Part I: Programming environments for motion, graphics, and geometry

Part I of this text book will discuss:

- simple programming environments
- program design
- informal versus formal notations
- reducing a solution to primitive operations, and programming as an activity independent of language.

The purpose of an artificial programming environment

A *program* can be *designed* with the barest of tools, paper and pencil, or in the programmer's head. In the realm of such informal environments, a program design may contain vague concepts expressed in an informal notation. Before he or she can execute this program, the programmer needs a *programming environment*, typically a complex system with many distinct components: a computer and its operating system, utilities, and program libraries; text and program editors; various programming languages and their processors. Such real programming environments force programmers to express themselves in formal notations.

Programming is the realization of a solution to a problem, expressed in terms of those operations provided by a given programming environment. Most programmers work in environments that provide very powerful operations and tools.

The more powerful a programming environment, the simpler the programming task, at least to the expert who has achieved mastery of this environment. Even an experienced programmer may need several months to master a new programming environment, and a novice may give up in frustration at the multitude of concepts and details he or she must understand before writing the simplest program.

The simpler a programming environment, the easier it is to write and run small programs, and the more work it is to write substantial, useful programs. In the early days of computing, before the proliferation of programming languages during the 1960s, most programmers worked in environments that were exceedingly simple by modern standards: Acquaintance with an assembler, a loader, and a small program library sufficed. The programs they wrote were small compared to what a professional programmer writes today. The simpler a programming environment is, the better suited it is for learning to program. Alas, today simple environments are hard to find! Even a home computer is equipped with complex software that is not easily ignored or bypassed. For the sake of education it is useful to invent artificial programming environments. Their only purpose is to illustrate some important concepts in the simplest possible setting and to facilitate insight. Part I of this book introduces such a toy

programming environment suitable for programming graphics and motion, and illustrates how it can gradually be enriched to approach a simple but useful graphics environment.

Textbooks on computer graphics. The computer-driven graphics screen is a powerful new medium for communication. Visualization often makes it possible to present the results of a computation in intuitively appealing ways that convey insights not easily gained in any other manner. To exploit this medium, every programmer must master basic visualization techniques. We refer the reader interested in a systematic introduction to computer graphics to such excellent textbooks as [BG 89], [FDFH 90], [NS 79], [Rog 85], [Wat 89], and [Wol 89].

1. Reducing a task to given primitives: programming motion

Learning objectives:

- primitives for specifying motion
- expressing an algorithm in informal notations and in high- and low-level programming languages
- program verification
- program optimization

A robot car, its capabilities, and the task to be performed

Some aspects of programming can be learned without a computer, by inventing an artificial programming environment as a purely mental exercise. The example of a vehicle that moves under program control in a fictitious landscape is a microcosmos of programming lore. In this section we introduce important concepts that will reappear later in more elaborate settings.

The environment. Consider a two-dimensional square grid, a portion of which is enclosed by a wall made up of horizontal and vertical line segments that run halfway between the grid points (Exhibit 1.1). A robot car enclosed within the wall moves along this grid under computer control, one step at a time, from grid point to adjacent grid point. Before and after each step, the robot's state is described by a location (grid point) and a direction (north, east, south, or west).

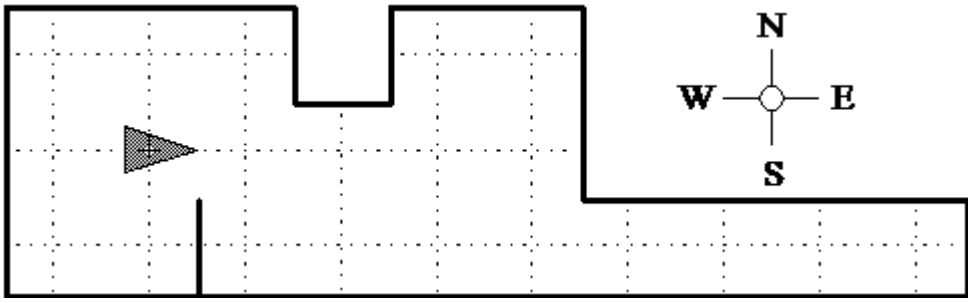

Exhibit 1.1: The robot's crosshairs show its current location on the grid.

The robot is controlled by a program that uses the following commands:

```
   left             Turn 90 degrees counterclockwise.
   right              Turn 90 degrees clockwise.
   forward          Move one step, to the next grid point in front of
you
   goto #           Send program control to the label #.
   if touch goto #  If you are touching a wall to your front, send
                    program control to the label #.
```

A program for the robot is a sequence of commands with distinct labels. The labels serve merely to identify the commands and need not be arranged either consecutively or in increasing order. Execution begins with the first command and proceeds to successive commands in the order in which they appear, except when flow of control is redirected by either of the goto commands.

