Extended Exercise: Interactive Games

A Supplement to "How to Design Programs"

1 Interaction via Keystrokes

PREREQUISITE: 7. The Varieties of Data TEACHPACK: draw.ss

Many programs interact with users via keystrokes. Text editing programs interpret every keystroke. Some indicate that the user is typing plain text; others request that the editor changes fonts, transposes words, and so on. Game programs also heavily use arrow keys. They allow players to move objects, fire weapons, open locks, and so on.

The **draw.ss** teachpack supports programs that wish to observe a user's keystroke actions when the teachpack's canvas is the active window. Every keystroke is an event. DrScheme observes these events and provides a function that delivers the keystrokes, if the user has touched the keyboard. Clearly, the first thing we need then is a data definition that describes this class of events:[1]

A *KeyEvent* is either

1. a *Boolean*, i.e., false, if the user didn't press a key;
2. a *Character*, e.g., #\a, #\space, if the user pressed an alphanumeric key,
3. a *Symbol*, e.g., 'up, 'down, 'left, 'right, if the user pressed a special key (or other events happened).

The data definition puts together an interesting mix of distinct classes of data into one class. Naturally, a function that consumes and processes a *KeyEvent* must distinguish all those cases.

[1] Recall that char? is the predicate that recognizes characters and that char=? is a predicate for comparing characters.

The second thing we need is a function that observes the events and delivers them, one at a time:

```
;; get-key-event : → KeyEvent
;; observes the keyboard and produces KeyEvents as they happen
```

When a program evaluates (*get-key-event*) and the user has triggered some event with the keyboard since the last time the expression was evaluated, the expression produces a character or a symbol. If not, it just produces false.

Now consider the following sample function:

```
;; move-how-far? : KeyEvent → Number
;; to interpret a 'left or 'right keystroke as a move
;; into the appropriate direction by 10 pixels
(define (move-how-far? ke)
  (cond
    [(boolean? ke) 0]
    [(char? ke) 0]
    [else ; we now know that (symbol? ke) is true
      (cond
        [(symbol=? 'left ke) -10]
        [(symbol=? 'right ke) +10]
        [else 0])]))
```

It consumes a keystroke and produces the number of pixels that some object has to move to the left (negative) or right (positive). The **cond** expression distinguishes the three subclasses of data in the data definition; the nested **cond** recognizes the kind of key that the user pressed.

Since *KeyEvent*s are reported as ordinary forms of data, we can also make and test examples for this function:

```
(move-how-far? false) "should be" 0

(move-how-far? #\a) "should be" 0

(move-how-far? 'left) "should be" -10

(move-how-far? 'up) "should be" 0
```

Once the function is tested with obvious inputs, we can compose it with *get-key-event* and use it on true input data in a program like this:

```
(start 300 300)
(sleep-for-a-while 3)
(move-how-far? (get-key-event))
```

This expression checks whether any keystroke has happened and produces a number.

```
;; drive-ball : N Posn → Boolean
;; draw and move small red ball according to user input
(define (drive-ball n a-posn)
  (cond
    [(zero? n) true]
    [else (and
           (draw-shape a-posn)
           (sleep-for-a-while .05)
           (clear-shape a-posn)
           (drive-ball (sub1 n)
                       (make-posn (+ (posn-x a-posn)
                                     (move-how-far? (get-key-event)))
                                  (posn-y a-posn))))]))

;; Posn → true
(define (draw-shape a-posn)
  (draw-solid-disk a-posn 3 'red))

;; Posn → true
(define (clear-shape a-posn)
  (clear-solid-disk a-posn 3 'red))

(start 300 300)

(drive-ball 1000 (make-posn 150 150))
```

Figure 1: Moving a ball

When we design programs that react to keystrokes, we usually need to process many keystrokes, not just one. With what we know so far, we can't process more than a fixed number of *KeyEvent*s. To overcome this obstacle, we need to use one function per program whose design is beyond the level of this section. Take a look at figure 1 for one such function. This small program runs for several seconds. During that time, it draws the ball, sleeps for a while to allow users to see the object, clears the object, and moves it

left or right, depending on user input. Then it repeats the process. Drawing and clearing the object from the canvas in rapid succession at different locations gives the impression of a quickly moving object.

