Seven grayscale conversion algorithms (with pseudocode and VB6 source code)

Oct 1, 2011 • by Tanner Helland 

I have uploaded a great many image processing demonstrations over the years, but today's project - grayscale conversion techniques - is actually the image processing technique that generates the most email queries for me. I'm glad to finally have a place to send those queries!

Despite many requests for a grayscale demonstration, I have held off coding anything until I could really present something *unique*. I don't like adding projects to this site that offer nothing novel or interesting, and there are already hundreds of downloads - in every programming language - that demonstrate standard color-to-grayscale conversions. So rather than add one more "here's a grayscale algorithm" article, I have spent the past week collecting every known grayscale conversion routine. To my knowledge, this is the only project on the Internet that presents seven unique grayscale conversion algorithms, and at least two of the algorithms - custom # of grayscale shades with and without dithering - were written from scratch for this very article.

So without further ado, here are seven unique ways to convert a full-color image to grayscale.

Grayscale - An Introduction

Black and white (or monochrome) photography dates back to the mid-19th century. Despite the eventual introduction of color photography, monochromatic photography remains popular. If anything, the digital revolution has actually *increased* the popularity of monochromatic photography because any

digital camera is capable of taking black-and-white photographs (whereas analog cameras required the use of special monochromatic film). Monochromatic photography is sometimes considered the "sculpture" variety of photographic art. It tends to abstract the subject, allowing the photographer to focus on form and interpretation instead of simply reproducing reality.

Because the terminology *black-and-white* is imprecise - black-and-white photography actually consists of many shades of gray - this article will refer to such images as *grayscale*.

Several other technical terms will be used throughout my explanations. The first is *color space*. A *color space* is a way to visualize a shape or object that represents all available colors. Different ways of representing color lead to different color spaces. The RGB color space is represented as a cube, HSL can be a cylinder, cone, or bicone, YIQ and YPbPr have more abstract shapes. This article will primarily reference the RGB and HSL color spaces.

I will also refer frequently to *color channels*. Most digital images are comprised of three separate color channels: a red channel, a green channel, and a blue channel. Layering these channels on top of each other creates a full-color image. Different color models have different channels (sometimes the channels are colors, sometimes they are other values like *lightness* or *saturation*), but this article will primarily focus on RGB channels.

How all grayscale algorithms fundamentally work

All grayscale algorithms use the same basic three-step process:

1. Get the red, green, and blue values of a pixel
2. Use fancy math to turn those numbers into a single gray value
3. Replace the original red, green, and blue values with the new gray value

When describing grayscale algorithms, I'm going to focus on step 2 - using math to turn color values into a grayscale value. So, when you see a formula like this:

```
Gray = (Red + Green + Blue) / 3
```

Recognize that the actual code to implement such an algorithm looks like:

```
For Each Pixel in Image {

   Red = Pixel.Red
   Green = Pixel.Green
   Blue = Pixel.Blue

   Gray = (Red + Green + Blue) / 3

   Pixel.Red = Gray
   Pixel.Green = Gray
   Pixel.Blue = Gray

}
```

On to the algorithms!

Sample Image:

This bright, colorful promo art for The Secret of Monkey Island: Special Edition will be used to demonstrate each of our seven unique grayscale algorithms.

Method 1 - Averaging (aka "quick and dirty")

This method is the most boring, so let's address it first. "Averaging" is the most common grayscale conversion routine, and it works like this:

```
Gray = (Red + Green + Blue) / 3
```

Fast, simple - no wonder this is the go-to grayscale algorithm for rookie programmers. This formula generates a reasonably nice grayscale equivalent, and its simplicity makes it easy to implement and optimize (look-up tables work quite well). However, this formula is not without shortcomings - while fast and simple, it does a poor job of representing shades of gray relative to the way humans perceive luminosity (brightness). For that, we need something a bit more complex.

Method 2 - Correcting for the human eye (sometimes called "luma" or "luminance," though such terminology isn't really accurate)

It's hard to tell a difference between this image and the one above, so let me provide one more example. In the image below, method #1 or the "average method" covers the top half of the picture, while method #2 covers the bottom half:

If you look closely, you can see a horizontal line running across the center of the image. The top half (the average method) is more washed-out than the bottom half. This is especially visible in the middle-left segment of the image, beneath the cheekbone of the background skull.

The difference between the two methods is even more pronounced when flipping between them at full-size, as you can do in the provided source code. Now might be a good time to download my sample project (available at the bottom of this article) so you can compare the various algorithms side-by-side.

