Chapter 8

Animations involving numbers

8.1 Model and view

The examples of Chapter 6 all compute the next picture in the animation from the previous picture. This turns out to be a rather restrictive way to look at things. For example, suppose we wanted an animation of a digital clock.

How would you go about computing the next picture from the previous one? Well, first you would have to look at the picture and "read" it: from the pattern of dark and light pixels in the picture, recognize each of the digits, interpret the sequence of digits as a number, add 1 to that number, and finally convert the result back into a picture. We're asking the computer to do things that humans do easily, but computers aren't very good at — things that took our eyes millions of years of evolution, and our infant selves years of training.

A more sensible approach would be to store the current time directly as a *number* (which computers are very good at), and add one second to it at every "tick". We'd still need to convert a number to a picture, but this is considerably easier than analyzing a picture and reading a number from it.

In Chapter 6, I said I was simplifying things. The truth is that an event handler doesn't necessarily take in an image; it takes in a "model", which (for now) can be *either an image or a number*. For each animation you write, you'll need to decide which of these to use, and stick to that decision throughout the animation.

You can tell DrRacket what kind of model you're using by adding another event handler to `big-bang`: either `(check-with image?)` or `(check-with number?)` depending on whether you're using images or numbers. This makes no difference if your animation is written perfectly, but if (as will almost certainly happen once or twice) you get confused, and write part of it to work on images and another part to work on numbers, a `check-with` handler allows DrRacket to give you better error messages, so you can figure out more easily what went wrong. We'll explore `check-with` more thoroughly in Chapter 14, but for now suffice it to say that `image?` is a built-in function that checks whether something is an image, and `number?` similarly checks whether something is a number.

The details are in Figure 8.1.

Figure 8.1: Event handlers for animations with image or numeric models

The big-bang function has the contract

```
; big-bang :  model (start) handler ... -> model
```

where *model* is either *number* or *image*.

tick handlers must have the contract

```
; function-name :  model (old) -> model (new)
```

They are installed with `(on-tick function-name interval)`. The *interval* is the length of time (in seconds) between clock ticks; if you leave it out, the animation will run as fast as it can.

key handlers must have the contract

```
; function-name :  model (old) key -> model (new)
```

The "key" parameter indicates what key was pressed; we'll see how to use it in Chapter 18.

They are installed with `(on-key function-name)`.

mouse handlers must have the contract

```
; function-name :  model (old)
;                  number (mouse-x) number (mouse-y) event
;                  -> model (new)
```

The first parameter is the old model; the second represents the *x coordinate*, indicating how many pixels from the left the mouse is; the third number represents the *y coordinate*, indicating how many pixels down from the top the mouse is; and the "event" tells what happened (the mouse was moved, the button was pressed or released, *etc.*); we'll see in Chapter 18 how to use this.

They are installed with `(on-mouse function-name)`.

draw handlers must have the contract

```
; function-name :  model (current) -> image
```

and are installed with `(on-draw function-name width height)`.

If you leave out the *width* and *height*, the animation window will be the size and shape of the result the first time the draw handler is called.

An especially simple draw handler, `show-it`, is predefined: it simply returns the same image it was given, and it's useful if you need to specify the width and height of the animation window but don't want to write your own draw handler.

To specify the model type , put in another event handler: either `(check-with image?)` or `(check-with number?)`, depending on whether your animation uses images or numbers as its model.

SIDEBAR:

This idea of distinguishing between a “model” of reality and the way that model appears visually is called the “model-view framework”. If you have friends or relatives who do computer programming, tell them you’re studying the model-view framework and they should know what you mean.

8.2 Design recipe for an animation, second version

Now that we know about different types of models, we’ll add to the earlier design recipe. The result is in Figure 8.2.

To illustrate it, let’s work some examples. Everything in Chapter 6 still works, using images as our models. But we can now do much more. We’ll start, as before, with an unmoving “animation”.