Teaching Note: The goal is still to keep the functional portion separated from the event-based portion. To this end, we use a program design discipline that separates the action into one generative recursive function (always provided), which controls the game and intercepts the keystrokes, and ordinary functions in the sense of HtDP, Part I that accomplish the work. This is also a good strategy for a realistic implementation.

1.1 Finger Exercises

Exercise 1.1.1 Copy and paste the above program for moving a red ball on a canvas into DrScheme and use the program to move the ball around. Adjust the argument to *sleep-for-a-while* to your computer. ∎

Exercise 1.1.2 Design the function *up-or-down*, which consumes a *KeyEvent* and produces true when the input is 'up, 'down, #\u, or #\d. ∎

Exercise 1.1.3 Design *move-4-directions*. The function consumes a *KeyEvent* and produces a *Posn*. The latter represents how far the ball on the canvas has to move in one of the four directions ('left, 'right, 'up, 'down). ∎

2 Stopping a UFO

The goal of this extended exercise is to develop a simple interactive game. Imagine the approach of a UFO, falling out of the blue sky. You're riding a modern AUP (anti-UFO platform), and your task is to stop the UFO from crossing the line, i.e., the bottom of the canvas. Your powerful AUP can move left or right, and it can shoot at the UFO in straight lines. The means of last resort is to make sure the UFO crashes into your AUP. If the UFO makes it across the line, you lost; otherwise you win.

Figure 2 illustrates what the scene may look like. The (green) saucer on the canvas is the UFO; the line at the bottom is your AUP. The straight lines going up and through the UFO are the shots that the AUP fired.

The section consists of three subsections. Each corresponds to a stage in the design process. The subsections illustrate what we call the iterative refinement process (see section 16). The goal of iterative refinement is

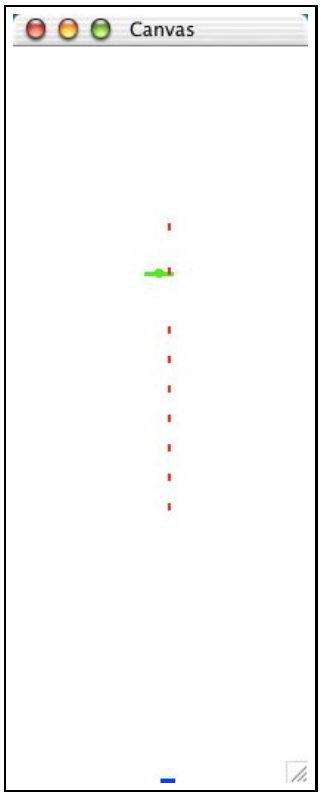

Figure 2: Stopping a UFO

to implement the core functionality of a product and to add pieces of the functionality step by step. Here we present the method via an example; take a look at the sections on iterative refinement in *How to Design Programs* for a thorough description of the idea.

2.1 UFOs

PREREQUISITE: 6.1 Structures TEACHPACK: draw.ss

The first goal is to create a UFO that drops from the top of a canvas to the bottom. The function that does the repetitive work and the fragment that starts the game are defined in figure 3. The following exercises show how to fill in the dots.

```
;; UFO → Boolean
;; fly UFO until it lands on bottom
(define (fly-until-down ufo)
  (cond
    [(at-bottom? ufo) true]
    [else
     (and (draw-ufo ufo)
          (sleep-for-a-while .05)
          (clear-ufo ufo)
          (fly-until-down (move-ufo ufo)))]))

...

;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(start WIDTH HEIGHT)
(fly-until-down (create-ufo (random WIDTH)))
```

Figure 3: Flying a UFO

We adopt the usual conventions from physics and think of the UFO as just a position on the canvas:

A *UFO* is a *Posn*.

When we draw the UFO we think of it as a green disk whose center is the *Posn* with which we represent it. Or, we think of it as something more elaborate, but for now this doesn't matter. We can always change our understanding; we just need to keep in mind what the *Posn* represents. To remind ourselves of this relationship (between data and information), we call the *Posn* that represents a UFO an *anchor point*.

