

CMSC 15100 Lab 6: Fireworks

Fall 2006

Complete the first 2 chapters (up to the integration with `world.ss`) in lab (optionally) with a partner. The rest you must do by yourself. Write both your name and your partner's name on the homework when you hand it in.

If you are in the 9:30am MWF section, handin your solution by emailing it to `robby@cs.uchicago.edu`. If you are in the TTh section, email your solution to `bboven@gmail.com` and `mulmuley@cs.uchicago.edu`. It must be the appropriate mailbox before lab starts in week 7.

1 Introduction

The goal of this lab is to design an interactive firework animation. Rather than using random number generation to control where the fireworks go, you will design a number of different firework animations and bind each one to a particular key on the keyboard. People can simulate a random firework animation by pounding away on the keyboard.

Note. For this lab, any function that could have been written with either `map` or `filter` *must* be written with `map` or `filter`.

Download the `star.ss` teachpack here:

<http://www.cs.uchicago.edu/~robby/courses/15100-2006-fall/lab6/star.ss>.

It defines the `star` function:

```
;; star : number number number number mode color -> image
;; to make an 'n' pointed star with inner-radius 'ir',
;; outer radius 'or', and 'mode' and 'color'
(define (star n ir or mode color) ...)

;; example
(star 20 80 200 'solid 'blue)
```



2 Exploding Stars

Consider the following data definitions:

```
;; an animation is:
;; - empty
;; - (cons frame animation)

;; a frame is:
;; (make-frame number number stage)
(define-struct frame (x y stage))

;; a stage is:
;; - (make-explosion number number color)
(define-struct explosion (iradius oradius color))
```

An animation represents a firework flying in the sky. Each animation is a list of frames, and each frame is a stage in the animation, placed at a particular point in the night sky. The *x* and *y* fields of a frame indicate where it is placed. The coordinates start from the upper-left and *x* indicates how far to the right and *y* indicates how far down. Finally, each stage is a explosion (drawn as a star) with an inner radius, an outer radius, and a color. As an example, this animation:

```
(list (make-frame 100 100 (make-explosion 10 20 'blue))
      (make-frame 140 140 (make-explosion 10 30 'blue))
      (make-frame 180 180 (make-explosion 10 40 'blue))
      (make-frame 220 220 (make-explosion 10 50 'blue)))
```

represents a star that moves and grows. It would look something like this:



To begin, design a function that builds the stages for an animation of an explosion that grows. **Note.** This function returns a list of stages, not frames.

```
;; expanding-stage : number number color → list-of-stage
;; to build the stages starting at step number 'step-num'
;; running for 'total-steps' steps.
(define (expanding-stage step-num total-steps color) ...)
```

It should produce a list whose length matches *step-num*, thus its template should be based on *step-num*. An important helper function for this function (and for others) is *interpolate* that accepts four numbers: a minimum, a maximum, a step number, and a total step count. It produces a number between the maximum and minimum, based on the step number and the total number of steps.

```
;; interpolate : number number number number -> number
(define (interpolate low high step total-steps)
  (+ (* low (/ step total-steps))
     (* high (- 1 (/ step total-steps))))))
```

Next, design a function that places a stage, always at the same coordinates.

```
;; place-stage : list-of-stage number number → animation
(define (place-stage los x y) ...)
```

and a function that renders a frame:

```
;; frame->image : frame → image
(define (frame->image frame) ...)
```

The frame->image should render the image on the (nearly) blank rectangle backdrop, where the x and y coordinates in the frame indicate how far down and to the right of the pinhole the image should be placed:

```
;; world-size : number
(define world-size 400)

;; backdrop : image
(define backdrop
  (put-pinhole
    (rectangle (+ world-size 2)
               (+ world-size 2)
               'outline
               'black)
    1 1))
```

3 User-controlled Exploding Stars

The goal of this part is to integrate the animated stars from the previous chapter into the world.ss teachpack.

This is the data definition for world that we will use.

```
;; a world is:
;; - empty
;; - (cons animation world)
```

Design the tock function:

```
;; tock : world → world
;; to remove the first element from each list in 'world',
;; and then to remove any empty lists from 'world'.
(define (tock world) ...)

;; examples as tests (to get you started; you must make more)
(equal? (tock empty) empty)
(equal? (tock (list (list (make-frame 1 2 (make-explosion 1 2 'red)))
                    (list (make-frame 1 2 (make-explosion 1 2 'red))
                          (make-frame 20 20 (make-explosion 5 5 'blue)))))
        (list (list (make-frame 20 20 (make-explosion 5 5 'blue)))))
```

The tock function must be written as the composition of two functions, one that removes the first element for each list in world and one that removes any empty lists from world.

