

## Building Images by Iterating Over Positions

**Summary:** By looking at a procedure that permits us to choose the color at each position in a region of the image, we expand our repertoire of procedures that permit us to build or modify images.

### Contents:

- Introduction
- The key procedure: `region.compute-pixels!`
- Detour: Positions
- Color Ranges, Color Blends, and More
- Simulating `image.map!`
- Computing New Colors Based on Color and Position
- Quick Reference

## Introduction

In our initial explorations of DrFu, we've found ways to create images by setting individual pixels and ways to modify existing images by transforming all the pixels, each with the same technique. Clearly, if we are going to build images, we need a richer set of techniques.

One reasonable strategy is to work with a procedure that allows us to set the colors of an image based on the position of the pixel. Why is that useful? As we saw in the initial tasks of drawing smiley faces, if you can write algorithms that set the color at particular positions, then you can draw circles and a host of other shapes. We can simulate such algorithms by looking at each position and checking if it meets certain criteria.

Of course, we will often need to check criteria using conditionals and we have not yet studied conditionals. Nonetheless, we can use the strategy of "compute a color for each pixel based on the pixel's location" in a number of other ways. As we will see here, it makes it easy to draw rectangles, to add color washes, and even to create variants of the `image.map!` procedure.

## The key procedure: `region.compute-pixels!`

DrFu provides one basic procedure for computing pixels based on position, `region.compute-pixels!`. Instead of setting every pixel in the image, `region.compute-pixels!` works only with pixels in a rectangular region. Why just a region? Efficiency. For a big image in which we only want to set colors in a small area, we would prefer not to have to look at (and do computations for) the rest of the image. By limiting the region, we can also more easily create simple forms. Why a rectangle? It's easier to specify.

This `region.compute-pixels!` procedure takes a few more parameters than procedures you've used in the past, six, to be specific.

- *image*, the image which we will modify;
- *first-col*, the first column of the region to modify;
- *first-row*, the first row of the region to modify;
- *last-col*, the last column of the region to modify;
- *last-row*, the last row of the region to modify; and
- *compute-pixel*, a function that, given a position, computes a color for the pixel at that position.

For example, the following computes a new color at each position in the rectangular grid from row 5, column 2 to row 10, column 15.

```
(region.compute-pixels! canvas 5 2 10 15 ____)
```

The `compute-pixel` procedure should have the form

```
(lambda (position) expression-to-compute-color)
```

What can we fill in for the function? That's the subject of the rest of this reading. However, there's one simple technique we can do with what we know so far: We can ignore the position and simply return a fixed color. Here, we set the preceding rectangular region to red.

```
(region.compute-pixels! canvas 5 2 10 15 (lambda (pos) (rgb.new 255 0 0)))
```

## Detour: Positions

If we are to write procedures that compute colors from positions, we need to know a bit about how DrFu represents positions. For the purposes of this assignment, there are only two basic procedures you need to work with positions:

- `(position.col pos)`, which, given a position value, returns the column component of that position.
- `(position.row pos)`, which, given a position value, returns the row component of that position.

There's also a constructor for positions. As you might expect, `(position.new col row)` creates a new position value for `(col,row)`. While we won't use `position.new` in this work, we'll find uses for it in the future.

## Color Ranges, Color Blends, and More

At times, we want to make an image that provides a range of colors, such as various kinds of blue or various kinds of grey. For example, in the laboratory on numeric values, we used a 17x1 image each of which had a different intensity of blue. We can build such a collection more efficiently with `region.compute-pixels!` by building a 17x1 image and setting the blue component of each pixel to 16 times the column number.

```
(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))))))
```

In a recent assignment, we dealt with an extension of color ranges, which we called *blends*. We can extend the previous example to express blends more concisely. For example, here is a 128-step blend from red to blue.

```
(define red-to-blue (image.new 129 1))
(image.show red-to-blue)
(region.compute-pixels! red-to-blue
  0 0
  128 0
  (lambda (pos) (rgb.new (* 2 (- 128 (position.col pos)))
    0
    (* 2 (position.col pos))))))
```

Note that we can use this technique to make larger images, too. (Perhaps we can soon stop using the zoom button to figure out what's going on.)

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

This blend is somewhat limited, as it goes from pure red to pure blue, and the way in which it works is somewhat concealed in the particular numbers used. You will have an opportunity to generalize this solution in the near future.

Of course, we can do more than just blend colors. We can do almost any computation that creates a color between 0 and 255. Here's an interesting one. Can you predict what it will do?

```
(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))))))))))
```

In the corresponding lab, you'll have an opportunity to experiment with similar computed colors.

## Simulating `image.map!`

In the computations done in the previous section, we determined a color for a pixel based only on the position of the pixel. In our earlier experiments at manipulating a whole image, we determined the color for each pixel based on the previous version of that color. The activities seem similar enough. Can we use

`region.compute-pixels!` to *modify* pixel values, and not just compute new pixel values? Certainly. We just ask for the pixel using the row and column. Here, we complement a small section of an image.

```
(region.compute-pixels! picture
  100 100
  120 120
  (lambda (pos)
    (rgb.complement (image.get-pixel picture (position.col pos) (position.row pos)))))
```

If we can simulate `image.map!` with `region.compute-pixels!`, why do we both to have a separate `image.map!` procedure? Because, as you will see, `image.map!` is significantly faster than its simulation with the `region.compute-pixels!` procedure. There are times in which you'll be willing to sacrifice the speed for new capabilities, such as when you want to transform only a small portion of the image.

*Warning! The fast implementation of `region.compute-pixels!` does not work in conjunction with the `image.get-pixel!` procedure. If you wish to use the two together, you must use the `region.compute-pixels-slow!` procedure.*

## Computing New Colors Based on Color and Position

Upon observing that `region.compute-pixels!` could simulate `image.pixel-map!`, only more slowly, we noted that you use `region.compute-pixels!` when One such new capability is the ability to compute new colors based on both color and position.

Here's a simple procedure that, given a color, a position, and the width and height of an image, computes a new version of that color at that position by reducing the red component proportional to its distance to the left of the middle (or increasing it proportional to its distance to the right of the middle) and reducing the blue component proportionally above the center and increasing proportionally below the center.

```
(define sample-combine
  (lambda (color pos width height)
    (rgb.new (* (+ 0.5 (* (/ (position.col pos) width))) (rgb.red color))
             (rgb.green color)
             (* (+ 0.5 (* (/ (position.row pos) height))) (rgb.blue color)))))
```

We can apply it to a region of our favorite picture with

```
(region.compute-pixels! picture
  100 100
  120 120
  (lambda (pos)
    (sample-combine (image.get-pixel picture (position.col pos) (position.row pos))
                    pos
                    (image.width picture)
                    (image.height picture))))
```

## Quick 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*.
- 

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.