Example

The following program moves the robot forward until it bumps into a wall:

```
1  if touch goto 4
2  forward
3  goto 1
4  { there is no command here; just a label }
```

In developing programs for the robot, we feel free to use any high-level language we prefer, and embed robot commands in it. Thus we might have expressed our wall-finding program by the simpler statement

```
while not touch do forward;
```

and then translated it into the robot's language.

A program for this robot car to patrol the walls of a city consists of two parts: First, find a wall, the problem we just solved. Second, move along the wall forever while maintaining two conditions:

1. Never lose touch with the wall; at all times, keep within one step of it.
2. Visit every spot along the wall in a monotonic progression.

The mental image of walking around a room with eyes closed, left arm extended, and the left hand touching the wall at all times will prove useful. To mirror this solution we start the robot so that it has a wall on its immediate left rather than in front. As the robot has no sensor on its left side, we will let it turn left at every step to sense the wall with its front bumper, then turn right to resume its position with the wall to its left.

Wall-following algorithm described informally

Idea of solution: Touch the wall with your left hand; move forward, turning left or right as required to keep touching the wall.

Wall-following algorithm described in English: Clockwise, starting at left, look for the first direction not blocked by a wall, and if found, take a step in that direction.

Let us test this algorithm on some critical configurations. The robot inside a unit square turns forever, never finding a direction to take a step (Exhibit 1.2). In Exhibit 1.3 the robot negotiates a left-hand spike. After each step, there is a wall to its left-rear. In Exhibit 1.4 the robot enters a blind alley. At the end of the alley, it turns clockwise twice, then exits by the route it entered.

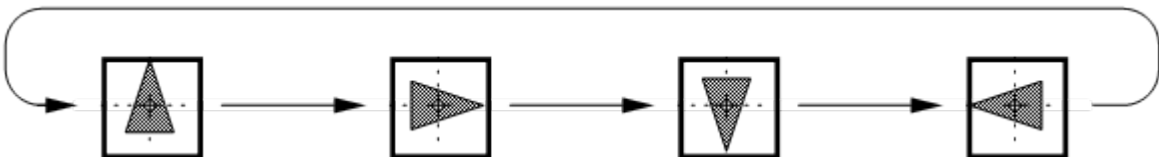

Exhibit 1.2: Robot in a box spins on its heels.

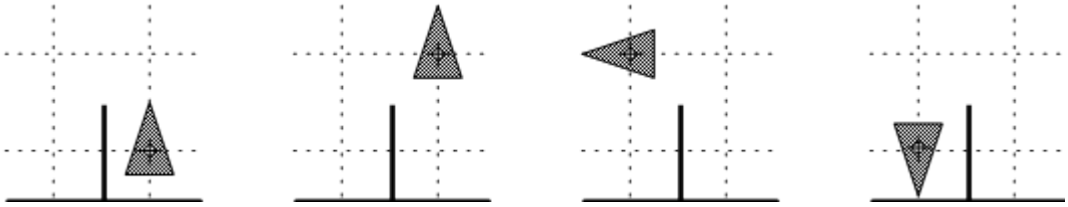

Exhibit 1.3: The robot turns around a spike.

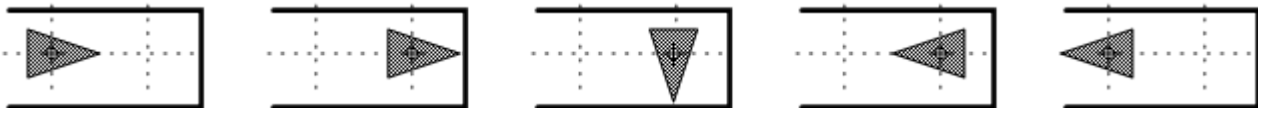

Exhibit 1.4: Backing up in a blind alley.

Algorithm specified in a high-level language

The ideas presented informally in above section are made precise in the following elegant, concise program:

```
{ wall to left-rear }
loop
   { wall to left-rear }
   left;
   { wall to left-front }
   while touch do
      { wall to right-front }
      right;
      { wall to left-front }
   endwhile;
   { wall to left-front }
   forward;
   { wall to left-rear }
forever;
{ wall to left-rear }
```

Program verification. The comments in braces are *program invariants*: Assertions about the state of the robot that are true every time the flow of control reaches the place in the program where they are written. We need three types of invariants to verify the wall-following program: "wall to left-rear", "wall to left-front", and "wall to right-front". The relationships between the robot's position and the presence of a nearby wall that must hold for each assertion to be true are illustrated in Exhibit 1.5. Shaded circles indicate points through which a wall must pass. Each robot command transforms its *precondition* (i.e. the assertion true before the command is executed) into its *postcondition* (i.e. the assertion true after its execution). Thus each of the commands 'left', 'right', and 'forward' is a *predicate transformer*, as suggested in Exhibit 1.6.