Exercises

Exercise 2.1.1 Design the function *create-ufo*, which consumes a number *n* and produces a UFO whose anchor point is at the top of the canvas *n* pixels to the right of the canvas origin. ∎

Exercise 2.1.2 Design the function *move-ufo*, which consumes a *UFO* (representation) and produces one whose anchor point is 3 pixels below the given one.

Modify *move-ufo* so that it produces a *UFO* that has dropped 3 pixels and has moved randomly to the left or right by up to 4 pixels. Hint: Use the function random, which consumes a positive number n and produces a number between 0 (inclusive) and n (exclusive). Two consecutive calls may or may not produce the same number. Design the function *random-range*, which consumes n and produces a number between $-n$ and $+n$.

Challenge: Revise *move-ran-ufo* so that a *UFO* that has disappeared on the left or right of the canvas reappears on the other side for the next time slice. Modify *fly-until-down* so that it uses *move-ran-ufo*. ∎

Exercise 2.1.3 Design the function *at-bottom?*, which consumes a *UFO* (representation) and determines whether its anchor point is level with, or below, the bottom of the canvas. ∎

Exercise 2.1.4 Design the function *draw-ufo*, which consumes the representation of a ufo and draws it on the canvas.

Also design *clear-ufo*, which consumes the representation of a ufo and clears it from the canvas.

Draw the UFO as a green disk of radius 3 around the anchor point. For the ambitious, draw the UFO as a green line of length 20 with a disk of radius 3 in the center. ∎

Now watch the UFO fly down from top to bottom (in a random walk).

2.2 AUPs

PREREQUISITE: 7.1 Varieties of Data TEACHPACK: draw.ss

Now it's time to develop our defenses. An AUP defends the bottom of the canvas where it can move left or right.

Exercise 2.2.1 Develop a (minimalist) data definition for AUPs. ∎

Exercise 2.2.2 Design the function *create-aup*, which consumes a number n and produces an *AUP* that is n pixels to the right of the canvas's left margin. ∎

Exercise 2.2.3 Design *move-aup*. The function consumes an *AUP* (representation) and a *KeyEvent*. It produces an *AUP* that has moved to the left or right by one (1) pixel, if the player has pressed the left or right arrow key; otherwise, it just returns the given *AUP*. ∎

Exercise 2.2.4 Design the function *draw-aup*, which consumes a *AUP* and draws it on the canvas.

Also design *clear-aup*, which consumes a *AUP* and clears it from the canvas.

Think of the AUP as a blue line of with 10. ∎

```
;; AUP → Boolean
;; move an AUP at most n times
(define (move-n-times n an-aup)
  (cond
    [(zero? n) true]
    [else
     (and (draw-aup an-aup)
          (sleep-for-a-while .05)
          (clear-aup an-aup)
          (move-n-times (sub1 n) (move-aup an-aup (get-key-event))))]))

...

;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(start WIDTH HEIGHT)

(move-n-times 10000 (create-aup 0))
```

Figure 4: Controlling an AUP

The program fragment in figure 4 allows players to control an *AUP*. It "loops" 10000 times and checks on keyboard events. It requires well-developed solutions for all the exercises in this subsection. Don't rush. Follow the design recipe.

2.3 Putting it all together

PREREQUISITE: 7.1 Varieties of Data TEACHPACK: draw.ss

With UFOs and AUPs in place we can create our first interactive game. Roughly speaking, we need to merge the code from figures 3 and 4 and, if we want to make it truly convenient, we should also create a function that produces an announcement about the winners and losers. Figure 5 shows the program fragment that performs all these tasks. The following exercises specify the few remaining tasks before this game can run.

```
;; String → String
;; let name defend against a UFO
(define (main name)
  (announcement
   (fly-until-down (create-ufo (/ WIDTH 2)) (create-aup (/ WIDTH 2)))
   name))

;; UFO AUP → Boolean
;; if the UFO is caught, produce true
(define (fly-until-down ufo aup)
  (cond
    [(at-bottom? ufo) (landed-on-aup? aup ufo)]
    [else
     (and (draw-scene ufo aup)
          (sleep-for-a-while .05)
          (clear-scene ufo aup)
          (fly-until-down (move-ran-ufo ufo) (move-aup aup (get-key-event))))]))

...

;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(start WIDTH HEIGHT)

(main "your name here")
```