This second algorithm plays off the fact that cone density in the human eye is not uniform across colors. Humans perceive green more strongly than red, and red more strongly than blue. This makes sense from an evolutionary biology standpoint - much of the natural world appears in shades of green, so humans have evolved greater sensitivity to green light. *(Note: this is oversimplified, but accurate.)*

Because humans do not perceive all colors equally, the "average method" of grayscale conversion is inaccurate. Instead of treating red, green, and blue light equally, a good grayscale conversion will weight each color based on how the human eye perceives it. A common formula in image processors (Photoshop, GIMP) is:

```
Gray = (Red * 0.3 + Green * 0.59 + Blue * 0.11)
```

Surprising to see such a large difference between the red, green, and blue coefficients, isn't it? This formula requires a bit of extra computation, but it results in a more dynamic grayscale image. Again, downloading the sample program is the best way to appreciate this, so I recommend grabbing the code, experimenting with it, then returning to this article.

It's worth noting that there is disagreement on the best formula for this type of grayscale conversion. In my project, I have chosen to go with the original ITU-R recommendation (BT.709, specifically) which is the historical precedent. This formula, sometimes called Luma, looks like this:

```
Gray = (Red * 0.2126 + Green * 0.7152 + Blue * 0.0722)
```

Some modern digital image and video formats use a different recommendation (BT.601), which calls for slightly different coefficients:

```
Gray = (Red * 0.299 + Green * 0.587 + Blue * 0.114)
```

A full discussion of which formula is "better" is beyond the scope of this article. For further reading, I strongly suggest the work of Charles Poynton. For 99% of programmers, the difference between these two formulas is irrelevant. Both are perceptually preferable to the "average method" discussed at the top of this article.

Method 3 - Desaturation

Next on our list of methods is *desaturation*.

There are various ways to describe the color of a pixel. Most programmers use the RGB color model, where each color is described by its red, green, and blue components. While this is a nice way for a machine to describe color, the RGB color space can be difficult for humans to visualize. If I tell you, "oh, I just bought a car. Its color is RGB(122, 0, 255)," you probably can't picture the color I'm describing. If, however, I say, "I just bought a car. It is a bright, vivid shade of violet," you can probably picture the color in question.

For this reason (among others), the HSL color space is sometimes used to describe colors. HSL stands for *hue*, *saturation*, *lightness*. *Hue* could be considered the name of the color - red, green, orange, yellow, etc. Mathematically, hue is described as an angular dimension on the color wheel (range [0,360]), where pure red occurs at 0°, pure green at 120°, pure blue at 240°, then back to pure red at 360°. Saturation describes how vivid a color is; a very vivid color has full saturation, while gray has no

saturation. Lightness describes the brightness of a color; white has full lightness, while black has zero lightness.

Desaturating an image works by converting an RGB triplet to an HSL triplet, then forcing the saturation to zero. Basically, this takes a color and converts it to its *least-saturated variant*. The mathematics of this conversion are more complex than this article warrants, so I'll simply provide the shortcut calculation. A pixel can be desaturated by finding the midpoint between the maximum of (R, G, B) and the minimum of (R, G, B), like so:

```
Gray = ( Max(Red, Green, Blue) + Min(Red, Green, Blue) ) / 2
```

In terms of the RGB color space, desaturation forces each pixel to a point along the neutral axis running from (0, 0, 0) to (255, 255, 255). If that makes no sense, take a moment to read this wikipedia article about the RGB color space.

Desaturation results in a flatter, softer grayscale image. If you compare this desaturated sample to the human-eye-corrected sample (Method #2), you should notice a difference in the contrast of the image. Method #2 seems more like an Ansel Adams photograph, while desaturation looks like the kind of grayscale photo you might take with a cheap point-and-shoot camera. Of the three methods discussed thus far, desaturation results in the flattest (least contrast) and darkest overall image.

Method 4 - Decomposition (think of it as *de-composition*, *e.g.* not the biological process!)

Decomposition using maximum values

Decomposition using minimum values

Decomposing an image (sounds gross, doesn't it?) could be considered a simpler form of desaturation. To decompose an image, we force each pixel to the highest (maximum) or lowest (minimum) of its red, green, and blue values. Note that this is done on a per-pixel basis - so if we are performing a *maximum* decompose and pixel #1 is RGB(255, 0, 0) while pixel #2 is RGB(0, 0, 64), we will set pixel #1 to 255 and pixel #2 to 64. Decomposition only cares about which color value is highest or lowest - not which channel it comes from.

Maximum decomposition:

```
Gray = Max(Red, Green, Blue)
```

Minimum decomposition:

```
Gray = Min(Red, Green, Blue)
```

As you can imagine, a maximum decomposition provides a brighter grayscale image, while a minimum decomposition provides a darker one.

This method of grayscale reduction is typically used for artistic effect.

Method 5 - Single color channel

Grayscale generated using only red channel values

Grayscale generated using only green channel values

Grayscale generated using only blue channel values

Finally, we reach the fastest computational method for grayscale reduction - using data from a single color channel. Unlike all the methods mentioned so far, this method requires no calcuations. All it does is pick a single channel and make that the grayscale value, as in:

```
Gray = Red
```

...or:

```
Gray = Green
```

...or:

```
Gray = Blue
```

Believe it or not, this weak algorithm is the one most digital cameras use for taking "grayscale" photos. CCDs in digital cameras are comprised of a grid of red, green, and blue sensors, and rather than perform the necessary math to convert RGB values to gray ones, they simply grab a single channel (green, for the reasons mentioned in Method #2 - human eye correction) and call that the grayscale one. For this reason, most photographers recommend **against** using your camera's built-in grayscale option. Instead, shoot everything in color and then perform the grayscale conversion later, using whatever method leads to the best result.

