2 The Science of Digital Media, Chapter 2: Digital Image Representation[1]

Beauty in art has the same base as truth in philosophy. What is the truth? The conformity of our judgements with that of others. What is a beautiful imitation? The conformity of the image with the thing.

Denis Diderot

Objectives for Chapter 2

[1] This material is based on work support by the National Science Foundation. The first draft of this chapter was written under Grant No. DUE-0127280. The final version was written under Grant No. 0340969. This chapter was written by Dr. Jennifer Burg (burg@wfu.edu).

- Understand the difference between bitmap and vector graphic representations of image data.
- Understand the difference between representing image data in the spatial versus the frequency domain, and be able to visualize image data as it is represented graphically in each domain.
- Be able to transform image data between the spatial and frequency domains by means of the discrete cosine transform.
- Know how base frequencies of the discrete cosine transform are represented as sinusoidal waves and visualized as changing grayscale or color values.
- Be able to apply the Nyquist theorem to an understanding of aliasing and moiré patterns.
- Understand an example demosaicing algorithm for color aliasing.
- Understand an example anti-aliasing algorithm.
- Understand color models, be able to visualize the important models graphically, and be able to transform from one model to another as needed.
- Understand how parametric curves are created in vector graphics, and be able to apply the equations that define them.
- Understand algorithms for producing fractal images.

Mathematics Needed for Chapter 2 (in addition to items listed for previous chapters)

- inverse of a function
- parametric equations
- graphical and mathematical projections
- explicit vs. implicit forms of functions
- imaginary numbers (for fractals only)
- complex numbers (for fractals only)
- derivatives
- tangents

Programming and Data Structures Background Needed for Chapter 2 (in addition to items listed for previous chapters)

- recursion

2.1 Introduction

Digital images are created by three basic methods: ***bitmapping***, ***vector graphics***, or ***procedural modeling***. ***Bitmap images*** (also called ***pixmaps*** or ***raster graphics***) are created with a pixel-by-pixel specification of points of color. ***Bitmaps*** are commonly created by digital cameras, scanners, paint programs like Corel Paint Shop Pro, and image processing programs like Adobe Photoshop. Vector graphic images – created in programs such as Adobe Illustrator and Corel Draw – use object specifications and

mathematical equations to describe shapes to which colors are applied. A third way to create digital images is by means of procedural modeling – also called ***algorithmic art*** because of its aesthetic appeal – where a computer program uses some combination of mathematics, logic, control structures, and recursion to determine the color of pixels and thereby the content of the overall picture. Fractals and Fibonacci spirals are examples of algorithmic art.

Bitmaps are appropriate for photographic images, where colors change subtly and frequently in small gradations. Vector graphic images are appropriate for cleanly delineated shapes and colors, like cartoon or poster pictures. Procedurally modeled images are algorithmically interesting in the way they generate complex patterns, shapes, and colors in non-intuitive ways. All three methods of digital image creation will be discussed in this chapter, with the emphasis on how pixels, colors, and shapes are represented. Techniques for manipulating and compressing digital images are discussed in Chapter 3.

2.2 Bitmaps

2.2.1 Digitization

A ***bitmap*** is two-dimensional array of pixels describing a digital image. Each ***pixel*** short for *picture element* is a number representing the color position (*r,c*) in the bitmap, where *r* is the row and *c* is the column.

> **Aside**: A bitmap is a pixel-by-pixel specification of points of color in an image file. But the prefix *bit* seems to imply that each pixel is represented in only one bit, and thus it could have only two possible values, 0 or 1, usually corresponding to black and white (though it could be any two colors). It seems that the word *pixmap*, short for *pixel map*, would be more descriptive, but this word has never really caught on. Most people use the term *bitmap* even when referring to images that use more than one bit per pixel.

There are three main ways that you could create a bitmap. One is through software – by means of a paint program. With such a program, you could paint your picture one pixel at a time, choosing a color for a pixel, then clicking on that pixel to apply the color. More likely, you would drag the mouse to paint whole areas with brush strokes, a less tedious process. In this way, you could create your own artwork – whatever picture you hold in your imagination – as a bitmap image.