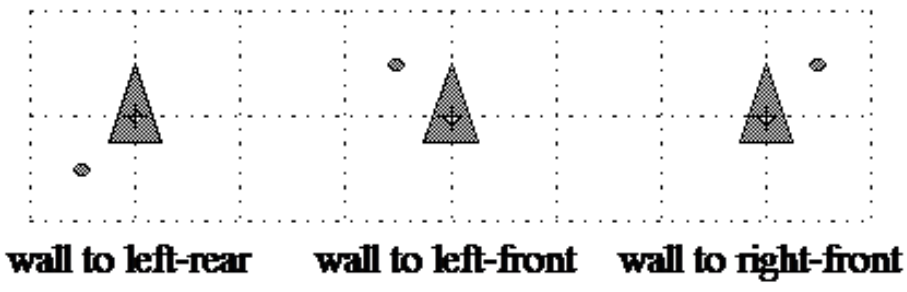

Exhibit 1.5: Three types of invariants relate the positions of robot and wall.

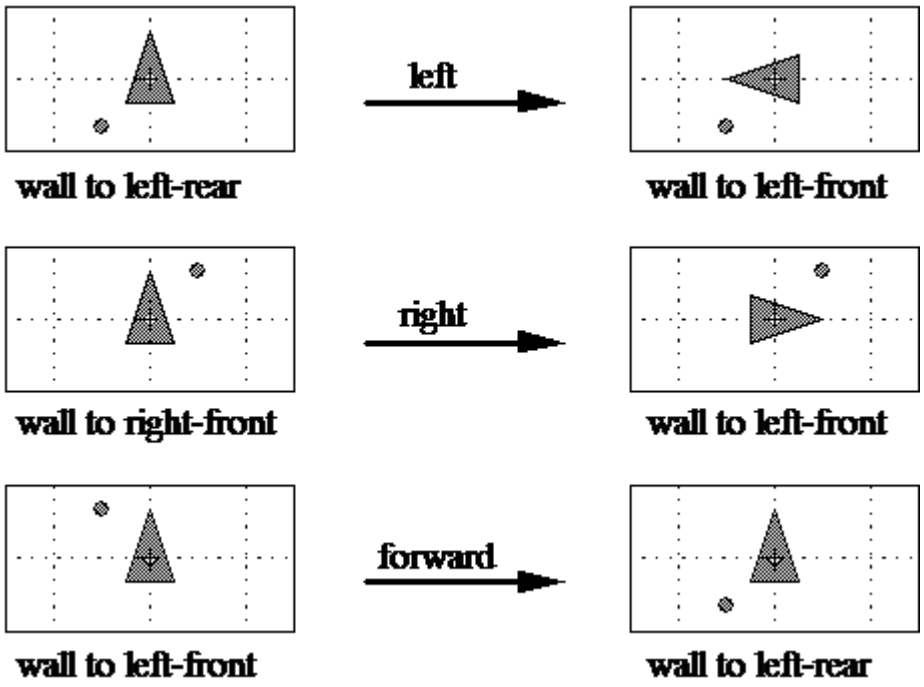

Exhibit 1.6: Robot motions as predicate transformers.

Algorithm programmed in the robot's language

A straightforward translation from the high-level program into the robot's low-level language yields the following seven-line wall-following program:

```
loop
   left;                1  left
   while touch do       2  if touch goto 4
                        3  goto 6
      right;            4  right
   endwhile;            5  goto 2
   forward;             6  forward
forever;                7  goto 1
```

The robot's program optimized

In *designing* a program it is best to follow simple, general ideas, and to decide on details in the most straightforward manner, without regard for the many alternative ways that are always available for handling details. Once a program is proven correct, and runs, then we may try to improve its efficiency, measured by time and memory requirements. This process of *program transformation* can often be done syntactically, that is merely by considering the definition of individual statements, not the algorithm as a whole. As an example, we derive a five-line version of the wall-following program by transforming the seven-line program in two steps.

If we have the complementary primitive 'if not touch goto #', we can simplify the flow of the program at the left as shown on the right side.

```
   { wall to left-rear }          { wall to left-rear }
1  left                        1  left
2  if touch goto 4             2  if not touch goto 6
3  goto 6
   { wall to right-front }        { wall to right-front }
4  right                       4  right
```

```
5 goto 2                    5 goto 2
6 forward                   6 forward
7 goto 1                    7 goto 1
```

An optimization technique called *loop rotation* allows us to shorten this program by yet another instruction. It changes the structure of the program significantly, as we see from the way the labels have been permuted. The assertion "wall to right-front" attached to line 4 serves as an *invariant of the loop* "keep turning right while you can't advance".

```
  { wall to right-front }
4 right
2 if touch goto 4
6 forward
1 left
7 goto 2
```

Programming projects

1. Design a data structure suitable for storing a wall made up of horizontal and vertical line segments in a square grid of bounded size. Write a "wall-editor", i.e. an interactive program that lets the user define and modify an instance of such a wall.
2. Program the wall-following algorithm and animate its execution when tracking a wall entered with the wall-editor. Specifically, show the robot's position and orientation after each change of state.