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

RGB Colors

Summary: We examine some basic operations for working with RGB colors.

Contents:

Introduction

In the previous reading, you learned about raster graphics and ways to set and get colors at various locations in a raster image. However, in your work, you were limited to either the colors already in the image or the colors that someone has named for you.

One of the great advantages of computational image making is that it is possible to describe colors that do not have a name. In fact, it is even necessary for named colors to have a common representation. After all, we may not agree on what precisely something like “blood orange” means.

Representing Colors

The most popular scheme for representing colors for display on the computer screen is RGB. We build each color by combining varying amounts of the three primary colors, red, green, and blue. (What, you think that red, yellow, and blue are the primary colors? It turns out that primary works differently when you’re transmitting light, as on the computer screen, than when you’re reflecting light, as when you color with crayons on paper.)

So, for example, purple is created by combining a lot of red, a lot of blue, and essentially no green. You get different purple-like colors by using different amounts of red and blue, and even different ratios of red and blue.

When we describe the amount of red, green, and blue, we can use between 0 and 255 units of each component color. Why do we start with 0? Because we might not want any contribution from that color. Why do we stop with 255? Because 255 is one less than 2^8 (256), and it turns out that numbers between 0 and 255 are therefore easy to represent on computers.

If there are 256 possible values for each component, there are therefore 16,777,216 different colors that we can represent in standard RGB. Can the eye distinguish all of them? Mine certainly can’t. Nonetheless, it is useful to know that this variety is available, and many eyes can make very fine distinctions between nearby colors.

Relevant Procedures

Let us now turn to the primary procedures we will use to work with RGB colors.

We build a new color with the (`rgb.new` *`red-component green-component blue-component`*) procedure. For example, we might set the center of the image to a purple-like color with the following.

```
(image.set-pixel! my-first-image 4 2 (rgb.new 128 0 128))
```

If we have color created by `rgb.new` or extracted with `image.get-pixel` (or computed in one of the ways you'll learn over the next few weeks), we can extract the red, green, and blue components with (`rgb.red` *`color`*), (`rgb.green` *`color`*), and (`rgb.blue` *`color`*). For example, we can remove the green component of the lower-right pixel of the sample image from the previous reading with

```
(define color (image.get-pixel my-first-image 8 4))
(image.set-pixel! my-first-image 8 4 (rgb.new (rgb.red color) 0 (rgb.blue color))
```

These procedures can also be useful for telling us a bit about the named colors. For example, let's figure out the components of blood orange.

```
> (define blood-orange (name->color "blood orange"))
> (rgb.red blood-orange)
204
> (rgb.green blood-orange)
17
> (rgb.blue blood-orange)
0
```

At times, we simply want to find out the three components of the color so that we can think about the meaning of those components. (That is, we will not compute with the values, but simply look at them.) While we could use each of `rgb.red`, `rgb.green`, and `rgb.blue`, as in the sample code above, that requires a bit more effort than seems necessary. Hence, DrFu also provides procedures called (`rgb->rgb-list` *`color`*) and (`rgb->string` *`color`*) that let us view the three components together.

```
> (rgb->rgb-list blood-orange)
(204 17 0)
> (rgb->rgb-list (name->color "corn flower blue"))
(66 66 111)
> (rgb->string blood-orange)
"204/17/0"
> (rgb->string (name->color "cornflower blue"))
"66/66/111"
```

[Design Detour] Computing with Color: Complementary Colors

n creating works, many artists and visual designers consider the applications of *complementary colors*. A pair of colors is complementary if the sum of the two colors is a kind of grey (including black or white). What does it mean to sum two colors? Well, it turns out that complementarity is really defined only for a

different representation of colors (hue, saturation, and value, or HSV). Nonetheless, we can come close to simulating it in RGB, so we will call complementary colors defined using their RGB values *pseudo-complementary colors*.

In RGB, we can add the colors by adding the corresponding components (capping the sum at 255) or by averaging the corresponding components. We'll use the former technique, because it can be a bit easier to analyze capping.

For example, the psuedo-complement of green (0/255/0) is magenta (255/0/255) because when we add them together, we get 255/255/255, which is white.

Depending on what you accept as the definition of "grey", colors can have many psuedo-complements. For example, consider the color 128/0/0, which is similar to blood red. One logical pseudo-complement to that color is 127/255/255 (similar to medium turquoise), since when we add the two colors together, we get 255/255/255, which is still white. However, one might also consider 0/128/128 (a color with no good name, but which DrFu thinks is close to steel blue) as a pseudo-complement, since when we add the two together, we get 128/128/128, a nice shade of medium grey.

In general, when we say "pseudo-complementary color", we mean the one which, when we add the RGB components to those of the first color, we get white. When we ask for multiple psuedo-complements for the same color, we'll mean those that, when added, give us a color in which all three components are the same (that is, a version of grey).

Quick Reference

- `(rgb.new` *`r g b`*`)` - make an RGB color with the given components
- `(rgb.red` *`color`*`)` - extract the red component of an RGB color
- `(rgb.green` *`color`*`)` - extract the green component of an RGB color
- `(rgb.blue` *`color`*`)` - extract the blue component of an RGB color
- `(rgb->string` *`color`*`)` - convert an RGB color to a more human readable representation
- `(rgb->rgb-list` *`color`*`)` - convert an RGB color to a more human readable representation

---

 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.