MC28 Lab 4
Lego Robotic Cars

Max Hailperin, Karl Knight

Due: April 16, 1997

Introduction

In this lab, you will work with a robotic car that can move and turn. You will write a procedure that allows you to command it to move from wherever it is to a specified (x, y) position. In order to do this, you'll have to keep track of its current position and heading. That is, you'll use a computational "object" with state to model the physical state of the car.

The car can hold a marker pen and thereby draw its path on paper. Your procedure for moving to a specified (x, y) position will thus allow you to do coordinate-based drawing. In fact, the final goal of this lab is to have the robot car trace a C-curve on a piece of paper.

You will need to use the NeXTs in this lab, since we have them, and not the SGIs, set up for use with the robot interface.

We have decided that you will do this lab in two-person teams (with one three-person team), partly due to the fact that we have only five robot cars. Each group should write up and hand in a single lab report.

Problem Description

Robotics is a branch of engineering which concerns the control of mechanized devices (robots) by computer programs. These devices typically have motors to provide movement and sensors to allow them to perceive and thereby respond to their physical environment.

You will be working with robot cars. We have a total of five robot cars, and their controlling software is described in greater detail below. Briefly, each car has two independent motors which drive its wheels. Each motor can run forward and reverse at various speeds, these actions being controlled by some built-in Scheme procedures. Some of the five cars also have working bumper sensors, but they won't be used in this lab.

You will control your car primarily through the use of four basic "turtle" procedures described more fully below. Two of these procedures allow you to go forward (resp. backwards) for a specified amount of time, and the other two procedures allow you to turn right (resp. left) a specified angle. (Turning is accomplished by simultaneously driving one side of the car forward and the other backwards.)

Hardware

The motors (and possibly sensors) on the Lego robots are connected by long wires to a black box called the *Hyperbot controller*. The Hyperbot box has a power cord that is plugged into an outlet and also has a communications cable that connects to the computer.

Turtle Graphics

The turtle graphics procedures described below are implemented in the Scheme file `~karl/Public/MC28/turtle.scm` using the Hyperbot primitives listed in the appendix.

`(fd` *time*`)` — procedure
`(bk` *time*`)` — procedure
Go forwards and backwards, respectively, for *time* seconds.

`(rt` *time*`)` — procedure
`(lt` *time*`)` — procedure
Turn right or left, respectively, for *time* seconds.

`(rtd` *angle*`)` — procedure
`(ltd` *angle*`)` — procedure
Turn right or left, respectively, *angle* degrees.

`(rtr` *angle*`)` — procedure
`(ltr` *angle*`)` — procedure
Turn right or left, respectively, *angle* radians.

In lab

1. The first task for you and your partner is to hook up a robot to a NeXT and get it to move forwards, backwards, left, and right. We will show you how to connect the robots to the NeXTs (namely, the communications cable from the Hyperbot controller should be connected to serial port A on the NeXT). After doing this, you should evaluate the file `~max/MC28/turtle.scm` to get the turtle procedures listed above. Now you should be able to get your robot car to move.

   Important: if the car is moving in a dangerous or undesirable fashion, first pick it up; once the wheels are in the air it won't go anywhere. Then you can abort the computation if necessary.

   You are welcome to look at how the turtle procedures are implemented using the Hyperbot primitives, but we suggest that you do this only after having completed the programming required in this lab.

2. The following definitions (and all others in the remainder of this handout) are in the file `~max/MC28/robot.scm`. First of all, we have the `c-curve` procedure from MC27 lab 2, except with `overlay` replaced

by `begin`. This is because we are going to physically draw one half and then the other:

```
(define c-curve
  (lambda (x0 y0 x1 y1 level)
    (if (= level 0)
        (line x0 y0 x1 y1)
        (let ((xmid (/ (+ x0 x1) 2))
              (ymid (/ (+ y0 y1) 2))
              (dx (- x1 x0))
              (dy (- y1 y0)))
          (let ((xa (- xmid (/ dy 2)))
                (ya (+ ymid (/ dx 2))))
            (begin (c-curve x0 y0 xa ya (- level 1))
                   (c-curve xa ya x1 y1 (- level 1))))))))
```

In order that the lines are drawn by the robot, we need to redefine `line`:

```
(define line
  (lambda (x-start y-start x-end y-end)
    (move-to-without-drawing! the-robot x-start y-start)
    (move-to-with-drawing! the-robot x-end y-end)))
```

