How to Design Worlds:
Imaginative Programming in DrScheme

Matthias Felleisen
Robby Findler
Kathi Fisler
Matthew Flatt
Shriram Krishnamurthi

Copyright © 2008

This work is licensed under the
Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.
If you create a derivative work, please include the version information below in your attribution.

This book is available free-of-cost from `http://world.cs.brown.edu/`.
This is **version 1**, generated on August 18, 2008.

Contents

Preface: Welcome to Our World

Here are some examples of interactive programs being run:

```
http://world.cs.brown.edu/1/projects/spaceflight/v11-win.swf
http://world.cs.brown.edu/1/projects/flight-lander/v9-success.swf
http://world.cs.brown.edu/1/projects/firefighter/final.swf
http://world.cs.brown.edu/1/projects/chicken/v6-win.swf
http://world.cs.brown.edu/1/projects/cowabunga/v7.swf
```

You can write these and other programs with just a few weeks of programming knowledge (and simpler versions of them with even less). This book will teach you how.

Writing interactive games and animations is both instructive and fun. Writing such programs is, however, often challenging. Interactive programs are complex because the user, not the programmer, is in charge of the program's execution: the program itself is often a passive receiver of commands, reacting only when some stimulus in the external world (whether a user's keystroke or the tick of a clock) occurs. Testing such programs appears even harder because without the interaction there appears to be no program to speak of, but automating the testing of interactive programs is especially challenging. Developing clean, well-tested interactive programs therefore seems especially daunting to beginners.

Over the past few years, we have built an infrastructure called the World that makes writing such programs accessible even to rank beginners. In our approach, programmers write pure, non-interactive functions. These functions are no harder to test than any other functions the beginning programmer might write. The World then enables the programmer to combine these functions in ways that respond to various stimuli—keyboard and mouse inputs, timers, etc.—with little overhead. As a result, well-tested non-interactive functions are easily turned into the key portions of surprisingly sophisticated interactive games, drawing packages, and other applications. This book acquaints you with the World and with the steps you must take to use it to write programs.

This book is, however, not a self-contained introduction to programming. For now, we intend it as a companion volume to *How to Design Programs* (HTDP) by Felleisen, Findler, Flatt, and Krishnamurthi. HTDP is already decomposed into progressively enriched data-structures. Therefore, this book does not teach Scheme or the design recipe; it assumes knowledge of both from HTDP. It instead focuses on incremental projects that build upon increasingly richer data representations to build applications. You should also look at the prologue of the next edition of HTDP.

Some projects are presented as a whole, while others are divided into several sections. These sections are placed in categories based on the data they require, and these categories are ordered in increasing complexity. The early versions of these projects are, therefore, likely to be only very coarse approximations of the final versions; as a result, however, this style of presentation offers examples of how to decompose a large project into simpler, incremental versions. The emphasis is on always having a running version of the program, even if it isn't as interesting as it could be. This gives the programmer a sense of increasing accomplishment, thereby motivating progress.

Acknowledgments

All the solutions were prepared by Danny Yoo (WPI) and Zhe Zhang (Brown), individually and collaboratively. We thank them for their meticulous and imaginative work. Danny is responsible for suggesting that the aliens abduct cows. Thanks also to Danny for catching errors in the prose and for his Flash generator. Thanks also to Renee Ciezki for catching an error.

We thank the numerous university and school educators who have tested each version of this infrastructure and provided valuable feedback.

We developed our approach in response to other like-minded efforts in programming languages research. Of these, Greg Cooper's FrTime programming language stimulated our thinking about interactive programs, while Paul Hudak's book, *The Haskell School of Expression*, was similarly thought-provoking.

Finally, we especially thank Emmanuel Schanzer for his inspired and inspiring work on the Bootstrap project, which presents this approach to a whole new class of students.

Using This Book for Self-Study

If you are reading this book outside a formal academic context—to teach yourself programming, say—welcome! All the material you need should be available from the book's Web site. In particular, we've made an effort to provide complete solutions to some of the exercises in the book, so you can check your work against our own solutions.

Note that this is a book about programming; it therefore implicitly *asks you to write programs*. Most of the development of the projects takes place in exercises; we offer only partial advice and then ask you, the reader, to construct solutions. Therefore, this book is not meant for armchair reading. If you don't write the programs, you won't learn what this book has to offer.

Chapter 1

How to Obtain and Use the World

Use version 4.1 of DrScheme or later (or if you must use version 4, replace *key=?* with *equal?* in this book). DrScheme contains a Teachpack called `world.ss`. To install it, select Choose Teachpack... from the Language menu, and add the `world.ss` Teachpack. Click Run. DrScheme will register the use of the Teachpack, and make its procedures available for use.

DrScheme treats images just like numbers, symbols, and other Scheme values, which means you can write functions that consume and produce them. Here are just a few of these functions (look in Help Desk under the Help menu for many more):

circle :: *num* × *mode* × *color* → *image*
nw:rectangle :: *num* × *num* × *string* × *string* → *image*
empty-scene :: *num* × *num* → *scene*
place-image :: *image* × *num* × *num* × *scene* → *scene*

Advanced Material

> You might wonder what the prefix *nw:* stands for, or the distinction between a "scene" and an "image". Every image has a *pinhole*, which is the point where the image is anchored. When you combine images, their pinholes determine how they line up (hence the name "pinhole": imagine a hole in each image through which you're passing a string). For a circle, there is an obvious pinhole: its center. For a rectangle there is no single obvious pinhole; *nw:rectangle* creates a rectangle whose pinhole is located at the top-left corner. A scene is just an image whose pinhole is at coordinate $(0,0)$. For most programs in this book, you can safely ignore pinholes by focusing solely on scenes.

The World Teachpack also provides the following functions:

big-bang :: *num* × *num* × *num* × *world* → *true*
on-tick :: (*world* → *world*) → *true*
on-key-event :: (*world* × *key-event* → *world*) → *true*
on-redraw :: (*world* → *scene*) → *true*
stop-when :: (*world* → *bool*) → *true*

Chapter 2

How to Obtain and Use Images

We've provide images for each of the projects; you can find them on the project Web pages.

Of course, you don't have to use our images at all! Indeed, part of the fun is in customizing these programs to look as you want them. The Internet is full of image galleries (search for the phrase "clip art") where you can find imaginative and interesting pictures to use in your programs.

On the other hand, beware: it's easy to get bogged down in searching for images while failing to write proper and correct programs in the first place. We therefore suggest that you always begin your programs with extremely simple images—rectangles and circles, for instance—and get the program logic right. Think of searching in clip-art galleries as dessert. It's okay to sample some of your dessert early, but you do want to leave some for the end, don't you?

Part I

Introduction

Chapter 3

Project Descriptions

The projects are as follows:

Chicken The chicken wants to cross the road, for reasons that remain a universal mystery. You must help it dodge obstacles traveling in different directions and at varying velocities. [§ 7]

The solutions to this problem are public.

