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

Transforming RGB Colors

Summary: We examine the building blocks of one of the common kinds of algorithms used for RGB colors: Generating new colors from existing colors.

Contents:

Introduction

We have just started to learn about images and colors, and so the operations we might do on images and colors are somewhat basic. How will we expand what we can do, and what we can write? In part, we will learn new Scheme techniques, applicable not just to image computation, but to any computation. In part, we will learn new functions in the DrFu library that support more complex image computations. In part, we will write our own more complex functions.

So, what kinds of things might we do with images? One common algorithmic approach to images is the construction of *filters*, algorithms that systematically convert one image to another image. Complex filters can do a wide variety of things to an image, from making it look like the work of an impressionist painter to making it look like the image has been painted onto a sphere. However, it is possible to write simple filters with not much more Scheme than you know already.

Over the next few readings and labs we will consider filters that are constructed by transforming each color in an image using an algorithm that converts one RGB color to another. In the first RGB lab, you began to think about such algorithms as you computed the pseudo-complement of an RGB color or varied the components of the color. In this reading, we will consider the basic building blocks of filters: DrFu's basic operations for transforming colors and the ways to combine them into more complex color transformations. In the next reading, we will see how to use those transformations to transform whole images. After that, we'll explore how you write new transformations.

Some Basic Transformations

Rather than writing every transformation from scratch, we will start with a few basic transformations that DrFu includes.

The simplest transformations are `rgb.darker` and `rgb.lighter`. These operations make a color a little bit darker and a little bit lighter. If you apply them repeatedly, you can darker and darker (or lighter and lighter) colors.

```
> (define sample (cname->rgb "blue violet"))
> (rgb->string sample)
"159/95/159"
> (define darker-sample (rgb.darker sample))
> (rgb->string darker-sample)
"175/111/175"
> (define lighter-sample (rgb.lighter sample))
> (rgb->string lighter-sample)
"143/79/143"
> (define doubly-darker-sample (rgb.darker (rgb.darker sample)))
```

In addition to making the color uniformly darker or lighter, we can also increase individual components using `redder`, `greener`, and `bluer`.

```
> (define sample (cname->rgb "blue violet"))
> (rgb->string sample)
"159/95/159"
> (rgb->string (rgb.redder sample))
"167/95/159"
> (rgb->string (rgb.greener sample))
"159/103/159"
> (rgb->string (rgb.bluersample))
"159/95/167"
```

The `rgb.rotate` procedure rotates the red, green, and blue components of a color. It is intended mostly for fun, but it can also help us think about the use of these components.

```
> (define sample (cname->rgb "blue violet"))
> (rgb->string sample)
"159/95/159"
> (rgb->string (rgb.rotate sample))
"95/159/159"
```

The `rgb.phaseshift` procedure is another procedure with less clear uses. It adds 128 to each component with a value less than 128 and subtracts 128 from each component with a value of 128 or more. While this is somewhat like the computation of a pseudo-complement, it also differs in some ways. Hence, DrFu also provides an `rgb.complement` procedure that computes the pseudo-complement of an RGB color.

From Transforming Colors to Transforming Pixels

Now that we know some basic transformations to apply to colors, we can use those transformations in a variety of ways. First, we can use it to change one pixel in an image. How? We get the color of the pixel, transform it, and then set the color of the pixel. For example, here's how we might phase shift the top-left pixel in the image called `landscape`. in an image.

```
> (image.set-pixel! landscape 0 0 (rbg.phaseshift (image.get-pixel landscape 0 0)))
```

What if we instead wanted to make pixel at (3,2) a bit redder? We'd write something like the following.

```
> (image.set-pixel! landscape 2 3 (rgb.redder (image.get-pixel landscape 2 3)))
```

How about if we wanted to darken the top-left pixel of a different image, one called `portrait`? It would be much the same.

```
> (image.set-pixel! portrait 0 0 (rgb.darker (image.get-pixel portrait 0 0)))
```

As we just noted, each of these examples is quite similar. The examples differ in the image, the position, and the transformation, but the rest of the code is the same. (For example, we need to call both `image.set-pixel!` and `image.get-pixel` in the same way.) We also see ourselves duplicating a lot. In each case, we need to write the name of the image twice and the position twice. As you might guess, having to repeat the same information again and again often leads to errors.

When computer programmers realize that they are writing nearly identical expressions again and again and again, they tend to write new functions that encapsulate the common portions. Many call this process *refactoring*. The designers of DrFu certainly expected people to change pixels, and did so themselves. To help programmers, they refactored the code and devised a more concise way to change a pixel, which they called (`image.transform-pixel!` *`image column row transformation`*). Hence, to do the same three operations given above, using `image.transform-pixel!`, we would write the following.

```
> (image.transform-pixel! landscape 0 0 rgb.phaseshift)
> (image.transform-pixel! landscape 2 3 rgb.redder)
> (image.transform-pixel! portrait 0 0 rgb.darker)
```

This code is certainly a bit more concise, and perhaps even easier to understand. However, behind the scenes, it does exactly the same thing that the previous code. (We'll see how one might write `image.transform-pixel!` in a later lesson. For now, accept that it's been written and does the same thing as the earlier code.)

Is there anything surprising about the `image.transform-pixel!` procedure? We hope you won't find it surprising, but some of you who have programmed before may note something a bit puzzling - We've made one procedure (`rgb.phaseshift`, `rgb.redder`, or `rgb.darker`) a parameter to another procedure (`image.transform-pixel!`). Not all programming languages permit you to make procedures parameters, but those that do can help you write more clearly and concisely, as in this example.

From Transforming Pixels to Transforming Images

If you think back to the beginning of this reading, you may recall that we suggested that one reason to learn to transform colors is that by transforming colors, you can also build filters. Do we have enough information to write a filter for a four-by-three image? Certainly. Suppose we wanted to compute the complement of this image. We could write a sequence of calls to the `image.transform-pixel!` procedure.

```
(image.transform-pixel! canvas 0 0 rgb.complement)
(image.transform-pixel! canvas 0 1 rgb.complement)
(image.transform-pixel! canvas 0 2 rgb.complement)
(image.transform-pixel! canvas 0 3 rgb.complement)
(image.transform-pixel! canvas 1 0 rgb.complement)
```

```
(image.transform-pixel! canvas 1 1 rgb.complement)
(image.transform-pixel! canvas 1 2 rgb.complement)
(image.transform-pixel! canvas 1 3 rgb.complement)
(image.transform-pixel! canvas 2 0 rgb.complement)
(image.transform-pixel! canvas 2 1 rgb.complement)
(image.transform-pixel! canvas 2 2 rgb.complement)
(image.transform-pixel! canvas 2 3 rgb.complement)
```

That's certainly an awful lot of typing, even for a small image. Still, it's all we know how to do right now. In the next reading, we'll consider some disadvantages of this technique and learn how to get DrFu to automatically figure out all of the calls for an image.

Reviewing Key Concepts

So, what should you take away from what we've just learned? You now know a few new functions in DrFu, particularly functions that transform colors. You've now learned about a technique that computer scientists use, refactoring, which involves writing new functions that encapsulate common code. (You have not yet learned how to write these refactored procedures.) You've seen that Scheme permits procedures to take other procedures as parameters, and that this permission supports refactoring.

Immediately, knowing the particular transformations will be helpful. In the future, knowing about refactoring and knowing how to use procedures as parameters will be even more helpful.