Figure 5: Crashing a UFO

Exercises

Exercise 2.3.1 Design the function *announcement*. It consumes a *Boolean*, which represents the result of dropping the UFO and defending with an AUP, and a *String*, which is the name of the player. From these it produces a string that announces whether the player has won. Hint: The primitive string-append concatenates two strings. ∎

Exercise 2.3.2 Design the function *draw-scene*. It consumes an *AUP* and a *UFO* and draws them on the canvas.

Also design *clear-scene*, which consumes a *AUP* and a *UFO* and clears them from the canvas.

The functions should return true if both drawing or clearing actions succeed. ∎

Exercise 2.3.3 Design the function *landed-on-aup?*. It determines whether some given *UFO* has landed on a given *AUP*. Hint: Recall the geometric interpretation that goes with each data representation. Then draw pictures and determine what it means to figure out whether the two geometric shapes overlap. Approximate "landing" as best as you can; expect to see these simplifications as you play the game. ∎

2.4 One shot at the UFO

PREREQUISITE: 7.1 Varieties of Data TEACHPACK: draw.ss

The chief engineer has figured out how to let AUP's fire a gun—once. This means that an AUP now has two chances to stop the UFO. Either the AUP shoots and that one shot hits the UFO or it manages to stop the UFO via a crash. We already have a program that does the latter; let's develop a program that simulates the new ability to shoot.

Exercises

Exercise 2.4.1 Develop a data definition for representing a shot. Make up examples and show what each example means in figure 2. ∎

Exercise 2.4.2 Design the function *create-shot*, which consumes a *Posn*, representing the position of the AUP, and produces the representation of a shot that has just left the AUP. Assume the shot leaves from the middle of the AUP. ∎

Exercise 2.4.3 Design the function *move-shot*, which consumes a *Shot* (representation) and produces one that has risen 5 pixels. ∎

Exercise 2.4.4 Design the function *draw-shot*, which consumes the representation of a shot and draws it on the canvas.

Also design *clear-shot*, which consumes the representation of a shot and clears it from the canvas.

Think of a shot as a vertical red line of length 5. ∎

Now the code in figure 6 almost works for AUPs that can fire one shot. It is similar to the function in figure 5 except that it stops the game when the UFO is hit and it needs to manage key events for two functions, *move-aup* and *create-shot*, not just one.

Exercises

Exercise 2.4.5 Revise *draw-scene* and *clear-scene* from exercise 2.3.2. In addition to an *AUP* and a *UFO* they now also consume a *Shot/f*, which is defined as follows:

```
;; A Shot/f is one of the following:
;; — a shot ; (see exercise 2.4.1)
;; — false
```

Hint: Although *Shot/f* is the last argument, think of it as the primary argument for the design. ∎