Cowabunga! You're an alien. You've been sent on a mission to find and abduct a cow. You must guide your craft to land on one of many moving cows. Your spacecraft's propulsion means you cannot come in contact with the ground, or you will crash and get taken by federal agents for "experiments". [§ 8]

The solutions to this problem are available to instructors only.

Firefighter The land is on fire! You must fly a plane that drops water on the flames to quench them. The plane has a limited amount of water, and flames that are not fully extinguished eventually recover. [§ 10]

This project is presented without decomposition, so the reader must develop it in its entirety. The solutions to this problem are available to instructors only.

Flight Lander You are piloting a plane. You can move it up and down, but it has only a limited amount of fuel, and increasing elevation consumes fuel. The force of gravity increases the rate of descent. The plane must land on ground, not on water. Several hot-air balloons are moving through the air; colliding with one would, of course, be calamitous. [§ 5]

This project is done in full detail, to serve as a demonstration.

Spaceflight The planets move around a sun; some of the planets have moons that revolve around them. Your spacecraft must travel from one planet to the other without exhausting its fuel and without flying into the sun. It can use the sun's mass as a gravitational sling-shot. [§ 9]

The solutions to this problem are available to instructors only. This is a fairly challenging problem that requires some understanding of elementary physics.

Chapter 4

How the World Works

Most readers might find the following presentation somewhat abstract on first reading. If you do, skip it and dive into an example. When you find yourself confused or curious about how World programs really work, then you'll be ready to read this chapter.

4.1 What is the World?

The World is a data structure that represents everything about the program that might change under program control. We indicate both *that* something has changed and *how* it changes in the same way: by writing functions. These functions consume one value for the World—representing the old World—and produce another value for it, representing the new World. Of course, sometimes the world does not change; a function represents this by producing the same world that it consumed (i.e., behaving as the identity function).

Note that because the World only represents things that change, these functions may require additional information, namely facts about the world that *don't* change. These facts includ what image to use to represent an object, what color to draw a fixed background, etc. Because the parameters of functions represent those parts of it that are "variable", this fixed information is encoded directly in the function instead of being passed as a parameter. Of course, instead of hard-coding this information directly inside the function, it is good practice to define Scheme constants (using **define**) and refer to those in the function body.

4.2 When Does the World Change?

The World can change in response to many stimuli. The most common ones you will use are:

- clock ticks,
- keystrokes and mouse behavior.

We call both of these *events*. Conceptually, however, they represent two very different forms of stimuli. The former happens continuously, independent of any user input; the latter happens only when the user performs a keyboard or mouse action (which may even be never at all). Therefore, changes in your program that need to happen independent of the user (e.g., updating the location of a moving object) should be tied to the clock rather than to user input.

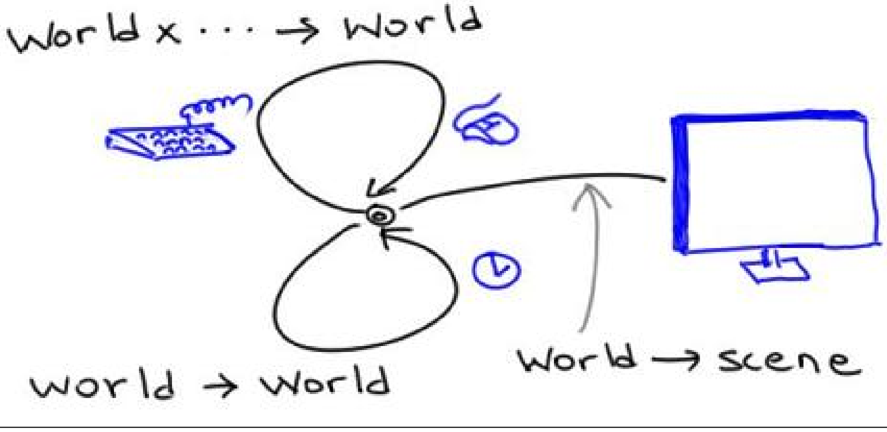
4.3 Putting Together the Pieces

Ultimately, you will write several functions that consume the current representation of the World. All these functions will fall into one of two categories. Some of these functions will compute new Worlds, while others will convert the World into a representation on the screen (which we call a *scene*).

A function that responds to clock tick events consumes a World and generates a (potentially updated) World. The functions that respond to keyboard and mouse events require some additional information indicating *what* behavior occurred: which key the user pressed or released, which mouse button was pressed, and so on. Be sure to study the contracts in § 1 in more detail.

Exercise 4.3.1 *Why does the function responding to clock ticks not get additional information, such as how much time has elapsed since the game began? Given that it does not, supposing a particular program needed this information (e.g., in a game where the score depends on how long the player takes to finish a task), is there any way of reconstructing this information?*
Hint: *Remember, there are no a priori constraints on the data structure that represents the World. Think about what the World* needs *to represent, and work from there towards the data structure.*

The picture below—called an *automaton*—shows how these functions fit together. On every tick, the World Teachpack applies a function to compute a new World value. Likewise, on every keyboard and mouse event, the Teachpack applies corresponding functions you define to compute new World values. Whenever the World changes, the Teachpack applies a function you provide that converts the current World into a scene, which it presents on the display.

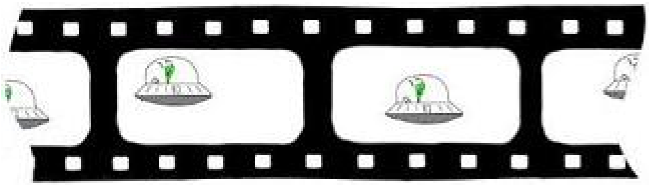


Note

You might just wonder what happens if two things happen at the same time: e.g., a keystroke and a mouse movement, or a keystroke and a clock tick. On your computer, it is essentially impossible for things to happen at *exactly* the same time: your computer will order (or *sequentialize*) these so that one happens before the other. Therefore, the picture above remains accurate.

4.4 The Movie Principle

Many World programs are exactly like motion pictures (i.e., movies). In a movie, the characters do not actually "move". Rather, a movie consists of a series of scenes, and in each one the characters are in slightly different positions:



When these are shown quickly enough, your eye and brain imagine movement. (The same principle applies to animated films such as cartoons, except most cartoons don't try to be "realistic". When you design a World program, you have the choice of how realistic you make your visual output.)

World programs that consist of animations behave the same way. Your program displays a scene, computes and displays a new scene, and another new scene, and so on. When the program does this in sufficiently rapid succession, the viewer will perceive motion. For many of your programs a good choice of update frequency is `1/30` (i.e., update thirty times per second), because that speed is generally fast enough to give humans the impression of motion.

