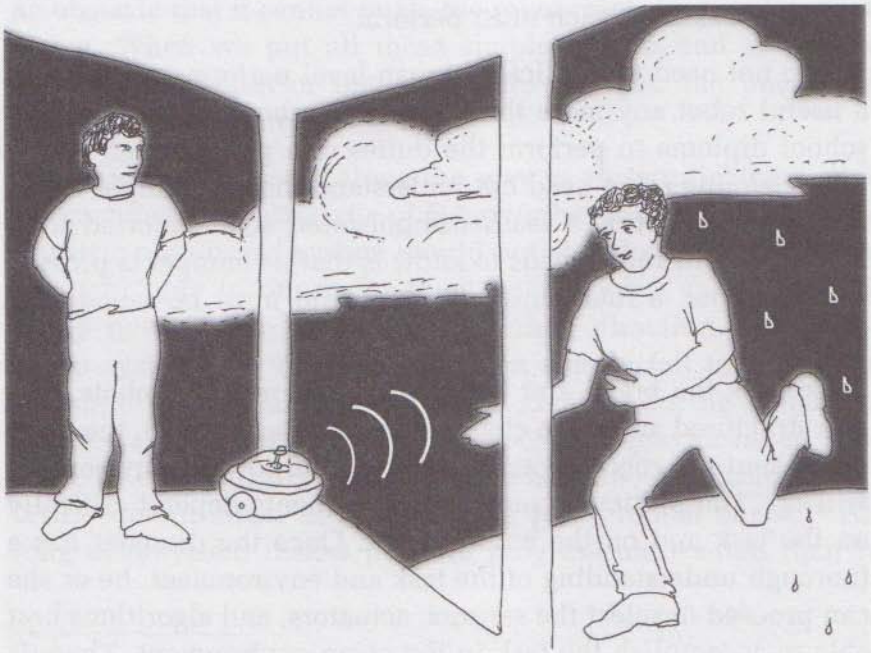

Figure 1.9

Humans experiencing the world map sensory information into a rich set of high-level concepts such as friend, rain, and running. Oblivious to all this, a robot viewing the same scene with a sonar sensor experiences the world as a single number—the time of flight of a sonar pulse. *(Drawing courtesy of Sara Farragher.)*

lapses into a single number—the time between the transmission of a sonar ping and reception of an echo. A robot equipped with a pyroelectric sensor does not recognize that a human has crossed its path; rather, the robot notices only that the voltage output by the sensor has just changed polarity.

Programming a robot means accepting this impoverished view of the world and taking effective action—it *can* be done but we have to change our perspective. People have the complex world-view that we do because this view contributes to our survival. Insects have a much simpler view and yet, given their environment and goals, manage quite handily. Insects, for example, don't need to identify chairs or tables or talk about furniture with other insects; they can, however, discover a food source on a table or in the crevice of a chair and carry nourishment back to the nest. The key to effective performance for a human, insect, or robot is matching sensory, motor, and intellectual abilities with the tasks that each must perform.

We do not need to duplicate human-level performance to build a useful robot any more than, say, a grasshopper needs a high school diploma to perform the duties of a grasshopper. And a floor-cleaning robot need not understand that its bumper is, for example, contacting a leather-upholstered sofa imported from Milan. All this robot needs to know is that its bumper is pressed and therefore, a rotate-in-place command must be sent to the motors.

Sensing is the bread and butter of behavior-based robots. The less structured and more changeable the robot's world, the more dependent the robot is on its sensors. But not just any sensing will do. The particular sensing requirements depend critically on the task and on the environment. Once the designer has a thorough understanding of the task and environment, he or she can proceed to select the sensors, actuators, and algorithms best able to accomplish the task in the given environment. There is no safe extreme—failing to supply the robot with a crucial sensor can lead to failure; incorporating a sensor that does not contribute to the task may lead to economic failure.

What does a behavior-based robot do with the sensory information it collects? The robot connects that information as directly as possible to actuation. Thus behavior-based robots are typically highly *reflexive*. As soon as a relevant condition is recognized, the robot takes appropriate action. Unlike the manipulator robot described in the Introduction, a behavior-based robot does not first collect all information (relevant or not) about its environment, plan what to do, and then execute the plan. Instead, as soon as a behavior-based robot has relevant information, it acts on that data.