Exercise 2.4.6 Design the function *hit-shot?*, which determines whether a shot has hit a UFO. The function consumes an *Shot* and a *UFO*. It produces true if there is any overlap between the *UFO* and *Shot*. Hint: Recall the geometric interpretation that goes with each data representation. Then draw pictures and determine what it means to figure out whether the two geometric shapes overlap. Approximate "hit by a shot" as best as you can; expect to see these simplifications as you play the game. Enjoy! ∎

```
;; String → String
;; let name defend against a UFO
(define (main name)
  (announcement
   (fly-until-down (create-ufo (/ WIDTH 2)) (create-aup (/ WIDTH 2)) false)
   name))

;; UFO AUP (Shot or false) → Boolean
;; if the UFO is caught, produce true
(define (fly-until-down ufo an-aup a-shot)
  (cond
    [(at-bottom? ufo) (landed-on-aup? an-aup ufo)]
    [(and (not (boolean? a-shot)) (hit-shot? a-shot ufo))
     (draw-scene ufo an-aup a-shot)]
    [else
     (and (draw-scene ufo an-aup a-shot)
          (sleep-for-a-while .05)
          (clear-scene ufo an-aup a-shot)
          (manage (get-key-event) (move-ufo ufo) an-aup (move-shot/f a-shot)))]))

;; KeyEvent UFO AUP (Shot or false) → Boolean
(define (manage ke ufo an-aup a-shot)
  (cond
    [(boolean? ke) (fly-until-down ufo an-aup a-shot)]
    [(char? ke) (fly-until-down ufo an-aup a-shot)]
    [(symbol=? ke 'up)
     (cond
       [(boolean? a-shot)
        (fly-until-down ufo an-aup (create-shot an-aup))]
       [else (fly-until-down ufo an-aup a-shot)])]
    [else (fly-until-down ufo (move-aup an-aup ke) a-shot)]))

;; (Short or false) → (Shot or false)
(define (move-shot/f a-shot)
  (cond
    [(boolean? a-shot) a-shot]
    [else (move-shot a-shot)]))
...
;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(start WIDTH HEIGHT)

(main "your name here")
```

Figure 6: Crashing or shooting down a UFO

2.5 Many Shots

PREREQUISITE: 10.2 Lists that Contain Structures TEACHPACK: draw.ss

The true goal is to simulate an AUP-UFO fight like the one in figure 2. The AUP in the figure can obviously fire many shots, not just one. Since "many" clearly means "arbitrary" and "unknown", we need a list of shots. Put differently, we need to revise all the data definitions and all the functions that deal with shots.

Exercises

Exercise 2.5.1 Develop a data definition for representing a list of *Shot*s. ∎

Exercise 2.5.2 Design the function *move-all-shots*, which consumes a list of shots and produces one where each shot has been moved with *move-shot*. ∎

Exercise 2.5.3 Design the function *draw-all-shots*, which consumes a list of shots, draws all of them, and produces true if all drawing actions succeed.

Design the function *clear-all-shots*, which consumes a list of shots, clears all of them, and produces true if all drawing actions succeed. ∎

Exercise 2.5.4 Design the function *hit-by-shot?*. It consumes a list of *Shot*s and a *UFO*. It produces true if one of the *Shot*s has hit the *UFO*; it produces false if none of the *Shot*s has hit the *UFO*. ∎

Exercise 2.5.5 Revise *draw-scene* and *clear-scene* from exercise 2.4.5. Instead of a *Shot/f*, the functions now consume a list of *Shot*s. ∎

It's time to play. And you're ready to play. The program fragment in figure 7 contains the code for a game that allows AUPs to fire many shots. After you have played enough, try to understand (write down) how the program evolved and how we planned out this series of exercises.

2.6 Eliminating Similarities

2.6.1 Abstracting Similar Functions

PREREQUISITE: 19 Similarities in Definitions
TEACHPACK: draw.ss

```
;; String → String
;; let name defend against a UFO
(define (main name)
  (announcement
   (fly-until-down (create-ufo (/ WIDTH 2)) (create-aup (/ WIDTH 2)) empty)
   name))

;; UFO AUP (Shot or false) → Boolean
;; if the UFO is caught, produce true
(define (fly-until-down ufo an-aup shots)
  (cond
    [(at-bottom? ufo) (landed-on-aup? an-aup ufo)]
    [(hit-by-shot? shots ufo) (draw-scene ufo an-aup shots)]
    [else
     (and
       (draw-scene ufo an-aup shots)
       (sleep-for-a-while .05)
       (clear-scene ufo an-aup shots)
       (manage (get-key-event) (move-ufo ufo) an-aup (move-all-shots shots)))]))

;; KeyEvent UFO AUP (Shot or false) → Boolean
(define (manage ke ufo an-aup shots)
  (cond
    [(boolean? ke) (fly-until-down ufo an-aup shots)]
    [(char? ke) (fly-until-down ufo an-aup shots)]
    [(symbol=? ke 'up)
     (fly-until-down ufo an-aup (cons (create-shot an-aup) shots))]
    [else (fly-until-down ufo (move-aup an-aup ke) shots)]))
...
;; Constants
(define WIDTH 200)
(define HEIGHT 500)

;; run program, run
(start WIDTH HEIGHT)

(main "your name here")
```