Part II

A Worked Example

Chapter 5

Flight Lander

5.1 On-Line Material

All the material referenced in this chapter is on-line:

`http://world.cs.brown.edu/1/projects/flight-lander/`

In particular, you can find all the videos referenced in this chapter at that Web page.

5.2 The Problem Setup

First, watch this video:

`http://world.cs.brown.edu/1/projects/flight-lander/v9-success.swf`

It shows a plane, controlled by a user operating the keyboard, landing successfully after avoiding obstacles. Now watch these three videos:

`http://world.cs.brown.edu/1/projects/flight-lander/v9-collide.swf`

`http://world.cs.brown.edu/1/projects/flight-lander/v9-sink.swf`

`http://world.cs.brown.edu/1/projects/flight-lander/v9-out-of-gas.swf`

They show three unfortunate cases of pilot error (where the "pilot" is the user at the keyboard): the plane colliding with a balloon, the plane landing on water, and the plane running out of fuel.

By the end of this chapter, you will have written all the relevant portions of this program. Your program will: animate the airplane to move autonomously; detect keystrokes and adjust the plane accordingly; account for the use of fuel; detect collisions with balloons; and check for landing on water and on land. Because this is a lot to do, we will build up to this result incrementally. As a result the initial versions will seem very staid, but the complexity will accumulate over versions.

5.3 Version: Plane Moving Across Screen

We will start with the simplest version of this program: one in which the airplane moves horizontally across the screen. Watch this video:

`http://world.cs.brown.edu/1/projects/flight-lander/v1.swf`

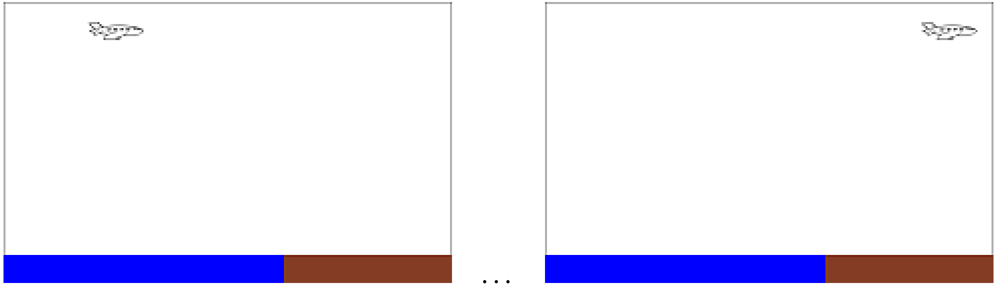
To create this scene, first we need an image of an airplane. Here's the one we used:

which you can obtain from our Web site (§ 2). Once we've chosen an image for the plane, let's give it a name in Scheme:

(**define** *PLANE*)

Henceforth, we can use *PLANE* whenever we need to refer to this image.

Now look again at the video. Specifically, let's look at two separate moments in time:



What's common to these two images? The water and the land. What's different? The position of the plane.

Note

> The *World* consists of everything that changes. Things that stay the same do not get recorded in the World.

Given this, we can define our first World:

The World

> The World is a number, representing the *x*-position of the plane on the screen.

5.3.1 How World Programs Work

To actually draw the movie we've seen above, we have to do two things:

- Update the World as time passes.
- Given a World, produce the corresponding visual display.

As time changes, DrScheme will consult our program to determine how to update the value of the World, then consult it again to determine how to present this to the user, and repeat this forever (or until we tell it to stop); for a more detailed description, see § 4. Now let's write each of these steps in turn.

5.3.2 Updating the World

The plane does not actually "move". Rather, the World Teachpack runs a clock; on every tick it updates all the information in the World, uses the World to generate a new scene, erases the old scene, displays the new one in its place... and repeats forever. By generating new scenes frequently enough, we get the impression of motion. (If you want to understand better how "motion" works, see § 4.4.)

How do we update the plane's position? Because the World consists of just the *x*-coordinate of the plane, to move it to the right, we simply add some value to the current value of the world. Let's first give this distance a name, to make it easier to change later:

```
(define PLANE-MOVE-X 20)
```

Given this, what is the plane's new *x*-coordinate?

```
(+ ; current x-position
   PLANE-MOVE-X)
```

(The *y*-coordinate doesn't change in this initial version.) Since we have a "variable expression", we should make a function out of this. Let's first understand the intended behavior of this function:

```
(check-expect (move-plane-x-on-tick 0) (+ 0 PLANE-MOVE-X))
(check-expect (move-plane-x-on-tick 10) (+ 10 PLANE-MOVE-X))
(check-expect (move-plane-x-on-tick 800) (+ 800 PLANE-MOVE-X))
```

Once we've written sufficiently many test cases, we can derive the function's definition from them:

```
(define (move-plane-x-on-tick x)
  (+ x PLANE-MOVE-X))
```

Before you proceed, be sure to run the tests!

To actually update the plane's position periodically, we must (a) tell the World how often to move the plane, and (b) tell it how to compute the plane's new position. To do the latter, we simply notify the World of the function that computes the new position given the current one:

```
(on-tick-event move-plane-x-on-tick)
```

We'll see how to do the former below, in § 5.3.4.

5.3.3 Displaying the World

All World programs produce output using graphics. Specifically, a World program must generate a *scene*. The exact technical definition of a scene needn't concern us for now; all we need to know are the operations described in § 1 for creating and manipulating scenes.

Recall that we said that the World represents only those attributes that change, not the ones that stay the same. This is because we don't need to keep track of the latter as they change, because they don't! Instead, we always know where they are. For instance, the land and water are at fixed positions, so we can simply draw them when we need to produce a new scene. Specifically, here's the simplest kind of scene, an empty one:

```
(empty-scene 800 500)
```

Because we might change the size of the scene, though, it would be wise to record its size using constants:

```
(define WIDTH 800)
(define HEIGHT 500)
(empty-scene WIDTH HEIGHT)
```

Now we can overlay the land and water. Again, it's useful to have some constants:

```
(define BASE-HEIGHT 50)
(define WATER-WIDTH 500)
```

Using these constants, here's an image to represent water:

```
(define WATER (nw:rectangle WATER-WIDTH BASE-HEIGHT "solid" "blue"))
```

and another one to represent land:

```
(define LAND (nw:rectangle (− WIDTH WATER-WIDTH) BASE-HEIGHT "solid" "brown"))
```

Be sure to check the value of each of these constants to make sure they look as you would expect. Now that you have the individual bits of water and land, we have to combine these to create our scene:

```
(define BACKGROUND
  (place-image WATER
               0
               (− HEIGHT BASE-HEIGHT)
               (place-image LAND
                            WATER-WIDTH
                            (− HEIGHT BASE-HEIGHT)
                            (empty-scene WIDTH HEIGHT))))
```

Once again, make sure this image is the one you expected!