All these features are apparent in BSim's Collection task. The robot does not plan—it reacts. But the robot's reflexes, and the way they are combined, are carefully engineered to elicit the overall behavior we want. When the robot is not pushing a puck, the robot wanders about. When the robot bumps into a puck, the robot responds by homing on the light. If the robot bumps into an obstacle that it cannot push, the robot reacts by avoiding that object. When we put all these simple actions and responses together, the behavior that we desire emerges: the pucks are moved to the vicinity of the light.

An occasional misstep along the way to accomplishing a goal is a common characteristic of behavior-based systems. A deterministic plan-based system would not abandon pucks far from the light only to return for them later. Such a system would likely never move pucks in the wrong direction. But plan-based systems are brittle[12] and often simply fail to work—an assumption is violated or the world changes during execution and the system stops, unable to proceed. Behavior-based systems, by contrast, strike a reasonable bargain—they trade away brittle determinism acquiring in its place robust chaos.[13] As long as a system makes positive progress more often than it

[12]When computer scientists talk about brittle performance, they mean that small errors on the input side of a program cause large changes or catastrophic failure on the output side. By contrast, a robust program is one that suffers only a small performance decline when small input errors occur.

[13]The slogan of the MIT AI Lab's Mobile Robot group was, "Fast, cheap, and out of control!"

makes negative progress, the system will ultimately accomplish its goal.[14]

Summary

In this chapter we have taken a quick look at the raw material we will use for robot programming—sensors, actuators, and reflexive behaviors. We've thought abstractly about how the environment affects sensors, how sensed information is processed, how actuators deal with that output, and how the environment can affect actuation. The lessons we've learned include:

- An autonomous robot is a device that connects sensing to actuation in an intelligent way.
- Robots accomplish tasks.
- To perform its task, a robot must ask certain questions about its environment and situation.
- Sensors provide answers to the questions a robot must ask.
- Unambiguous answers are necessary for effective performance.
- Sensors are rarely able to answer the exact question we would like the robot to ask. We must tailor our methods to the questions sensors are able to answer.
- The details of the sensors used and their placement on the robot critically affects the way the robot operates.
- Complex global behavior can result from simple behaviors acting together.

[14]And how does the robot know when the goal is accomplished? In our example, it doesn't. Unless we can provide the robot with a no-pucks-remain-to-be-found sensor, the Collection task will run indefinitely, always on the lookout for a wayward puck. Relying on only local information, behavior-based methods exhibit robust performance. But inferring global truths from only local information is challenging. This fact is sometimes seen as a deficiency of behavior-based methods. Consequently, melding the advantages of behavior-based programming with more deterministic sorts of robot control is an active area of research.

3 Behaviors

In the previous chapter, we learned about basic control systems. A control system is a key element in any robot program, but a simple control system turns the robot into a sort of one-trick pony—making the robot act in the same way all the time. To perform useful work in the real world, we must have our robots do different things under different circumstances. And here enters the concept of behaviors.

Triggers and Control Systems

Recall the example from the first chapter where we used the simulated robot to carry out the Collection task. One of the operations that the robot performed was homing on the light source. To home on the light source the robot ran a light-seeking control system. But the robot did not spend all of its time homing on the light source. Rather, the robot performs the homing operation only when it is pushing a puck. The presence of the puck *triggers* the homing behavior.

Primitive behaviors, as we use the term in behavior-based robotics, have two parts:

1. A control component that transforms sensory information into actuator commands.
2. A trigger component that determines when it is appropriate for the control component to act.

Figure 3.1 diagrams a generic behavior.

In our Collection example, the absence of a puck means that the robot should not home on the light—when the robot is not pushing a puck, it should go out and find one to push. Without a puck, the primitive homing behavior is untriggered. But when a puck is in contact with the robot's bumper, the homing behavior is triggered and the robot tries to move toward the light source. In this case, the trigger part of the behavior looks at the bumper sensor to decide if homing is appropriate, and the control system part of the behavior looks at the photocells to determine how to home.