It is difficult to predict the results of this method of grayscale conversion. As such, it is usually reserved for artistic effect.

Method 6 - Custom # of gray shades

Grayscale using only 4 shades - black, dark gray, light gray, and white

Now it's time for the fun algorithms. Method #6, which I wrote from scratch for this project, allows the user to specify how many shades of gray the resulting image will use. Any value between 2 and 256 is accepted; 2 results in a black-and-white image, while 256 gives you an image identical to Method #1 above. This project only uses 8-bit color channels, but for 16 or 24-bit grayscale images (and their resulting 65,536 and 16,777,216 maximums) this code would work just fine.

The algorithm works by selecting X # of gray values, equally spread (inclusively) between zero luminance - black - and full luminance - white. The above image uses four shades of gray. Here is another example, using sixteen shades of gray:

This grayscale algorithm is a bit more complex. It looks something like:

```
ConversionFactor = 255 / (NumberOfShades - 1)
AverageValue = (Red + Green + Blue) / 3
Gray = Integer((AverageValue / ConversionFactor) + 0.5) * ConversionFactor

Notes:
-NumberOfShades is a value between 2 and 256
-technically, any grayscale algorithm could be used to calculate AverageValue; it simply
 an initial gray value estimate
-the "+ 0.5" addition is an optional parameter that imitates rounding the value of an in
 conversion; YMMV depending on which programming language you use, as some round automat
```

I enjoy the artistic possibilities of this algorithm. The attached source code renders all grayscale images in real-time, so for a better understanding of this algorithm, load up the sample code and rapidly scroll between different numbers of gray shades.

Method 7 - Custom # of gray shades with dithering (in this example, horizontal error-diffusion dithering)

This image also uses only four shades of gray (black, dark gray, light gray, white), but it distributes those shades using error-diffusion dithering.

Our final algorithm is perhaps the strangest one of all. Like the previous method, it allows the user to specify any value in the [2,256] range, and the algorithm will automatically calculate the best spread of grayscale values for that range. However, this algorithm also adds full dithering support.

What is dithering, you ask? In image processing, dithering uses optical illusions to make an image look more colorful than than it actually is. Dithering algorithms work by interspersing whatever colors are available into new patterns - ordered or random - that fool the human eye into perceiving more colors than are actually present. If that makes no sense, take a look at this gallery of dithered images.

There are many different dithering algorithms. The one I provide here is one of the simplest error-diffusion mechanisms: a one-dimensional diffusion that bleeds color conversion errors from left to right.

If you look at the image above, you'll notice that only four colors are present - black, dark gray, light gray, and white - but because these colors are mixed together, from a distance this image looks much sharper than the four-color non-dithered image under Method #6. Here is a side-by-side comparison:

When few colors are available, dithering preserves more nuances than a non-dithered image, but the trade-off is a "dirty," speckled look. Some dithering algorithms are better than others; the one I've used falls somewhere in the middle, which is why I selected it.

As a final example, here is a 16-color grayscale image with full dithering, followed by a side-by-side comparison with the non-dithered version. As the number of shades of gray in an image increases, dithering artifacts become less and less noticeable. Can you tell which side of the second image is dithered and which is not?


THE SECRET OF
MONKEY ISLAND
SPECIAL EDITION

Because the code for this algorithm is fairly complex, I'm going to refer you to the download for details. Simply open the `Grayscale.frm` file in your text editor of choice, then find the `drawGrayscaleCustomShadesDithered` sub. It has all the gory details, with comments.

Conclusion

If you're reading this from a slow Internet connection, I apologize for the image-heavy nature of this article. Unfortunately, the only way to really demonstrate all these grayscale techniques is by showing many examples!

The source code for this project, like all image processing code on this site, runs in real-time. The GUI is simple and streamlined, automatically hiding and displaying relevant user-adjustable options as you click through the various algorithms:

Each algorithm is provided as a stand-alone method, accepting a source and destination picturebox as parameters. I designed it this way so you can grab whatever algorithms interest you and drop them straight into an existing project, without need for modification.

Comments and suggestions are welcome. If you know of any interesting grayscale conversion algorithms I might have missed, please let me know.

(Fun fact: want to convert a grayscale image back to color? If so, check out my real-time image colorization project.)

Download the source code from GitHub

Got a comment, question, or other feedback on this page? Submit an issue at GitHub.

tannerhelland
get updates

Find this content useful? Support me on Patreon, or consider making a one-time donation.