Now that we know how to draw the background scene, we're ready to place the plane on top of it. The expression to do so would look like this:

```
(place-image PLANE
             ; some x position here
             50
             BACKGROUND)
```

but what *x* position do we use? Actually, that's just what the World represents. So we create a function out of this expression:

```
(define (place-plane-x x)
  (place-image PLANE
               x
               50
               BACKGROUND))
```

Happily, the World Teachpack requires that we provide a function that, given the current value of the World, will produce a scene that uses it appropriately. Having defined just this function, we must simply notify the World Teachpack about it:

```
(on-redraw place-plane-x)
```

It's difficult to test such a program, isn't it? Sometimes errors in one part will temporarily cancel out errors elsewhere, briefly giving the impression everything works. Also, you may not run the program long enough to get to the parts that don't work. This is why its imperative that you *test early*: test every single function you write on as many cases as you can. The World infrastructure makes this kind of development especially easy, since most of the work is done by your functions, each of which you can test independent of graphics and interaction. So you have no excuse for not testing extensively before you start to run your final program.

5.3.4 Watching it Work

Now that we've seen how to update the world and how to display upon each update, we just need to tell DrScheme how often to process updates. For many programs, a good update rate is 30 times per second, which is typically the rate necessary for humans to perceive smooth and continuous motion. We therefore invoke the *big-bang* operation, provided by the World Teachpack, to tell DrScheme to initiate an animation:

```
(big-bang WIDTH HEIGHT 1/30 0)
```

The four parameters are as follows. The first two specify the width and height of the enclosed scene (usually, these will be the same as the size of the scene our program creates). The third specifies how often the World should move the plane (`1/30` means "do it thirty times in one second"). The fourth parameter gives the initial value of the World—in this case, where on the *x*-axis the plane should begin (we choose `0` to represent the left edge of the window).

Once you've entered the above expressions, you're ready to see the program in action! Be sure to load the World Teachpack (§ 1) and then run your program. When you do, you should see essentially the same image as in the video.

Advanced Material

> If we want the plane to appear to move faster, we can achieve this by altering at least two different parameters in the above animation. What are these, and how must they be altered? Explain, and test your answer by modifying the animation. This demonstrates that the frequency we supply to the World isn't the only factor that determines how fluid an animation will appear.

5.4 Version: Wrapping Around

When you run the program above, you notice that after a while, the plane just disappears. This is because it has gone past the right edge of the screen; it is still being "drawn", but in a location that you cannot see.

That's not very useful![1] Instead, when the plane is about to go past the right edge of the screen, we'd like it to reappear on the left by a corresponding amount—”wrapping around”, as it were.

Here's the video for this version:

`http://world.cs.brown.edu/1/projects/flight-lander/v2.swf`

Let's think about what we need to change. Clearly we need to modify the function that updates the plane's location, since this must now reflect our decision to wrap around. But the task of drawing the plane doesn't change at all: we've already made it depend on *where* we draw the plane, and *how* we draw it hasn't changed at all! Similarly, notice that the definition of the World does not change, either.

Therefore, we need to modify only *move-plane-x-on-tick*. The Scheme function *modulo* is just what we need here.[2] That is, we want the *x*-location to always be modulo the width of the scene:

```
(define (move-plane-wrapping-x-on-tick x)
  (modulo (+ x PLANE-MOVE-X)
          WIDTH))
```

Notice that, instead of copying the content of the previous definition, we can simply reuse it,

```
(define (move-plane-wrapping-x-on-tick x)
  (modulo (move-plane-x-on-tick x)
          WIDTH))
```

which makes our intent clearer: compute whatever position we would have before, but adapt the coordinate to remain within the scene's width.

Well, that's a *proposed* re-definition. Be sure to test this function thoroughly: it's tricker than you might think! Have you thought about all the cases? For instance, what happens if the plane is half-way off the right edge of the screen?

Once you have a satisfactory definition for this function, you should be able to run your program. Be sure to do so and ensure the plane behaves as desired.

Advanced Material

> It *is* possible to instead leave *move-plane-x-on-tick* unchanged and perform the modular arithmetic in *place-plane-x* instead. We choose not to do that for the following reason. In this version, we really do think of the plane as circling around and starting again from the left edge (imagine the world is a cylinder...). Thus, the plane's *x*-position really does keep going back down. If instead we allowed the World to increase monotonically, then the World would really be representing the total distance traveled, contradicting our definition of the World.

[1] Also, after a while you will get an error because DrScheme is being asked to draw the plane in a location beyond what the graphics system in the computer can manage.

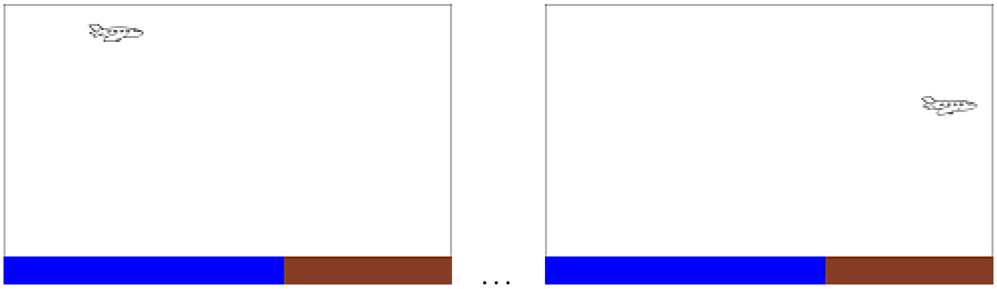
[2] Don't know what it does? Look it up in the Help Desk, under the Help menu.

5.5 Version: Descending

Of course, we need our plane to move in more than just one dimension: to get to the final game, it must both ascend and descend as well. For now, we'll focus on the simplest version of this, which is a plane that continuously descends. Here's a video:

`http://world.cs.brown.edu/1/projects/flight-lander/v3.swf`

From this, let's examine two scenes:



What's staying the same? Once again, the water and the land. What's changing? The position of the plane. But, whereas before the plane moved only in the *x*-direction, now it moves in both *x* and *y*. That immediately tells us that our definition of the World is inadequate, and must be modified.

Observe that the World must provide enough information that, just given that (and knowledge of the things that don't change), we can re-construct the current scene. As we can tell from the images above, we can perform that reconstruction given just the current position of the plane in the *x*- and *y*-dimensions. But how do we represent this new kind of position?

We could define our own structure to hold this pair of numbers, but fortunately DrScheme already provides us with a structure for this purpose: *posn*, short for *position*. Recall that we have the following operations at our disposal:

make-posn :: *num* × *num* → *posn*
posn-x :: *posn* → *num*
posn-y :: *posn* → *num*

Given this, we can revise our definition:

The World

The World is a *posn*, representing the *x*- and *y*-position of the plane on the screen.