The two kinds of moving are actually the same except whether the pen is raised while moving. We also avoid moving at all if the robot is already in the right position:

```
(define move-to-without-drawing!
  (lambda (robot x y)
    (if (at? robot x y)
        'no-motion-needed
        (begin (pen-up! robot)
               (move-to! robot x y)
               (pen-down! robot)))))

(define move-to-with-drawing!
  (lambda (robot x y)
    (if (at? robot x y)
        'no-motion-needed
        (move-to! robot x y))))
```

As you can see above, the various robot operations are accessed by procedures that take a robot object as their first argument. The robot object itself is defined as follows:

```
(define the-robot (make-robot))
```

The definitions of the robot operations are show below. The `pen-up!` and `pen-down!` operations signal errors because we haven't built hardware yet for raising or lowering the pen. This will not be a problem, provided you start the c-curve at the robot's current location, since the c-curve is traced out in one continuous (albeit complicated) curve. The other robot operations, `at?` and `move-to!`, are the crux of your assignment. You should also define a third robot operation, `reset!`, which resets the robot's coordinate system so the current position is considered $(0, 0)$ and the current heading is considered to be down the positive x-axis. This is useful between drawings.

Note that we've chosen to represent the robot by a three-element vector storing the current (x, y) position and the current heading. However, we have localized the view of the object as a vector within the constructor and the basic `set-...!` and `get-...` procedures. Thus, the code you write shouldn't need to worry about indexing into a vector.

```
(define make-robot
  (lambda ()
    (let ((robot (make-vector 3)))
      (set-x! robot 0)
      (set-y! robot 0)
      (set-heading! robot 0)
      robot)))

(define set-x!
  (lambda (robot value)
    (vector-set! robot 0 value)))

(define set-y!
  (lambda (robot value)
    (vector-set! robot 1 value)))

(define set-heading!
  (lambda (robot value)
    (vector-set! robot 2 value)))

(define get-x
  (lambda (robot)
    (vector-ref robot 0)))

(define get-y
  (lambda (robot)
    (vector-ref robot 1)))

(define get-heading
  (lambda (robot)
    (vector-ref robot 2)))
```

```
(define at?
  (lambda (robot x y)
    ; You need to write this.
    ))

(define reset!
  (lambda (robot)
    ; You need to write this.
    ))

(define move-to!
  (lambda (robot x y)
    (let ((dx (- x (get-x robot)))
          (dy (- y (get-y robot))))
      ; You need to finish writing this.
      )))

(define pen-up!
  (lambda (robot)
    (error "pen raising hardware not built yet")))

(define pen-down!
  (lambda (robot)
    (error "pen lowering hardware not built yet")))
```

In the `move-to!` operation, you need to both move the physical car (using the turtle procedures) and also update the computational object's state to reflect the new position and heading. When you are figuring out how to do this, don't be afraid to turn and move your own body as well as drawing pictures. A useful scheme primitive is `atan`. The expression `(atan` y x`)` gives the arctangent in radians of y/x. However, unlike the usual arctangent function, the result can be in any of the four quadrants depending on the signs of y and x. Said another way, it returns the angle (in radians) formed by the ray from $(0,0)$ through (x,y) with the positive x-axis.

You should be able to draw c-curves with expressions like

```
(begin (reset! the-robot)
       (c-curve 0 0 3 0 5))
```

Of course, the results are fairly crude because the car doesn't always turn (or move) exactly the desired amount. If the turns are consistently off, you can try adjusting the definition of `degrees-per-second` in the `turtle.scm` file. Better results could be achieved with more precise mechanisms (for example, the rubber-band drive could be replaced). However, a yet better solution would be a *feedback* system that measures how far the car actually turns or moves, rather than just assuming that a certain number of seconds will always produce a certain motion. We don't have the necessary sensors on our cars to do this, however. (During J-term '96, some of the RoboPong teams succesfully implemented such feedback mechanisms, however.)

3. Even within the crude limitations of our hardware, you can produce better or worse results by varying the strategy you use. Suppose the car is headed up the positive y-axis and wants to draw a line in the positive x direction. It could do any of the following things, or many others:

   (a) Turn right $\pi/2$ and go forward.
   (b) Turn left $3\pi/2$ and go forward.
   (c) Turn left $\pi/2$ and go backward.
   (d) Turn right $5\pi/2$ and go forward.

   Every time you change heading you have to decide among options like these; the choices you make will affect how crude the drawing is and also will affect how tangled the car gets in its wires. Assuming you have some time left after getting your initial procedures working, you may want to experiment a bit with alternative strategies, until you find the one you like best. (Be sure to save your first version, in case your later ones don't work or behave worse rather than better.)

Post lab

Write a lab report as usual; it should include your procedure definitions together with any necessary commentary.

