Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

Laboratory: Building Images by Iterating Over Positions

Summary: In this laboratory, you will explore techniques for building and modifying images by iterating over the positions in an image.

Contents:

Reference:

- (position.new *col row*) - create a new position value for (*col*,*row*).
- (position.col *pos*) - get the column from a position value.
- (position.row *pos*) - get the row from a position value.

- (region.compute-pixels! *image first-row first-col last-row last-col compute-color*) - compute new colors for all the pixels in the region from (*first-row*,*first-col*) to (*last-row*,*last-col*) in *image*.

Preparation

Create and show a new 200x200 image called `canvas`.

Exercises

Exercise 1: Drawing Rectangles

As you learned in the reading, one of the simplest ways to use `region.compute-pixels!` is to create a rectangle of a uniform color. For example, we might add a small square near the middle of `canvas` with

```
(define rect-color (cname->rgb "hot pink"))
(region.compute-pixels! canvas 90 90 110 110 (lambda (pos) rect-color))
```

a. Confirm that these instructions work as advertised.

b. What do you expect the following instructions to do? What do you expect to have happen to the pixels that were in the first rectangle?

```
(define rect-color (cname->rgb "firebrick"))
(region.compute-pixels! canvas 95 100 120 120 (lambda (pos) rect-color))
```

c. Check your answer experimentally.

d. Consider the following instructions. What do you expect to have happen when you execute them?

```
(define rect-color (rgb.new 255 255 255))
(region.compute-pixels! canvas
                        0 0
                        (image.width canvas) (image.height canvas)
                        (lambda (pos) rect-color))
```

e. Check your answer experimentally and then read the notes on this problem.

f. Using a similar instruction, make the entire canvas black.

g. Write instructions to draw a blue rectangle in the upper-left corner of `canvas`, a green rectangle in the upper-right corner, a yellow rectangle in the lower-right corner, and a red rectangle in the lower-left corner. You can choose the size of the rectangles.

Exercise 2: Ranges

a. The reading claims that the following instructions will provide a range of intensities of blue. Check to confirm that the claim is correct.

```
(define blues (image.new 17 1))
(image.show blues)
(region.compute-pixels! blues 0 0 16 0
                        (lambda (pos) (rgb.new 0 0 (* 16 (position.col pos)))))
```

b. Use similar code to create a range of intensities of green in a 17x17 image, keeping each column the same intensity of green.

c. Use similar code to create a range of intensities of red bottom (instead of from left to right) in a 17x17 image. In this case, make the intensities vary from top to bottom, instead of from left to right. That is, each row should have the same intensity of red.

d. Sometimes we don't want our color ranges to fill the entire image. Write instructions to put the previous red blend in the square that starts at 60,60 and ends at 76,76 in `canvas`.

Exercise 3: Two-Dimensional Color Ranges

In the previous problem, you made images that showed color intensities ranging over a single dimension, either horizontally or vertically. However, we can certainly vary colors in both dimensions.

Write instructions to create a 17x17 image that shows a range of intensities of blue that vary over all the pixels. In particular, make the blue component proportional to the sum of the row and column number.

Optional: Fill in the image again, making the blue component proportional to the product of the row and column number.

Exercise 4: Generalizing Color Ranges

As you should recall from recent homework assignments, one thing that a good programmer does is generalize her code, so that it works in a variety of situations. The color range code we've written so far is fairly specific to the image size (and even the image placement).

For example, let's consider the computation of the blue pixels in a 17x17 image. The core part of that code is likely to read as follows:

```
(define blues (image.new 17 17))
(region.compute-pixels! blues 0 0 16 16
                        (lambda (pos) (rgb.new 0 0 (* 16 (position.col pos)))))
```

The number 16 appears three times in this code. The first time it represents the final column of the image. The second time, it represents the final row of the image. The third time, it represents 256/16.

a. Rewrite these instructions so that they provide a blue range that fills `blues`, but will work no matter what width and height `blues` happens to have. For example, if I want a shorter, wider, image, I might write

```
(define blues (image.new 65 8))
your code
```

b. Sometimes we want to draw color ranges in the middle of an image. Suppose we've defined the values `range-left`, `range-top`, `range-width`, and `range-height` and want to draw a blue color range within the specified area of `canvas`. Write code that does so. For example, one might write

```
(define range-left 80)
(define range-top 50)
(define range-width 17)
(define range-height 8)
your code
```

c. Rewrite your code from part b as a function of five parameters.

```
(define draw-blue-range
  (lambda (image range-left range-top range-width range-height)
     your modified code))
```

You should be able to plug in the code from part b, substituting `image` for `canvas`

d. Test you code by drawing a variety of blue color ranges. Here are a few to get you started.

```
(draw-blue-range canvas 0 0 17 2)
(draw-blue-range canvas 90 0 20 9)
```

Exercise 5: Color Blends

a. The reading suggested that the following code will provide a blend from red to blue. Check that claim.

```
(define red-to-blue (image.new 129 16))
(image.show red-to-blue)
(region.compute-pixels! red-to-blue
                        0 0
                        128 15
                        (lambda (pos) (rgb.new (* 2 (- 128 (position.col pos)))
                                               0
                                               (* 2 (position.col pos)))))
```

b. Write similar code to provide a blend from yellow to red.

c. Put this blend in `canvas`, using the rectangle from 10,120 to 138,136.