We've just changed our data definition for the World; now the design recipe becomes invaluable! It tells us to examine everything that consumes or produces something matching this data definition, and adjust it accordingly. We've written two operators so far—*move-plane-wrapping-x-on-tick* and *place-plane-x*—and they both consume (and *move-plane* produces) a World, so we have to re-examine both of them.

5.5.1 Moving the Plane

First, let's consider *move-plane-wrapping-x-on-tick*. Previously our plane moved only in the *x*-direction; now we want it to descend as well, which means we must add something to the current *y* value. We'll call this offset *PLANE-MOVE-Y*:

```
(define PLANE-MOVE-Y 5)
```

Let's write some test cases for the new function. Here's one:

```
(check-expect (move-plane-xy-on-tick (make-posn 10 10)) (make-posn 30 15))
```

Another way to write the expected answer is to re-use the functions we have already written and tested earlier:

```
(check-expect (move-plane-xy-on-tick (make-posn 10 10))
              (make-posn (move-plane-wrapping-x-on-tick 10)
                         (+ 10 PLANE-MOVE-Y)))
```

Note

Which method of writing the tests is better? *Both!* They each have different advantages:

- The former method has the benefit of being very concrete: there's no question what you expect, and it demonstrates that you really can compute the desired answer from first principles.
- The latter method has the advantage that, if you change the constants in your program (such as the width of the program), seemingly correct tests do not suddenly fail.

Ideally, you should reconcile this distinction by splitting the difference: write the answer as concretely as possible (the former style), but using the constants to compute the concrete answer (the advantage of the latter style).

Before you proceed, have you written enough test cases? Are you sure? Have you, for instance, tested what should happen when the plane is near the edge of the screen in either or both dimensions? We thought not—go back and write more tests before you proceed!

Using the design recipe, be sure to rewrite *move-plane-wrapping-x-on-tick*. You should get something like this:

```
(define (move-plane-xy-on-tick p)
  (make-posn (move-plane-wrapping-x-on-tick (posn-x p))
             (+ (posn-y p) PLANE-MOVE-Y)))
```

The new definition both consumes and produces *posn*s. Note that what it does with the *x* component of the *posn* is exactly the same as what the earlier *move-plane* did to the World overall (when the world was just the plane's *x*-position), which is why we reused the previous definition. Be sure to test this thoroughly!

5.5.2 Drawing the Scene

We have to also examine and update *place-plane-x*. If we examine our earlier definition we notice that we placed the plane at an arbitrary *y*-coordinate; now we have to take the *y*-coordinate from the World:

```
(define (place-plane-xy p)
  (place-image PLANE
               (posn-x p)
               (posn-y p)
               BACKGROUND))
```

Don't forget to test this, too. (Notice that we can't really reuse the previous definition, because it hard-coded the *y*-position, which we must now make a parameter.)

5.5.3 Finishing Touches

Are we done? It would seem so: we've examined all the procedures that consume and produce *posn*s and updated them appropriately. Actually, we're forgetting one small thing: the fourth parameter to *big-bang* is the initial value of the World. If we've changed the data definition of the World, we'll need to reconsider this parameter, too. Here's a reasonable initial value under the new definition of the World:

```
(big-bang WIDTH HEIGHT 1/30 (make-posn 0 0))
```

Advanced Material

> It's a little unsatisfactory to have the plane truncated by the screen. You can use *image-width* and *image-height* to obtain the dimensions of an image, such as the plane. Use these to ensure the plane fits entirely within the screen for the initial scene, and similarly in *move-plane-xy-on-tick*.

5.6 Version: Responding to Keystrokes

Now that we have the plane descending, there's no reason it couldn't ascend as well. Here's a video:

`http://world.cs.brown.edu/1/projects/flight-lander/v4.swf`

We'll use the keyboard to control its motion: specifically, the "up" key will make it move up, and the "down" key will make it descend further (natch). The World Teachpack makes it easy to write programs that react to keystrokes by providing a function that consumes a description of the keystroke and the current World, and generates a (possibly different) World in response.

There are many keys on the keyboard, but for now we're interested only in two. Therefore, for us a keystroke is:

> A *key* is one of
>
> - 'up, representing the up-arrow

- 'down, representing the down-arrow
- other symbols, representing othey keystrokes

This dictates the following template:

```
(define (f a-key)
  (cond
   [(key=? a-key 'up) ... ]
   [(key=? a-key 'down) ... ]
   [else ... ]))
```

Now we can fill in the fragments:

```
(define KEY-DISTANCE 10)

(define (alter-plane-y-on-key p a-key)
  (cond
   [(key=? a-key 'up)
    (make-posn (posn-x p)
               (− (posn-y p) KEY-DISTANCE))]
   [(key=? a-key 'down)
    (make-posn (posn-x p)
               (+ (posn-y p) KEY-DISTANCE))]
   [else p]))
```

Notice that if we receive any key other than the two we expect, we leave the World as it was: from the user's perspective, this has the effect of just ignoring the keystroke. The structure of the data definition for keys suggests we ought to use an **else** clause; what we do in it (whether leave the world unchanged, or halt with an error, or perform some other action) is determined by what user interface we wish to provide.

No matter what you choose, be sure to test this! Can the plane drift off the top of the screen? How about off the screen at the bottom? Can it overlap with the land or water?

Once we've written and thoroughly tested this function, we simply need to notify the World Teachpack about its existence:

```
(on-key-event alter-plane-y-on-key)
```

Now your plane moves not only with the passage of time but also in response to your keystrokes. You can keep it up in the air forever!

5.7 Version: Landing

Remember that the objective of our game is to *land* the plane, not to keep it airborne indefinitely. That means we need to detect when the plane reaches the land or water level and, when it does, terminate the animation:

`http://world.cs.brown.edu/1/projects/flight-lander/v5.swf`

First, let's try to characterize when the animation should halt. This means writing a function that consumes the current World and produces a boolean value: true if the animation should halt, false otherwise. This requires a little arithmetic based on the plane's size:

```
(define (on-land-or-water? p)
  (>= (posn-y p) (− HEIGHT BASE-HEIGHT)))
```

Advanced Material

> Note that this isn't quite right; it doesn't take into account the size of the plane's image. Fix that problem.
>
> Now try using a different plane image. Does your definition still work correctly?

We need to notify the World Teachpack of the existence of this function:

```
(stop-when on-land-or-water?)
```

And with that, you should be able to ensure that your program does correctly check for the flight terminating (whether happily or unhappily).