Next, design the world->image function:

```
;; night-sky : image
(define night-sky (nw:rectangle world-size world-size 'solid 'black))
```

```
;; world->image : world -> image
;; to render the first frame in each animation in 'world',
;; and to superimpose them together on the 'night-sky'
(define (world->image world) ...)
```

To create the animation itself, you should define a key event handler that transforms a key event into a particular firework. The key event handler has this signature:

```
;; handle-key : world (char or symbol) -> world
(define (handle-key w c) ...)
```

Search in Help Desk for on-key-event for an explanation of how it is used.

This function should check to see if *c* is a character, and if the character is between #\a and #\y, using the `char<=?` function. If it is, the character can be converted to a number between 0 and 24:

```
;; c->n : character -> number
;; to convert a character between #\a and #\y to a number between 0 and 24
(define (c->n c) (- (char->integer c) (char->integer #\a)))
```

Then, that number can be converted into two numbers, one that can be used to pick a particular kind of firework, and one that can be used to pick the position of the firework:

```
;; first-num : number[0-24] -> number[0-4]
(define (first-num n) (modulo n 5))

;; second-num : number[0-24] -> number[0-4]
(define (second-num n) (quotient n 5))
```

To pick the location of the firework, use these functions (that I got from a geometry text). They correspond to five equally spaced spots around a circle of radius 100, centered on the world's image.

```
;; num->x-coordinate : integer[0-4] -> number[0-world-size]
;; to compute an x coordinate for the n-th firework position
(define (num->x-coordinate n) (scale-coordinate (cos (n->angle n))))

;; num->y-coordinate : integer[0-4] -> number[0-world-size]
;; to compute an y coordinate for the n-th firework position
(define (num->y-coordinate n) (scale-coordinate (sin (n->angle n))))

;; helper functions
(define (n->angle n) (- (* (/ n 5) 2 pi) (/ pi 2)))
(define (scale-coordinate c) (floor (+ (/ world-size 2) (* 100 c))))
```

Once the number conversion functions are set, the `handle-key` function should add the new animation it creates to `world`, making a world with one more animation in it. Each key you type should produce a slightly different firework, either in a different position, or with a different color.

Once everything is well-tested, put it all together to play with `world.ss` (you may wish to adjust the constant 0.2 a little, in order to speed up the animations, if your computer is faster than mine):

```
;; run program, run
(big-bang (+ world-size 2)
  (+ world-size 2)
  0.2
  empty)
(on-tick-event tock)
(on-redraw world->image)
(on-key-event handle-key)
```

4 Extending the stage Data Definition, i

Now that you have some working fireworks, it is time to make the animations a little richer. The goal of this chapter is to extend the stage data definition to support a flying ball going to the position of the explosion, and then adapt the rest of the program to properly handle and to exploit the change to stage. To begin, here is the new definition for stage:

```
;; a stage is either:
;; - (make-explosion number number color)
;; - (make-ball color)

(define-struct explosion (iradius oradius color))
(define-struct ball (color))
```

It has one more alternative, meant to represent a little ball with a color. This is then going to be used to show the (unexploded) firework flying across the sky.

Next, design the function shooting-animation:

```
(define start-x (/ world-size 2))
(define start-y world-size)

;; shooting-animation : number number number number -> animation
;; to produce a moving ball animation that starts at (start-x, start-y)
;; and goes to (end-x, end-y) in 'total-steps', but is currently on 'step'.
(define (shooting-animation step total-steps end-x end-y) ...)
```

Once you have finished shooting-animation, extend all of the existing functions that consume stage or frame values to cope with the new kind of stage. Note that the data definition for stage changed in a specific way. Accordingly, your templates should only change in a specific way, and thus you know how to change your functions.

Finally, extend the handle-key function so that all of the fireworks begin with a ball that moves to the appropriate position before the explosion starts.

5 Extending the stage Data Definition, ii

Design at least 2 more new kinds of stages, and adapt your program so that they are used in the animations. Here are some ideas of new kinds of stages:

- explosions that consist of multiple, overlaid stars with different numbers of points and colors;
- explosions that have several, evenly spaced stars, all growing;
- a series of falling dots, to be part of an animation that comes after an explosion.