Exercise 6: Bordering Blends

At the end of some of the previous exercises, you created interestingly colored rectangles in the middle of `canvas`. Sometimes, to make such rectangles stand out, we would like a fixed-color border around the rectangles.

a. Describe at least two ways to create such borders. After doing so, read the notes on this problem.

b. Pick one of those two ways, and draw the yellow-red blend from the previous problem in the same location, but with a four-pixel-wide black border.

Exercise 7: More Complex Computations

You may recall a more complex computation of colors from the reading, one that used `sin` to determine values. Here's a simplified version of that code:

```
(define what-am-i (image.new 40 50))
(region.compute-pixels! what-am-i
                        0 0 39 49
                        (lambda (pos)
                        (rgb.new 0
                                 (* 255 (abs (sin (* pi .025 (position.col pos)))))
                                 0)))
```

a. What range of values does `(* pi .025 (position.col pos))` compute?

b. What range of values does `(sin _)` compute?

c. What range of values does `(abs (sin _))` compute?

d. What range of values does `(* 255 (abs (sin _)))` compute?

e. Given that analysis, what kind of image do you expect the preceding code to draw?

f. Check your answer experimentally.

g. The reading had a somewhat more complex set of instructions.

```
(define what-am-i (image.new 40 50))
(region.compute-pixels! what-am-i
                        0 0 39 49
                        (lambda (pos)
                        (rgb.new 0
                                  (* 255 (abs (sin (* pi .025 (position.col pos)))))
                                  (* 255 (abs (sin (* pi .020 (position.row pos))))))))
```

Explain why there is a .020 rather than a .025 in the computation of the blue component.

h. Predict the appearance of the computed image.

i. Check your answer experimentally.

Exercise 8: Expanding Images

Consider the final image from the previous exercise. Suppose we wanted to make a similar, but larger, version of the image. To do so, we'd certainly need to change the width and height of the image and the ending column and row.

Do we need to change the `.025` and `.020`? We can, but it depends on how we define "similar".

If you change `.025` and `.020`, you might scale them similarly to how you scale the image. For example, if we double the width and height of the image, we might write

```
(define what-am-i (image.new 80 100))
(region.compute-pixels! what-am-i
                        0 0 79 99
                        (lambda (pos)
                        (rgb.new 0
                                 (* 255 (abs (sin (* pi .0125 (position.col pos)))))
                                 (* 255 (abs (sin (* pi .01 (position.row pos))))))))
```

a. What do you expect to have happen if you use the preceding code to create a larger, similar, image?

b. Check your answer experimentally.

c. What do you expect to have happen if you restore the .025 and .020 to the preceding code?

d. Check your answer experimentally.

Explorations

Choose either, both, or neither of these explorations.

Exploration 1: Simulating Depth

As Professor Kluber suggested when he visited, one way we can simulate depth in an image is by overlaying one thing over another.

a. Create an image you find aesthetically pleasing by overlaying fixed-color rectangles on top of each other.

b. Create an image you find aesthetically pleasing by overlaying computed-color rectangles on top of each other.

Exploration 2: Complex Color Computations Continued

In the final exercises in this lab, you used trigonometric functions to compute color values. Clearly, we can use these functions, and others, in a variety of ways. For example, we need not linearly scale the row and column, or can scale them in a way to get different ranges. Similarly, we can choose functions other than `sin` to compute results. In the end, all that we care is that we end up with each component value in the range 0 .. 255.

Experiment with a variety of functions and multipliers to find a combination that you find pleasing.

For example, consider the original computation of the green component

```
(* 255 (abs (sin (* pi .025 (position.col pos)))))
```

We might replace the `0.25` with another number. We might replace the `sin` with `cos`. We might cube the result of that multiplication with

```
(expt (* .025 (position.col pos)) 3)
```

We might even square the result of the sine computation.

Notes

Notes on Exercise 1: Drawing Rectangles

While the obvious answer to part d is “it makes the image white”, there is a subtle bug in the code, which should cause it to report an error, rather than fill the image with the color white.

We include this intentional bug, as we have in the past, to remind you that you need not just consider what code is *supposed* to do; you should also make sure that it is logically correct.

Notes on Exercise 6: Bordering Blends

One technique for putting a border around a rectangular part of the image is to draw a slightly larger rectangle *before* you draw that part of the image. For example, if our blend is from 10,20 to 25,35, we might first draw a rectangle from 8,18 to 27,37.

Another technique is to draw four thin rectangles, one above, one to the left, one to the right, and one to the bottom. For example, if our blend is from 10,20 to 25,35, we might draw a two-pixel border with

```
(define bgcolor (rgb.new 0 0 0))
(region.compute-pixels! canvas  8 18 27 19 (lambda (pos) bgcolor))
(region.compute-pixels! canvas  8 18  9 37 (lambda (pos) bgcolor))
(region.compute-pixels! canvas 26 18 27 37 (lambda (pos) bgcolor))
(region.compute-pixels! canvas  8 36 27 37 (lambda (pos) bgcolor))
```

The second technique is a bit more complicated to write, but has the advantage that we can add the border *after* we’ve drawn the image.

Copyright © 2007 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/2.5/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.