Worked Exercise 8.2.1 ***Develop an animation*** *whose “model” is a number, and whose “view” is a blue circle of that radius. For now, it won’t change.*

Solution: Since the animation won’t actually change, we don’t need a tick-handler, a key-handler, or a mouse-handler; we’re not sure yet whether we’ll need a draw handler.

We’ve been told to use a number as the model, so that decision is made. Since the model isn’t an image, we’ll definitely need a draw handler.

The draw handler needs to take in a number and return an image of a blue circle whose radius is that number. Let’s name the function `blue-circle-of-size`.

Contract:
```
; blue-circle-of-size :  number(radius) -> image
```

Examples:
```
(check-expect (blue-circle-of-size 2)
              (circle 2 "solid" "blue"))
(check-expect (blue-circle-of-size 37)
              (circle 37 "solid" "blue"))
```

Function skeleton:
```
(define (blue-circle-of-size radius)
  ...)
```

Inventory:
```
(define (blue-circle-of-size radius)
  ; radius      number
  ; "blue"      a string I know I’ll need
  ...)
```

Body:
```
(define (blue-circle-of-size radius)
  ; radius      number
  ; "blue"      a string I know I’ll need
  (circle radius "solid" "blue")
  )
```

Figure 8.2: Design recipe for an animation, version 2

1. **Identify what handlers you'll need** (check-with, draw, tick, mouse, and/or key).
   - You should always have a `check-with` handler.
   - If your animation needs to change at regular intervals, you'll need a tick handler.
   - If your animation needs to respond to mouse movements and clicks, you'll need a mouse handler.
   - If your animation needs to respond to keyboard typing, you'll need a key handler.
   - You'll always need a draw handler. If your model is exactly the image you want to show in the animation window, you can use `show-it`; if not, you'll need to write your own draw handler.
2. **Decide what type a "model" is** — image or number, for now — and what it "means".

   What kinds of changes do you want to make in response to events? If they're easily described by arithmetic, use a number; if they're image operations (*e.g.* `rotate-cw`), use an image. If neither, see Chapter 10.

   If you decide to use a numeric model, you still need to decide what it *means*: a rectangle's height, a circle's radius, a string's length, something's x coordinate, ...

   If you decide to use an image model, follow the recipe of Chapter 6.

   If you decide to use a number as the model, you'll definitely need to write a draw handler.
3. **Write the contracts** for the handlers, using Figure 8.1. Again, the function names are up to you, but once you've chosen a type for your model, the contracts must be exactly as in Figure 8.1.
4. **Develop each of these functions**, following the usual design recipe for each one. Don't go on to the next one until the previous one passes all of its test cases.
5. **Decide on the initial number** the model should start at.
6. **Decide on the width and height** (if the draw handler doesn't produce something of the right size).
7. **Decide on the time interval between "ticks"** (if you have a tick handler).
8. **Call** `big-bang` with the initial picture and handlers (specified using `check-with`, `on-draw`, `on-tick`, `on-mouse`, and `on-key`). See whether it works.

Test this function to make sure it works correctly by itself.

To run the animation, we need to make some more decisions: what *is* the unchanging radius of the circle? (let's try 7), and what shape and size should the animation window be? I'll pick 100 wide by 50 high, which should be plenty big enough to show a radius-7 circle. The `big-bang` call is now

```
(big-bang 7
          (check-with number?)
          (on-draw blue-circle-of-size 100 50))
```

The result should be an animation window, 100x50, containing a blue circle of radius 7. Notice that when you close the window, it returns the number 7 to the Interactions pane.

▮

Practice Exercise 8.2.2 ***Try this*** *with different numbers in place of the 100, 50, and 7.*

Try this *with a string like* `"hello"` *instead of a number as the first argument to* `big-bang`*. What error message do you get?*

Take out *the* `(check-with number?)` *handler, and try that same mistake again. What error message do you get?*

Exercise 8.2.3 ***Develop an animation*** *of a small picture of your choice on a 200x200 white background. The model should be a number representing the x coordinate of the picture's location; the y coordinate should always be 50.*

Try it with several different numbers as initial models.