Advanced Material

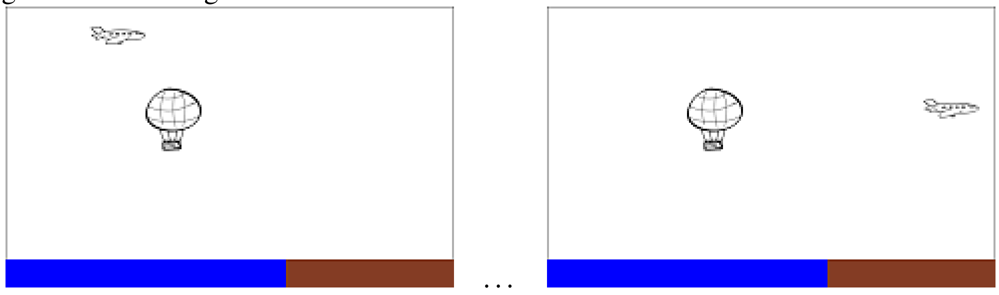
> In this version, the plane stops moving when it touches land or water. Extend it so that:
>
> 1. The plane stops on touching land, but continues descending (glug!) if it comes in contact with water, until it has completely submerged.
> 2. When the plane is under water, we should see only the portion of it that is still above water level.
>   **Hint**: This requires writing a fresh version of *place-plane-xy*.

5.8 Version: A Fixed Balloon

Now let's add a balloon to the scene. Here's a video of the action:

`http://world.cs.brown.edu/1/projects/flight-lander/v6.swf`

Consider again two still images from this video:



Notice that while the plane moves, everything else—including the balloon—stays immobile. Therefore, we do *not* need to alter the World to record the balloon's position. All we need to do is alter the conditions under which the program halts: effectively, there is one more situation under which it terminates, and that is a collision with the balloon. Therefore, we only need to alter the definition of the procedure we feed to *stop-when*.

When does the game halt? There are now two circumstances: one is contact with the balloon, and the other is contact with the land or water. The latter remains unchanged from what it was before, so we can focus on the former.

Where is the balloon, and how do we represent where it is? The latter is easy to answer: that's what *posn*s are good for. As for the former, we can decide where it is:

```
(define BALLOON (make-posn 600 300))
```

or we can allow Scheme to pick a random[3] position:

```
(define BALLOON (make-posn (random WIDTH) (random HEIGHT)))
```

(In practice, we might want to prevent the balloon from appearing too close to the edges, so we get the full effect of its presence.)

Given a position for the balloon, we just need to detect collision. Actually, this isn't so easy. DrScheme cannot tell us when two images overlap,[4] so we're left to identify a way to do this for ourselves. One simple way is as follows: determine whether the distance between the plane and the balloon is within some threshold:

```
(define (overlapping? plane-posn balloon-posn)
  (< (distance plane-posn balloon-posn)
     THRESHOLD))
```

where *THRESHOLD* is some suitable constant (which, ideally, should be computed based on the sizes of the plane and balloon images).

Advanced Material

> Of course, this definition of *overlapping?* does not take into account the shapes of planes and of balloons. We might want to think of the former as rectangles and the latter as circles, or even as combinations of these shapes, and accordingly write a more convincing check for collision. Use your imagination and your knowledge of geometry and of conditionals to improve this function.

What is the function *distance*? It consumes two *posn*s and a number and determines whether Euclidean distance between the *posn*s is smaller than the number. Let's write it. First a header, contract and template:

```
;; distance :: posn × posn × num → bool
;; consumes two positions and a threshold and determines whether
;; the distance between the positions is less than the threshold
```

[3] Given the parameter *n*, *random* will return a whole number in the range $[0,n)$.

[4] Do you see why not? Think about an "image" that consists of a lot of blank space with the actual image somewhere inside.

```
(define (distance p1 p2 t)
  . . . (posn-x p1) . . . (posn-y p1) . . .
  . . . (posn-x p2) . . . (posn-y p2) . . . )
```

Here are a few tests to get you started:

```
(check-expect (distance (make-posn 0 0) (make-posn 1 1) 1) false)
(check-expect (distance (make-posn 0 0) (make-posn 1 1) 2) true)
(check-expect (distance (make-posn 0 0) (make-posn 5 12) 12) false)
(check-expect (distance (make-posn 0 0) (make-posn 5 12) 13) true)
```

Be sure to write more! Once we do, all that's left is to define the function:[5]

```
(define (distance p1 p2 t)
  (< (sqrt (+ (square (− (posn-x p1) (posn-x p1)))
              (square (− (posn-y p1) (posn-y p2)))))
     t))
```

We'll leave it to you to define *square*, which consumes a number *n* and returns n^2.

Finally, we have to weave together these two termination conditions into a single one:

```
(define (game-ends? p)
  (cond
   [(on-land-or-water? p) true]
   [(overlapping? p BALLOON) true]
   [else false]))
```

and supply this as the halting condition to the World Teachpack:

```
(stop-when game-ends?)
```

Advanced Material

There is a more concise, equivalent way of writing the definition of *game-ends?* above:

```
(define (game-ends? p)
  (or (on-land-or-water? p)
      (overlapping? p BALLOON)))
```

If you don't see how this follows from the previous version, don't worry! So long as you can write the more verbose version, you'll always be okay. Indeed, sometimes it's safer to make all your reasoning completely explicit.

---

[5] If the body of this function does not make sense to you, don't panic! It uses a standard formula for the distance between two points. You should be able to look up this formula in an algebra or geometry textbook.

5.9 Version: Keep Your Eye on the Tank

Now we'll introduce the idea of limited fuel. In our simplified world, fuel isn't necessary to descend—gravity does that for you—but you do need fuel to climb. We'll assume that fuel is counted in whole-number units, and every ascension consumes one unit of fuel. When you run out of fuel, the program no longer responds to the up-arrow so you can no longer avoid either the balloon or landing on water.

In the past, we've looked at still images of the game video to determine what is changing and what isn't. For this version, we could easily place a little gauge on the screen to show the quantity of fuel left. However, we don't on purpose, to illustrate a principle.

Note

> You can't always determine what is fixed and what is changing just by looking at the image. You have to also read the problem statement carefully, and think about it in depth.

It's clear from our description that there are two things changing: the position of the plane and the quantity of fuel left. Therefore, the World must capture the current values of both of these. The fuel is best represented as a single number. However, we do need to create a new structure to represent the combination of these two.

The World

> The World is a structure representing the plane's current position and the quantity of fuel left.

Concretely, we will use this structure:

```
(define-struct world (plane fuel))
;; plane is a posn
;; fuel is a num
```

Advanced Material

> We could have also defined the World to be a structure consisting of three components: the plane's *x*-position, the plane's *y*-position, and the quantity of fuel. Why do we choose to use the representation above?

We can again look at each of the parts of the program to determine what can stay the same and what changes. Concretely, we must focus on the functions that consume and produce Worlds.

