Stage 2 Design

Daniel Brady

June 12, 2014

Problem Statement

Determine a suitable representation for the world and for inhabitants of that world. Then set up a world such that the user can change the direction of the snake using the keyboard, and in which the snake interacts with the other worldly inhabitants.

1 Setting up our data structures

In Stage 1, we defined a series of data structures that would represent our world, and at that point, our world consisted of just a snake.

> (struct snake (dir bod)
    #:transparent
    #:constructor-name make-snake)
> (struct seg (x y sprite)
    #:transparent
    #:constructor-name make-seg)
> (define THE-SNAKE (make-snake "left" (list (make-seg 13 69 HEAD-IMG)
                                               (make-seg 43 69 BODY-IMG)
                                               ...)))

We are now ready to move on to a more complex world. Our new world will have not just a snake, but also an environment. This environment could contain any manner of things (perhaps even other snakes!), and will be represented by a list of those things, just like the body of our snake is just a list of segments.

> (struct world (snake env)
    #:transparent
    #:constructor-name make-world)
The most basic environment a world can have is an empty one.

> (define THE-WORLD (make-world THE-SNAKE '()))

But at some point, that environment will have something in it. In a game of Snake, the environment is usually populated with food for the snake to chase after.

> (struct food (x y sprite)
    #:transparent
    #:constructor-name make-food)

Now, wait a minute: take a look at the food and seg structures. Do they seem at all similar to you?

> (struct seg (x y sprite)
    #:transparent
    #:constructor-name make-seg)

> (struct food (x y sprite)
    #:transparent
    #:constructor-name make-food)

Why, they are identical! They are the exact same data structure, except that we have given them different names. We could take advantage of this fact, and instead create a data type that can be used to represent all food’s and seg’s and other worldly (not other-worldly) objects.

> (struct fancy-point (x y sprite)
    #:transparent
    #:constructor-name make-fancy-point)

I think this makes sense, because the only real difference between the objects in the world, at least at this point in the design,¹ is the image being used to show the user that they exist.

> (define THE-SNAKE (make-snake "left" (list (make-fancy-point 13 69 HEAD-IMG)
                                           (make-fancy-point 43 69 BODY-IMG)
                                           ...)))
> (define THE-WORLD (make-world THE-SNAKE (list (make-fancy-point 456 293 FOOD-IMG)
                                           (make-fancy-point ....))))
2 Back to the drawing board

Now that we’ve created representations for our data, we can move on to figuring out how they will be drawn.

In Stage 1, we designed a function that rendered a snake (which was the entire world at that time). This function will be useful here, because even though we’ve changed the world, we still need to render a snake; but we also need to render the environment, as well, like in Figure 1. We need to modify our old function, though, just a little bit, to use our updated data types.

---

1Later on, perhaps, we might decide that different objects in the world will have other attributes, i.e. a power-up or obstacle.
> (define snake->img
   (lambda (snake-segments bg-img)
      (foldl (lambda (seg base-img)
              (place-image (fancy-point-sprite seg) ; (seg-sprite seg)
                          (fancy-point-x seg) ; (seg-x seg)
                          (fancy-point-y seg) ; (seg-y seg)
                          base-img))
            bg-img
            snake-segments)))

What we need to figure out how to draw now is the environment, that list of objects in the world. Right now, our environment contains only food, which makes this function easy.

> (define env->img
   (lambda (env bg-img)
      (foldl (lambda (food base-img)
              (place-image (fancy-point-sprite food)
                          (fancy-point-x food)
                          (fancy-point-y food)
                          base-img))
            bg-img
            env)))

Hmmm...again, am I the only one noticing the similarities here? Look! These two functions are EXACTLY THE SAME FUNCTION! All that’s different are the variable names and the names of the functions themselves, just like what happened with our data structures. So once again, let’s do some clean up.

> ; Takes a list of fancy-points and composes them into a single image.
 (define lofp->img
  (lambda (lofp bg-img)
    (foldl (lambda (fp base-img)
            (place-image (fancy-point-sprite fp)
                        (fancy-point-x fp)
                        (fancy-point-y fp)
                        base-img))
           bg-img
           lofp)))

We can now use this function to render any list of fancy-points, like, oh, say, the body of a snake or the environment of a world. Or both.
> (define world->img
  (lambda (world)
    (let ([snake (world-snake world)]
          [env (world-env world)])
      ; Draw the snake on top of the environment.
      (lofp->img (snake-bod snake)
        ; Draw the environment on top of an empty scene/background image.
        (lofp->img env (empty-scene WORLD-W WORLD-H))))))

3 Making the world go ‘round

The next step is to take a look at the function we wrote to update the world back in Stage 1, and see what modifications need to be made. Take a moment to think about what will be different in Stage 2, now that we have food in our world and not just a snake.

> (define update-world
  (lambda (world)
    (let* ([body (snake-bod world)]
           [head (first body)]
           [dir (snake-dir world)]
           [new-body (cons (make-seg (seg-x head)
                                       (seg-y head)
                                       BODY-IMG)
                        (seg-sprite head))
                        new-body)]
      (cond
         [(equal? dir "up")
          (make-snake dir
                       (cons (make-seg (modulo (seg-x head) WORLD-W)
                                     (modulo (- (seg-y head) BODY-H 5) WORLD-H)
                                     (seg-sprite head))
                              new-body))]
         ...))))

What exactly does this function do? Or, rather, what is it \textit{supposed} to do? \textit{update-world} is supposed to take a \textit{world}, and give back a \textit{world}; the \textit{world} it gives back may be the same, or may be different in some way, even entirely different, just as long as it is a \textit{world}.

The function we wrote in Stage 1 assumes that the \textit{world} it receives is actually just a \textit{snake}. That obviously can’t stay the same in the function we use at this stage. But we know that we will still need to do everything that this function does; we’ll just be needing to do a bit more. So let’s keep it around as a helper, but give it a new name so that it is precisely clear what kind of data it expects to work with.
> (define update-snake
  (lambda (snake)
    (let* ([body (snake-bod snake)]
           [head (first body)]
           [dir (snake-dir snake)]
           [new-body (cons (make-seg (seg-x head)
                                        (seg-y head)
                                        BODY-IMG)
                        (rest (reverse (rest (reverse body))))])
      (cond
       [(equal? dir "up")
        (make-snake dir
                     (cons (make-seg (modulo (seg-x head) WORLD-W)
                                 (modulo (- (seg-y head) BODY-H 5) WORLD-H)
                                 (seg-sprite head))
                           new-body))]
       ...)))))

Now let’s turn our attention to the function we’ll be needing for this stage in our game design. Now that we have food in our environment, we have a bit of added complexity to our world. Thankfully, though, we don’t have to worry about the foods moving around like we did our snake; in fact, we almost don’t have to worry about them at all! The only time we actually care about them is when they happen to be occupying the same location as the head of our snake; when that happens, we want the snake to ‘eat’ them.

In other words, whenever the head of the snake shares the same location as one of the foods in the environment, that food should be removed from the environment.

> (define update-env
  (lambda (env)
    ...)))

Once we have that implemented, we’re ready to write the newest version of update-world. Remember, it’s a function that takes a world and gives one back.

> (define update-world
  (lambda (world)
    (make-world (update-snake (world-snake world))
                (update-env (world-env world))))))

...and that is as far as we’ve gotten! Tomorrow, we will wrap things up here in Stage 2 so you can have the weekend to prepare for the third and final stage.

Happy coding!