Appendix: Hyperbot Primitives

The following procedures have been added to our Scheme and provide a simple programmer's interface to the robotic sensors and effectors, such as in Lego(R) Control kits, using the Hyperbot(TM) controller. You will not need to use these procedures directly; we include their descriptions for those who are curious or want to experiment. (You are welcome to continue playing with the Legos beyond the confines of this lab; just ask one of us.)

`(maybe-gc)` procedure
`(maybe-gc` *free-fraction-threshold*`)` procedure

Does a garbage collection if the fraction of the heap that is free is less than the specified free-fraction-threshold, which defaults to .1. If the threshold is explicitly specified, it must be a positive real number not greater than 1. (1 may be used to force an unconditional garbage collection.) This procedure is useful before turning motors on so that a garbage collection won't occur while the motors are on, perhaps delaying turning them off again.

`(real-time-clock)` procedure

Returns real time in milliseconds since scheme started.

(`wait-microseconds` *delay*) procedure

Sleeps for delay microseconds. The delay must be an exact integer.

(`wait-until` *predicate*) procedure

Repeatedly applies the nilary predicate until it returns true; currently sleeps .01 seconds between each attempt, but that may change.

(`make-hyperbot` *port*) procedure

Creates a Hyperbot object, which must be passed to all the Hyperbot operations; does all the resetting/initialization described for `hyperbot/reset`. Port must be either the symbol `a` or the symbol `b`, indicating which serial port the controller is plugged into. Other argument values are reserved for future use with networking, etc.

(`hyperbot/reset` *hyperbot*) procedure

Resets all state: motors off, maximum power 14, sensitivity 11, fast sensor sampling, all counters zeroed, communications channels cleared. This is useful if you have managed to wedge things. Also, this should always be done after pressing the stop button on the controller box. If you have wedged things so thoroughly that even this procedure doesn't work, it is worth trying to press the stop button and then use this procedure again.

(`hyperbot/close` *hyperbot*) procedure

Severs communications and frees resources. It is illegal to try doing any further operations on this Hyperbot object, but a new make-hyperbot can be done on the same port.

(`hyperbot/make-action` *hyperbot power1 power2 power3 power4*) procedure

Returns a nilary procedure which when invoked sets the output powers as specified. Any powers which are either omitted or specified as `#f` are left unchanged by the action. (Of course it isn't possible to omit a power unless all the subsequent powers are also omitted.) Any power which isn't false must be a real number in the range -1 to 1 inclusive. -1 means maximum reverse power, 0 means off, 1 means maximum forward power, and fractional values can be used to specify reduced power levels.

(`hyperbot/get-sensor` *hyperbot sensor*) procedure

Returns true iff the specified sensor contacts are closed; sensor must be an exact integer in the range 1–4.

(`hyperbot/get-counter` *hyperbot sensor*) procedure

Returns the number of times the specified sensor has cycled from open to closed and back to open since the counter was reset; the sensor number must be an exact integer 1–4.

(`hyperbot/reset-counter` *hyperbot sensor*) procedure

Resets to zero the counter associated with the specified sensor; the sensor number must be an exact integer 1–4.

`(hyperbot/get-period` *hyperbot sensor*`)` procedure

Returns the number of milliseconds the last cycle from open to closed and back to open took on the specified sensor; the sensor number must be an exact integer 1–4.

`(hyperbot/set-sensitivity` *hyperbot sensitivity*`)` procedure

Sets the specified sensitivity, which must be an exact integer in the range 0–15.

`(hyperbot/get-sensitivity` *hyperbot*`)` procedure

Returns current sensitivity, an exact integer in the range 0–15.

`(hyperbot/calibrate-light` *hyperbot*`)` procedure

Assuming an opto-sensor is connected to input number one and it is being exposed to "light" conditions, sets the sensitivity.

`(hyperbot/calibrate-dark` *hyperbot*`)` procedure

As with hyperbot/calibrate-light, but assumes "dark" exposure.

`(hyperbot/calibrate-average` *hyperbot*`)` procedure

Sets sensitivity by averaging light and dark settings.

`(hyperbot/set-sensor-mode` *hyperbot mode*`)` procedure

If mode is true, uses slow sampling; false means fast sampling, the default.

`(hyperbot/set-maximum-power` *hyperbot limit*`)` procedure

Limit must be an exact integer in the range 1–24. The default value, 14, is recommended for Lego 4.5v motors; use of higher values could cause damage.

`(hyperbot/get-maximum-power` *hyperbot*`)` procedure

Returns the current setting.