On each tick, we consume a world and compute one. The passage of time does not consume any fuel, so this code can remain unchanged, other than having to create a structure containing the current amount of fuel. Concretely:

```
(define (move-plane-on-tick w)
  (make-world (move-plane-xy-on-tick (world-plane w))
              (world-fuel w)))

(on-tick-event move-plane-on-tick)
```

The function that responds to keystrokes clearly needs to take into account how much fuel is left:

```
(define (alter-plane-y-fuel-on-key w a-key)
  (cond
   [(key=? a-key 'up)
    (cond
     [(> (world-fuel w) 0)
      (make-world (make-posn (posn-x (world-plane w))
                             (max 0
                                  (− (posn-y (world-plane w)) KEY-DISTANCE)))
                  (sub1 (world-fuel w)))]
     [else w])]  ;; if there's no fuel left, ignore the keystroke
   [(key=? a-key 'down)
    (make-world (make-posn (posn-x (world-plane w))
                           (min HEIGHT
                                (+ (posn-y (world-plane w)) KEY-DISTANCE)))
                (world-fuel w))] ;; going down doesn't consume fuel
   [else w]))
```

Be sure to test these above changes thoroughly. Updating the function that renders a scene from the world is left as an exercise for you (recall that the world has two fields; one of them corresponds to what we used to draw before, and the other isn't being drawn in the output).

5.10 Version: The Balloon Moves, Too

Until now we've left our balloon immobile. In part we did this to simplify the structure of the World: DrScheme already gave us the *posn* structure, and we ducked having to create a new structure of our design. Having to record the quantity of fuel, however, left us with no choice but to define a structure. Now that we've taken the plunge, let's proceed to make the game more realistic.

First we'll let the balloon move, as this video shows:

`http://world.cs.brown.edu/1/projects/flight-lander/v8.swf`

Obviously, the balloon's location needs to also become part of the World.

The World

> The World is a structure representing the plane's current position, the balloon's current position, and the quantity of fuel left.

Here is a Scheme structure that represents this information:

```
(define-struct world (plane balloon fuel))
;; plane is a posn
;; balloon is a posn
;; fuel is a num
```

With this definition, we obviously need to re-write all our previous definitions. Most of this is quite routine relative to what we've seen before. The only detail we haven't really specified is how the balloon is supposed to move: in what direction, at what speed, and what to do at the edges. We'll let you use your imagination for this one! (Remember that the closer the balloon is to land, the harder it is to safely land the plane.)

5.11 Version: The Balloon Moves with Variation

Suppose you decide that when the balloon reaches the edge of the scene, it should reverse direction. That's easily enough said. . . but writing the program isn't so easy. That's because we have another bit of information that is "hidden", i.e., not immediately clear from the display: namely, the *direction* in which the balloon is moving. To capture such a balloon, we need to represent not only its current position, but also its current direction of movement.

For now, we'll assume the balloon moves either directly up or down, not along diagonals. That means a balloon's direction can be represented in any number of ways:

- A boolean, where *true* represents one direction (say, up) and *false* represents the other (say, down).
- A symbol, such as `'up` for upward and `'down` for downward movement.
- The numbers $+1$ and -1 representing the two directions, but not necessarily how far to move in each direction (which would be a constant in the program).
- Positive and negative numbers representing how far to move in each direction.

For the present version any of these representations would suffice, but some of these *scale* better than others. That is, if we decided a balloon can move in more than two directions, the boolean is a bad representation. Similarly, as the number of directions grows, the set of symbols can grow unwieldy. In general, numbers are best for representing such information; as the balloon moves in more dimensions, we simply grow the number of numeric dimensions we represent.

Given that, we now consider a revised World.

The World

> The World is a structure representing the plane's current position, the balloon's current position, the balloon's direction of movement, and the quantity of fuel left.

While this is a perfectly reasonable definition, and encompasses all the things we've described until now, it does not lend itself directly to a Scheme data definition. The problem is that, as we've already discussed, the information about a balloon may grow. No matter how much it grows, however, from the World's perspective, it consists of three components:

1. the plane's information,
2. the balloon's information,
3. the fuel information.

(We haven't pinned this down precisely, but the fuel information is presumably part of the plane, not separate from it: it's the *plane*'s fuel we're interested in, not the entire global fuel supply. If so, we can apply the lesson below to combine the plane's fuel with its position, to produce an even cleaner data definition.)

Therefore, we'll use the following Scheme definition:

(*define-struct world* (*plane balloon fuel*))
;; *plane* is a *posn*
;; *balloon* is a *balloon-info*
;; *fuel* is a *num*

where

(*define-struct balloon-info* (*position direction*))
;; *position* is a *posn*
;; *direction* is a *num*

From here, we'll leave it as an exercise for you to re-define the rest of the functions in terms of these revised definitions.

Advanced Material

> As we've discussed, the plane's position and its fuel should not be two separate entities, but rather combined into a single structure representing knowledge about the plane. Rewrite the *world* data definition in terms of this, and re-define the functions accordingly.

5.12 Version: One, Two, . . . , Ninety-Nine Balloons!

Finally, there's no need to limit ourselves to only one balloon. How many is right? Two? Three? Ten? . . . Why fix any one number? As the upper photograph in figure 5.1—taken at the Albuquerque Balloon Fiesta—shows, there could be hundreds in the air. Similarly, many games have levels that become progressively harder; we could do the same, letting the number of balloons be part of what changes across levels. However, there is *conceptually* no big difference between having two balloons and five; the code to control each balloon is essentially the same. (The pilot's life is more difficult, however; the lower photograph in figure 5.1

Figure 5.1: Albuquerque Balloon Fiesta

shows planes in the air while the Fiesta's balloons are in flight.) Therefore, it would be best if we defined our program to not fix a limit to the number of balloons.

When we need to represent a *variable* number of the *same* kind of data, we naturally turn to lists. In this case, we wish to have a list of *balloon-info*s. Let's define this:

> A *list of balloons* is one of
>
> - *empty*, or
> - (*cons b l*) where
>   *b* is a *balloon-info*, and
>   *l* is a list of balloons.

Equipped with this data definition, we can now revise the definition of the World:

(*define-struct world* (*plane balloons fuel*))
;; *plane* is a *posn*
;; *balloons* is a *list of balloons*
;; *fuel* is a *num*

You should now use what you know about templates for lists of structures to rewrite the functions. Notice that you've already written the function to move *one* balloon. What's left?

1. Apply the same function to each balloon in the list.
2. Determine what to do if two balloons collide.

For now, you can avoid the latter problem by placing each balloon sufficiently spread apart along the *x*-direction. Because they move only up and down, they can't possibly collide.

Advanced Material

There are lots of extensions to this exercise:

- As the plane descends, make it accelerate with gravity. If it lands too hard, it crashes.
- Extend the balloon stuctures to record not only the *direction* of their movement but also their *speed*, so they can each move at different speeds.
- Introduce the concept of *wind*. After random periods of time, the wind blows with random speed and direction, causing the balloons to move laterally and the plane to accelerate or decelerate.
- Add a gauge to the screen showing the quantity of fuel left. Use colored rectangles: green when there's plenty of fuel left, yellow as the quantity declines, and red as it gets perilously low.
- The text above—

  > Notice that you've already written the function to move *one* balloon. [. . .] Apply the same function to each balloon in the list.