As far as the system outside the primitive behavior is concerned, it doesn't matter whether the absence of a trigger signal

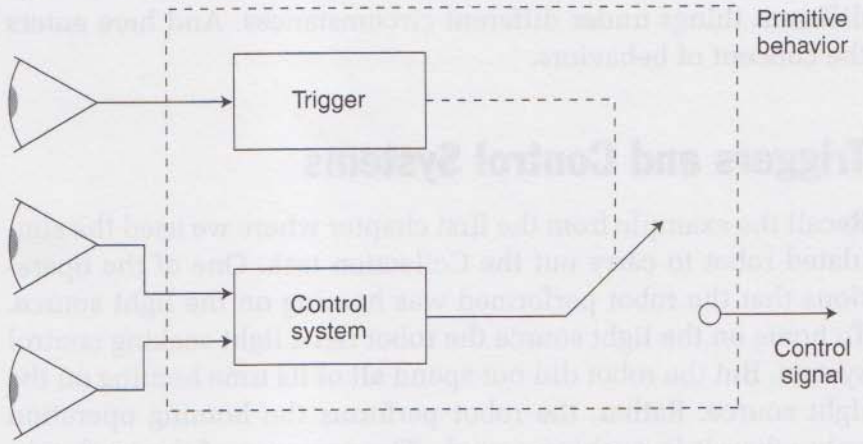

Figure 3.1

A primitive behavior includes a control component (a control system) that maps sensory information to actuation control commands and a trigger to determine the circumstances under which the control system should generate commands for the robot's actuators. Both control system and trigger can examine sensor values to decide what to do. The trigger and the control system may consult the same sensors or different sensors.

stops the control system from computing commands or only prevents the control system from sending commands to the actuators. (The latter case is indicated by the dashed line in the figure). The system as a whole behaves in exactly the same way regardless. Sometimes, allowing the control system to run constantly rather than turning the computation on and off can achieve some utility or program simplification. If the control system runs constantly, the trigger-controlled switch simply discards the output commands when they are not appropriate to the situation.

Servo and Ballistic Behaviors

A primitive behavior is a general construct that places few restrictions on the implementing code. But there are two categories that broadly describe behaviors: servo and ballistic. You have already seen examples of both types in the Collection task.

Typically, a servo behavior employs a feedback control loop as its control component. The light positioning behavior from Chapter 2 is an example of a servo behavior. Suppose that we increased the intensity of the light source when the robot is at rest at the equilibrium point. The robot would immediately respond by backing up. If we decrease the light intensity, the robot will move closer without reluctance. The trigger is always active, so that at every instant, the robot measures the light on the photocells, compares the sum with a stored intensity, and then moves forward or back depending on the difference.

A ballistic behavior, like a shell fired from a cannon, once triggered, follows a predictable trajectory through to completion. The Escape behavior is an example of a ballistic behavior. Escape executes when the robot bumps into a wall or other obstacle. When triggered, Escape does these things: First, the robot backs up a preset distance. Second, the robot spins in place a preset number of degrees. Third, the robot moves forward a preset distance. Then Escape becomes untriggered. (Step Three exists to make sure that the robot is in a different spot when another behavior assumes control. Without the move for-

ward, whatever behavior had run the robot into the wall might just run it into the wall again.)

Usually, we do not think of behaviors as "completing." A behavior just runs continuously, always trying to achieve some goal or maintain some value. But ballistic behaviors provide an exception to this model; as in the case of Escape, they often have a clear end point. You can view a ballistic behavior as a small plan or sequenced operation.

Ballistic behaviors, although sometimes essential, must be used with caution. Depending on how it is written, during the time that a ballistic behavior runs, the robot can be effectively blind. If conditions change during behavior execution or if the robot has made a mistake in initiating the behavior (it may have been started by a noise glitch, for example), trouble can result. Before resorting to a ballistic behavior, it is always best to try first to find a servo behavior that will accomplish the same result. Servo behaviors respond immediately to changing conditions and are less vulnerable to noise and other glitches.