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

Laboratory: Anonymous Color Transformations

Summary: In this laboratory, you will practice writing procedures in Scheme, primarily by writing anonymous functions that you use to filter images.

Contents:

Reference:

The syntax for an anonymous function of one paramter is as follows:

`(lambda (`*`parameter`*`) `*`expression`*`)`

Preparation

a. Load a moderate-sized image (no more than about 250x250) and name the loaded image `source`.

b. Create and show a new 3x3 image and name it `canvas`.

c. Zoom in on `canvas`.

d. Fill in the pixels of `canvas` as follows. Note that you may want to open this lab in a Web browser and cut and paste.

```
(image.set-pixel! canvas 0 0 (rgb.new 127 0 0))
(image.set-pixel! canvas 1 0 (rgb.new 127 0 127))
(image.set-pixel! canvas 2 0 (rgb.new 0 0 127))
(image.set-pixel! canvas 0 1 (rgb.new 127 127 0))
(image.set-pixel! canvas 1 1 (rgb.new 0 127 127))
(image.set-pixel! canvas 2 1 (rgb.new 255 255 255))
(image.set-pixel! canvas 0 2 (rgb.new 0 127 0))
(image.set-pixel! canvas 1 2 (rgb.new 0 0 0))
(image.set-pixel! canvas 2 2 (rgb.new 127 127 127))
```

Exercises

Exercise 1: Removing Color Components

As you may recall from the reading, we can remove the red component of all the pixels an image by mapping an anonymous function that uses 0 for the red component and maintains the green and blue components. That function might looks something like the following:

```
(lambda (color) (rgb.new 0 (rgb.green color) (rgb.blue color)))
```

a. Using `image.map` and this function, build a new image from `canvas` and see if it has the expected effect.

b. Using `image.map` and this function, build a new image from `source` and see if it has the expected effect.

c. Using `image.map` and an anonymous function you define, build new images from `canvas` and `source` that delete both the green and the blue components from each pixel.

d. Using `image.map` and an anonymous function you define, build new images from `canvas` and `source` that set the blue component to 255 (and preserve the red and green components).

Exercise 2: New Techniques for Changing Color Components

We have seen two ways to change the components of a color: We can set them to particular values (as in the previous exercise) or we can add or subtract fixed amounts from them. It is certainly possible to do more. For example, we could multiply the components by some value, rather than add to the components.

For example, here's an anonymous procedure that doubles the green component.

```
(lambda (color) (rgb.new (rgb.red color) (min 255 (* 2 (rgb.green color))) (rgb.blue color)))
```

a. Confirm that the procedure works as advertised.

b. Why do you think there is a `min` in the computation of the green component of the new color? After you have discussed your answer to this question with a nearby student, you may want to read the notes on this problem.

c. Using `image.map` and an anonymous procedure of your own design, build new images by halving the red, blue, and green components of each pixel in `source` and `canvas`.

d. In both the initial example and the procedure you wrote, we have used whole numbers to scale the components. What do you expect to have happen if we use some other number, such as 1.2 or 0.8, to scale the components?

e. Check your answer experimentally.

Exercise 3: Greyscale

One common image transformation is to that of converting a color image to greyscale. We know from past work that a grey has equal red, green, and blue components. Hence, our problem is to figure out what the identical component should be.

a. The simplest way to compute the common component value is to simply find the average of the three components:

```
(/ (+ (rgb.red color) (rgb.green color) (rgb.blue color)) 3)
```

Write a call to `image.map`, using an anonymous procedure of your design, that builds a greyscale version of `source` using this averaging technique.

b. Most careful computer graphics programmers have learned that the different colors have different apparent brightness. Hence, we tend to see a pure green as lighter than a pure red and a pure red as lighter than a pure blue. Careful analysis suggests that 30% of the red value, 59% of the green, and 11% of the blue provides the appropraite combination.

Write a call to `image.map`, using an anonymous procedure of your design, that builds a greyscale version of `source` using this weighted averaging technique.

Exercise 4: Enhancing Colors

In a preceding exercise, you found that you could make colors much lighter by multiplying components by some factor and much darker by dividing compnents by some factor. In the past, you found that `rgb.darker` subtracted values from every component and `rgb.lighter` added values to each component.