—is suggestive. If you know about *map* and similar operations, rewrite your functions in terms of these operations.

Use your imagination to make the game more sophisticated!

Part III

A Design Recipe

Chapter 6

How to Design Worlds

Now that you've seen an example, you're ready to start learning some lessons about how to design programs using World. Here's a first version of a recipe for designing World programs.

6.1 How to Identify the World's Contents

Given a set of videos showing different possible configurations for some version of a program:

1. Make sure you've covered all the interesting cases.

2. Draw yourself several successive scene pairs (before and after).

3. Circle what's different.

4. Everything that isn't subject to any change gets rendered right away.

5. Whatever's left needs a representation in the world.

6. Identify what brings about changes in the world. Is it the passage of time? User input? Both?

7. For each of the entities in the world, think hard about what kinds of data would best model it.

8. For each of the entities in the world, think also about how you are going to render it on the screen. Observe that some of the information about this entity may be fixed (e.g., if a plane can move in only one dimension, the other dimension is fixed).

9. Using the design recipe from HTDP, write test cases, design the functions, run!

6.2 How to Decompose a Problem

Sometimes, you may not be given a series of videos that make the program progressively harder; instead, you will only be given videos reflecting the very final version of the program. In this case, it's up to you to decompose the final program into smaller, more manageable steps—ideally, you'll be able to solve each step, then use your solution to build a solution for some later stage in the progression.

To begin with, use the steps given above. But once you've determined what goes in the World, don't immediately start to program! Instead, think about how best you can *build up* to the final definition World in small increments. Focus first on:

- the most important elements, and
- the simplest elements.

Keep adding elements, growing the structure, trying to reuse previous definitions for fragments of the World, composing the results of these functions into functions for the new, bigger World. Sometimes it may be easier to try one addition, remove it, try another, and then combine the two, rather than always adding to the World.

6.3 Subtleties

Keep in mind these two important subtleties when using the above recipe for designing World programs:

- Not all dynamics are visible! For instance, if you're drawing a traffic-light controller, the time since the light changed—which is used to trigger the next change—is a dynamic, even though it does not immediately appear to be part of the visible dynamics.
- When rendering, you don't necessarily want all static objects in the background and all dynamic ones in the foreground. For instance, when a plane sinks in the water, you want only the portion of the plane that is above the water visible. This lays the water (static) over the plane (dynamic).

Part IV

Structured Exercises

Chapter 7

Chicken

A chicken wants to cross the road; you must help it navigate the traffic and not get hit by any cars. We will build this program through a series of incremental exercises.

Based on the videos here:

`http://world.cs.brown.edu/1/projects/chicken/`

develop this program in the following sequence:

1. The scene contains ground on two sides of a road. The chicken responds to keystrokes to move in all four directions.
2. In addition to the chicken, there is also a car. When the car gets to the edge of the scene, it wraps around to the other side. The poor chicken gets no respite!
3. Now that we have both the chicken and the car working as desired, detect their collision and end the game if the chicken gets hit by a car. When this happens, also change the chicken image to represent a dead chicken.
4. Now we'll have multiple cars all going in a loop. All cars are of the same type.
5. Now that we have that working, let's have different kinds of cars on the road at the same time.
6. Finally, let's endow the road with two lanes, going in opposite directions. Of course, either kind of car can travel in either direction.

Chapter 8

Cowabunga!

In many parts of the world, UFOs have been abducting cows. This should not surprise you, as you are a UFO pilot who has been sent on just such a mission. As you have probably been warned, however, the base of your UFO is made of a special alloy that cannot be allowed to come in contact with the ground. Your task, therefore, is to land your UFO atop a cow without crashing on Earth... which, as you may have heard, can turn you into a participant in certain "experiments".

Based on the videos here:

`http://world.cs.brown.edu/1/projects/cowabunga/`

develop this program in the following sequence:

1. A UFO that descends, with the scene containing a single, stationary cow.
2. A descending UFO that is controlled horizontally by arrow keys.
3. Detect contact between the UFO and the cow.
4. Temporarily disable lateral movement of the UFO; instead, move the cow laterally.
5. Combine the above two, so both the UFO and the cow move laterally, in addition to the UFO descending.
6. Why stop at one cow? Support multiple cows, all moving in one direction.
7. Real cows are rarely so orderly. Each cow can move left or right, independently of the others.
8. Now detect collisions: when a cow collides with another cow, or reaches the edge of the scene, it automatically reverses direction.

Chapter 9

Spaceflight

You are piloting a rocket across the solar system. This solar system includes a sun, three orbiting planets—the middle of which is Earth, your destination—and a moon orbiting Earth. You may assume that the planets and satellite follow a circular orbit, unless you know something about the geometry of orbits and want to make your program more realistic. You succeed by reaching Earth; you fail if hit any other heavenly body, or if you run out of fuel. (Oh, did we mention you have only a limited amount of fuel?)

Based on the videos here:

`http://world.cs.brown.edu/1/projects/spaceflight/`

develop this program in the following sequence:

1. The rocket responds to keystrokes in the four principal directions.
2. The rocket can wrap around the edges of the scene.
3. The program halts if the rocket collides with the sun.
4. The sun exerts gravity on the rocket. This makes flying a little perilous, but you can also use the sun's gravity to slingshot, which will become important as you track fuel consumption.
5. Add three planets orbiting the sun.
6. Endow the planets, too, with gravitational force.
7. The program halts successfully if the rocket lands on Earth.
8. Add a moon, rotating around the Earth.
9. Add rocket-trails. A trail is a series of circles of decreasing size that record the rocket's former locations.
10. Keep track of the amount of fuel consumed by the rocket—all keystrokes consume fuel—and display the amount of fuel left. Once the rocket exhausts its fuel supply, it no longer responds to keystrokes.

Part V

Unstructured Exercises

Chapter 10

Firefighter

There are several fires on the ground. The plane has a limited capacity of water with which to extinguish the flames. Dropping water on a fire diminishes it, but may not completely put it out. Over time the fires grow larger, so if the pilot wastes too much water, the fires will eventually win. Here is a video of the program in action:

`http://world.cs.brown.edu/1/projects/firefighter/final.swf`

While developing the game, you should ignore whether the plane's image is pointing in the same direction as the plane is flying, i.e., it's okay if the plane flies "backwards". Once you have everything else working, that would be a nice detail to fix.

Another hint: You should try to raise the intensity of the fire at a much lower rate than you drop it. Otherwise the game becomes unplayable. (Note that you can use fractions and other real numbers to represent different quantities: you needn't limit yourself to integers.)