The result, like Exercise 8.2.1, won't actually move, but the picture will appear at a distance from the left edge determined by the "initial model" in the `big-bang` call. And, as with Exercise 8.2.1, we'll modify this animation shortly to do more interesting things.

8.3 Animations using `add1`

Recall that for the animation of Exercise 8.2.1, we decided that the model is a number indicating the radius of the circle. How would we *change* the radius? Well, if we wanted the circle to grow over time, we could add 1 to the radius at each clock tick.

A tick handler function, for an animation with a numeric model, must always have a contract of the form

```
function-name :  number -> number
```

Conveniently enough, the `add1` function (introduced in Chapter 7) has exactly this contract, so we can use it as a tick handler without needing to write our own. The result should be a circle that grows larger every tick:

```
(big-bang 7
          (check-with number?)
          (on-draw blue-circle-of-size 100 50)
          (on-tick add1 1/2))
```

(Remember, the `1/2` means the clock should tick every half second.)

Practice Exercise 8.3.1 ***Try this.***

Exercise 8.3.2 ***Modify*** *the display of Exercise 8.3.1 so that the circle appears* centered and unmoving *in a 200x200 white background, so it appears to grow around a fixed center.*

Exercise 8.3.3 ***Modify*** *Exercise 8.2.3 so that the picture moves 1 pixel to the right every 1/10 second.*

Hint: This doesn't require writing any new functions at all, only changing the `big-bang` call.

Exercise 8.3.4 ***Develop an animation*** *of a square, initially 1x1, which grows by 1 pixel in each dimension at each clock tick, centered in a 200x200 window.*

Note: You may find that the square seems to jiggle slightly from upper-left to lower-right and back. This is because DrRacket uses integers for the positions of images; when the square has an even size, its center is exactly halfway, but when it has an odd size, its "center" for drawing purposes is half a pixel above and to the left of its actual center. Why didn't this problem show up in Exercise 8.3.1?)

Exercise 8.3.5 ***Develop an animation*** *of a rectangle, initially 2x1, which grows by 1 pixel in height and 2 in width at each clock tick, centered in a 200x200 window.*

Hint: Have the model represent only the height; put together the right picture from this information.

Exercise 8.3.6 ***Develop an animation*** *that displays a small dot or star at a location that varies over time. The x coordinate should be simply t, and the y coordinate $t^2/20$, where t is the number of ticks since the animation started.*

Hint: Write a "helper" function `y-coord` that takes in the current value of t and computes $t^2/20$; use this function in your draw handler.

Exercise 8.3.7 ***Modify*** *the animation of Exercise 8.3.6 so that the x coordinate is* $100 + 50\cos(t/10)$ *and the y coordinate* $100 + 30\sin(t/10)$.

Hint: This will show up better if you use a short tick interval, or leave it out completely so the animation runs as fast as possible.

Exercise 8.3.8 *Add the variable definitions*

```
(define XCENTER 100)
(define YCENTER 100)
(define XSCALE 50)
(define YSCALE 30)
```

to your definitions pane, and replace the formulæ in Exercise 8.3.7 with

$$x = XCENTER + XSCALE * \cos(t/10)$$

and

$$y = YCENTER + YSCALE * \sin(t/10)$$

The animation should still work exactly as before. Check that it does.

Now change the definitions of some of the variables to different numbers and run the animation again. Can you predict what will happen?

There are two other "magic numbers" still in the program: the 1/10's inside the `cos` *and* `sin` *functions. Replace these too with variables; make sure the animation works as before, then try changing these values and predict what will happen. For example, what happens when these two numbers aren't the same: when one is twice the other, or three times the other, or slightly more or less than the other?*

Exercise 8.3.9 ***Write an animation*** *that shows a blue progress bar 20 high by 120 wide, initially just an outline but filling in from left to right at 1 pixel per quarter of a second.*

Note: Depending on how you write this, your animation will probably stop changing after 30 seconds, when the progress bar reaches 100% full. In fact, it's still running, but not showing any *visible* change. We'll learn in Chapter 15 how to have it actually stop at a specified point.