But what if you want to make the values more extreme, not just darker or lighter. In particular, what if you want to make the dark colors (or at least the components that make them dark) darker, and light colors lighter? Traditionally, we think about solving this problem with conditionals. That is, we'd like to write code equivalent to "if the blue component is less than 128, decrease it, otherwise increase it". But it is possible to do that computation without using explicit conditionals. How? That's your challenge in this exercise. But here's a hint: Rather than changing the value of the component, change the difference between that value and some other value (say, 127 or 128).

a. Write an instruction using `image.transform-pixel!` and an anonymous function of your choice that transforms pixel (0,0) of `canvas` so that the blue component increases if the component is more then 127 and decreases by if the component is less than 128.

We've asked you to transform a pixel, rather than a whole image, because transforming a pixel makes it easier to do testing. For example, we might see what happens with a dark blue color with three instructions.

```
(image.set-pixel! canvas 0 0 (rgb.new 0 0 100))
(image.transform-pixel! canvas 0 0 (lambda (color) (rgb.new (rgb.red color) (rgb.green color) __)))
(rgb->string (image.get-pixel canvas 0 0))
```

By changing the initial color, you can then ensure that your instruction works correctly on a wide variety of colors (or at least three: one in which the blue component is less than 128, one in which it is 128, and one in which it is more than 128).

b. Once you are confident that your procedure works, build new images by applying it to both `source` and `canvas`.

c. Use a similar technique to write an instruction that builds a new image by converting the red, green, and blue component of each pixel to either 0 or 255, whichever it is closest to.

Exercise 5: Black and White

Using the techniques from the previous exercises, write a call to `image.map` that converts your images to black and white.

Exercise 6: Flattening an Image

In many applications, such as traditional offset printing, it is difficult, if not impossible, to get the full 255 different red, green, and blue components. In fact, in most cases, we can get no more than four or eight different intensities of the color contribution. Hence, it is common practice to *flatten* an image, say, by ensuring that each component is either 0 or 1 less than a multiple of 32 (8 different levels) or 64 (4 different levels). Some designers have found this transformation useful as it gives a nice, cartoony, version of an image.

How do we do this conversion? Well, you could certainly use a conditional, if you knew conditionals. However, even if you knew conditionals, the code would be long. Instead, we should work out a formula, like the one that we used a few problems previous.

In this case, we can transform a component by dividing by, say, 32, rounding, and then multiplying by 32. In Scheme, our code for the red component might then look something like

```
(- (* 64 (round (/ (rgb.red coor) 64))) 1)
```

Using that idea, write an instruction to build a flattened version of `canvas` and `source`.

Exercise 7: Colors as Numbers

As most of you have noted, RGB colors are represented in DrFu as numbers. (We don't yet know what those numbers mean.) Hence, we can also write transformations that work directly with the numbers, instead of with the individual components.

a. Try each of the following (perhaps multiple times)

```
(image.show (image.map (lambda (color) (* 2 color)) source))
(image.show (image.map (lambda (color) (* 0.5 color)) source))
(image.show (image.map (lambda (color) (+ 500 color)) source))
```

b. Write a few mathematical transformations of your own. See if you can find a particularly appealing one. If so, share it with the class and we'll see if works well on other images.

Notes

Notes on Exercise 2: New Techniques for Changing Color Components

Why do we have the `min` in the computation of the green component? That is, why do we write `(min 255 (* 2 (rgb.green color)))` and not just `(* 2 (rgb.green color))`. Well, if the green component is more than 127, 2 times the green component will be more than 255, and we're not supposed to have a component whose value is more than 255.

What does the `min` do? Well, if the computed green component is less than 255, `(min 255 computed-value)` will give the computed component. If the computed component is more than 255, `(min 255 computed-value)` will give 255. Hence, we have ensured that the component is never more than 255, and, whenever possible, is the computed value.

When you do computations of color components and those computations may result in values less than 0, you should use `(max 0 computed-value)`.

Note: The current implementation of `rgb.new` reduces too-large components to 255 and increments too-small components to 0. Nonetheless, it is certainly possible the the implementation may change. More importantly, is is good practice to make sure that you provide reasonable numbers to the procedures you call.