Figure 7: Crashing or shooting down a UFO with many shots

Take a second look at figure 1. The function *drive-ball* relies on two auxiliary functions: *draw-shape* and *clear-shape*. If the shape is just a ball, the two functions just draw and clear a solid disk, respectively:

```
;; Posn → true
(define (draw-shape a-posn)
  (draw-solid-disk a-posn 3 'red))
```

```
;; Posn → true
(define (clear-shape a-posn)
  (clear-solid-disk a-posn 3 'red))
```

Now suppose we want the same program to move a stick figure, not a plain ball. In that case, the program would have to draw and clear the disk and a thin rectangle below the disk. That is, we would need to design these two functions:

```
;; Posn → true
(define (draw-shape2 a-posn)
  (and
    (draw-solid-rect a-posn 1 10 'red)
    (draw-solid-disk a-posn 3 'red)))
```

```
;; Posn → true
(define (clear-shape2 a-posn)
  (and
    (clear-solid-rect a-posn 1 10 'red)
    (clear-solid-disk a-posn 3 'red)))
```

Clearly, the two pairs of functions are prime examples of functions with similar definitions. We can easily abstract over both pairs. For the second pair, we get this general function:

```
;; (Posn Number Number Symbol → true) (Posn Number Symbol → true) Posn → true
(define (graphics-shape2 solid-rect solid-disk a-posn)
  (and (solid-rect a-posn 1 10 'red)
       (solid-disk a-posn 3 'red)))
```

To get back the two functions, we just pass in the appropriate primitives:

```
;; Posn → true
(define (draw-shape2 p)
  (graphics-shape2
    draw-solid-rect draw-solid-disk p))
```

```
;; Posn → true
(define (clear-shape2 p)
  (graphics-shape2
    clear-solid-rect clear-solid-disk p))
```

The advantage is that, in many cases, we can now change just a single function to get an entirely new shape to move across a canvas.

Exercises

Exercise 2.6.1 Modify *graphics-shape2* so that the *drive-ball* program in figure 1 moves

1. a square of size 3;

2. a cross-hair that intersect at the given *Posn*;

3. a pair of intersecting disks (each of size 5) that contain the given *Posn* in their intersection. ∎

Exercise 2.6.2 Develop an abstract function for *draw-ufo* and *clear-ufo* from exercise 2.1.4. ∎

Exercise 2.6.3 Develop an abstract function for *draw-aup* and *clear-aup* from exercise 2.2.4. ∎

Exercise 2.6.4 Develop an abstract function for *draw-scene* and *clear-scene* from exercise 2.3.2. ∎

Exercise 2.6.5 Develop an abstraction for *draw-shot* and *clear-shot* from exercise 2.4.4. ∎

2.6.2 Using Loops

PREREQUISITE: 21.2 Finger Exercises with Abstract List Functions

In addition to abstracting over similar functions, it is also good practice to define functions with applications of existing abstractions. Scheme provides a number of "loops", i.e., functions that traverse a piece of data and apply some give function to each "stop" during the traversal. For example,

```
(map move-shot
     (list (make-posn 100 500)
           (make-posn 100 460)
           (make-posn 120 420)))
```

applies the function *move-shot* to each *Shot* (i.e., *Posn*) on the given list. The result is the list

```
(list (make-posn 100 492) (make-posn 100 452) (make-posn 120 412))
```

In short, the expression moves an entire list of shots.

Exercises

Exercise 2.6.6 Use map to define *move-all-shots* from exercise 2.5.2. ∎

Exercise 2.6.7 Use andmap to define *draw-all-shots* and *clear-all-shots* from exercise 2.5.3. ∎

Exercise 2.6.8 Use ormap to define *hit-by-shot?* from exercise 2.5.4. ∎

Now that play some more, but use this second draft of the program.