8.4 Animations with other numeric functions

Of course, you can do much more interesting things to a numeric model than simply add 1 to it. For example, you can write an `add5` function that takes in a number and adds 5 to it, and use this in place of `add1` for the examples in Section 8.3, to get a blue circle that grows by 5 pixels at a time, or a digital counter that counts by 5 rather than 1.

Here's another example.

Worked Exercise 8.4.1 ***Write an animation*** *of a picture of your choice that moves right 1 pixel whenever the mouse is moved or clicked, and left 4 pixels whenever a key is typed on the keyboard.*

Solution:

What handlers do we need? The animation needs to respond to the mouse and the keyboard, so we'll need a mouse handler and a key handler. If we use a non-image model, we'll also need a draw handler.

Identify the model: The next step in designing an animation is deciding what type the "model" is, and what it "means". The only piece of information that changes in this animation is the x-coordinate of a picture, so let's say our model is a number indicating the x coordinate of the picture.

(You could try to do this animation using an image as the model, moving it to the right with `beside` and to the left with `crop-left`. Unfortunately, if it "moved" off the left edge of the screen, the picture would be reduced to nothing, and subsequent attempts to "move it right" wouldn't bring it back. Using a number as the model, we can move it off the left edge, then bring it back onto the screen, as many times as we like.)

Contracts for handlers: Draw handlers always have contract

```
; handle-draw :  model -> image
```

We've decided that for this animation, "model" means "number", so

```
; handle-draw :  number -> image
```

Key handlers always have contract

```
; handle-key :  model key -> model
```

For this animation, that becomes

```
; handle-key :  number key -> number
```

Mouse handlers always have contract

```
; handle-mouse :
  model number(mouse-x) number(mouse-y) event
  -> model
```

In our case, this becomes

```
; handle-mouse :
  number(old) number(mouse-x) number(mouse-y) event
  -> number(new)
```

Write the draw handler: Since our model is a number, we'll need a draw handler to convert it into an image. Let's use our favorite calendar picture, and (since the assignment says it moves only left and right) decide that it'll always be at a y-coordinate of, say, 50. Since it's moving only left and right, a window height of 100 should be plenty, with a window width of (say) 500.

If you did Exercises 8.2.3 and 8.3.3, you've already written a draw handler that will work for this; the only change is the size and shape of the background. Let's suppose you wrote one named `calendar-at-x`.

Write the mouse handler: We need to move right (*i.e.* increase the x coordinate) whenever there's a mouse event, but we don't care about any of the details of the event. So we'll write a function with the **contract**

```
; add1-on-mouse :  number(x)
;                  number(mouse-x) number(mouse-y)
;                  event -> number
```

(Remember, the first parameter to a mouse handler is always the current *model*, which for this animation is a number representing the *x* coordinate of the picture.) We're ignoring `mouse-x`, `mouse-y`, and `event`, so the following examples should be enough:

```
"Examples of add1-on-mouse:"
(check-expect
  (add1-on-mouse 3 29 348 "blarg") 4)
(check-expect
  (add1-on-mouse 15 503 6 "glink") 16)
```

The **skeleton** comes directly from the contract:

```
(define (add1-on-mouse x mouse-x mouse-y event)
  ...)
```

The **inventory** simply adds the four parameters:

```
(define (add1-on-mouse x mouse-x mouse-y event)
  ; x          number
  ; mouse-x    number (ignore)
  ; mouse-y    number (ignore)
  ; event      whatever (ignore)
  ...)
```

In the **body**, we merely need to add 1 to x:

```
(define (add1-on-mouse x mouse-x mouse-y event)
  ; x          number
  ; mouse-x    number (ignore)
  ; mouse-y    number (ignore)
  ; event      whatever (ignore)
  (+ x 1) ; or, if you prefer, (add1 x)
  )
```

Test this, and we can go on to the next handler.

Write the key handler: This is quite similar to the mouse handler. My solution is

```
; sub4-on-key :  number (x) key -> number

(check-expect (sub4-on-key 7 "dummy argument") 3)
(check-expect (sub4-on-key 4 "whatever") 0)

(define (sub4-on-key x key)
  ; x   number
  ; key whatever (ignore)
  (- x 4)
  )
```

Test this, and we can go on to running the animation.

Initial model: Let's start halfway across the window, at an x coordinate of 250.

Call `big-bang`: We've already made all the decisions, so all that's left is

```
(big-bang 250
          (check-with number?)
          (on-draw calendar-at-x)
          (on-mouse add1-on-mouse)
          (on-key sub4-on-key))
```

∎

Exercise 8.4.2 ***Write an animation*** *of a picture of your choice that starts at the top of the screen, and moves down by 5 pixels every half second. Use a number, not an image, as your model.*

Exercise 8.4.3 ***Write an animation*** *of a dot that doubles in size every 5 seconds, but shrinks by 4 pixels every time a key is typed on the keyboard.*

Exercise 8.4.4 ***Write an animation*** *in which a red disk — say,*

```
(circle 15 "solid" "red")
```

— alternates every second between x-coordinate 20 and x-coordinate 60 on a fixed background picture. (Try

```
(beside (circle 20 "solid" "blue") (circle 20 "solid" "green"))
```

as the background picture.)

Hint: Use `overlay/xy` or `place-image` to place the dot at a specified x coordinate, which should be the model.

Exercise 8.4.5 ***Write an animation*** *of a progress bar (as in Exercise 8.3.9) that starts at 120, and is cut in half every second thereafter: 60, 30, 15, . . .*

8.5 Randomness in animations

We can use randomness to make our animations more interesting.
Hint: Perhaps the most common mistake I've seen students make on these exercises is putting randomness into a draw handler, which is almost never a good idea. If something is supposed to be changing randomly, you probably want to *remember* that change for later events, which means it needs to affect the model. Draw handlers don't affect the model, only the way the model appears on the screen right now. **If you use `random` in a draw handler, you're probably doing something wrong!**

Exercise 8.5.1 ***Write an animation*** *of a picture of your choice that appears each second at a different x-coordinate (and the same y-coordinate), chosen from among the five choices 20, 60, 100, 140, 180.*

Hint: There are 5 choices, so you'll need to call `(random 5)` somewhere in the simulation. And your draw handler will have to convert from a 0-4 choice to one of the specified numbers; what algebraic formula would do that? Be sure to *test* your function using `check-member-of`.

Hint: You *can* do this with no model at all, putting all the work in a draw handler that ignores its input and uses `random`. But that goes crazy as soon as the user moves the mouse or types on the keyboard — a good example of why it's better to put the randomness in the tick handler, not the draw handler.

Exercise 8.5.2 ***Write an animation*** *of a picture of your choice that moves either 1 pixel left, 1 pixel right, or not at all, with equal probability, four times a second.*

Hint: How many choices are there? How can you convert them into a modification of the state of the model?

Exercise 8.5.3 ***Write an animation*** *that starts with a blank screen, and each half second adds a small dot at a completely random location — both the x coordinate and the y coordinate are chosen at random.*

Hint: Since you need to keep all the previous dots and add one more, your "model" should probably be an image rather than a number. How can you add a dot at a specified location to an existing image?

Hint: It would be especially nice if you could start the animation with *any* image, of any size or shape, and it would sprinkle dots at random all over that background, without rewriting the handlers.

8.6 Review of important words and concepts

Now that we know how to write functions with numeric values, we have a lot more flexibility in creating animations: we can have a number as the "model", change it one way on clock ticks, change it another way on mouse actions, and change it a third way on keyboard actions, as long as we write a suitable draw handler to convert from number to image.

Racket's random number generator can be used to make animations more unpredictable and interesting.

8.7 Reference

There are no new built-in functions or syntax rules in this chapter.