More commonly, however, bitmap images are reproductions of scenes and objects we perceive in the world around us. One way to create a digital image from real world objects or scenes is to take a snapshot with a traditional analog camera, have the film developed, and then scan the photograph with a digital scanner – giving you a bitmap image that can be stored on your computer.

A third and more direct route is to shoot the image with a digital camera and transfer the bitmap to your computer. Digital camera's can have various kinds of memory cards – sometimes called *flash memory* – on which the digital images are stored. You can transfer the image to your computer either by making a physical connection (e.g., USB) between the camera and the computer or by inserting the camera's memory card into a slot on your computer, and then downloading the image file. Our emphasis in this book will be on bitmap images created from digital photography.

Digital cameras use the same digitization process discussed in Chapter 1. This digitization process always reduces to two main steps: sampling and quantization. Sampling rate for digital cameras is a matter of how many points of color are sampled

and recorded in each dimension of the image. You generally have some choices in this regard. For example, a digital camera might allow you to choose from1600 × 1200, 1280 × 960, 1024 × 768, and 640 × 480. Some cameras offer no choice.

In digital cameras, quantization is a matter of the color model used and the corresponding bit depth. We will look at color models more closely later in this chapter. Suffice it to say for now that digital cameras generally use RGB color, which saves each pixel in three bytes, one for each of the color channels – red, green, and blue. (A higher bit depth is possible in RGB, but three bytes per pixel is common.) Since three bytes is 24 bits, this makes it possible for 2^{24} = 16,777, 216 colors to be represented.

Both sampling and quantization can introduce error – error in the sense that the image captured does not represent, with perfect fidelity, the original scene or objects that were photographed. If you don't take enough samples over the area being captured, the image will lack clarity. The larger the area represented by a pixel, the blurrier the picture because subtle transitions from one color to the next cannot be captured.

We illustrate this with the figures below. Imagine that you are looking at a natural scene like the one pictured in Figure 2.1. Suppose that you divide the image into 15 rows and 20 columns. This gives you 15 × 20 rectangular sample areas. You (a hypothetical camera) sample the image once per rectangle, using as your sample value the average color in each square. If you then create the image using the sample values, the image (Figure 2.2) obviously lacks detail. It is blocky and unclear.

Figure 2.1 Digital image

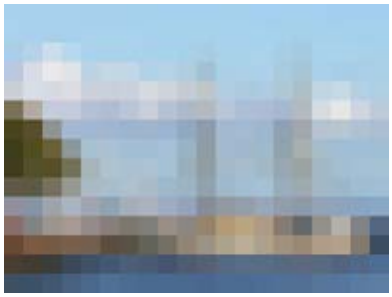


Figure 2.2 Image, undersampled

Figure 2.3 Image, reduced bit depth

A low bit depth, on the other hand, can result in patchiness of color. The bit depth used by digital cameras is excellent for color detail. However, after taking a picture and loading it onto your computer, you can work with it in an image processing program and from there reduce its bit depth. (You might do this to reduce the file size.) Consider this exaggerated example, to make the point. If you reduce the number of colors in the boat picture from a maximum of 16,777,216 to a maximum of 12, you get the image in Figure

2.3. Whole areas of color have been made a single color, as you can see in the clouds. This is the effect of quantization error on a digital image.

2.2.2 Pixel Dimensions, Resolution, and Image Size

We defined pixel in the context of a bitmap, but it has another meaning in the context of a computer display – where pixel is defined as a physical object a point of light on the screen. Sometimes the terms ***logical pixel*** and ***physical pixel*** are used to distinguish between the two usages. When you display a bitmap image on your computer, the logical pixel – that is, the number representing a color and stored for a given position in the image file – is mapped to a physical pixel on the computer screen.

For an image file, ***pixel dimensions*** is defined as the number of pixels horizontally (i.e., width, w) and vertically (i.e., height, h) denoted $w \times h$. For example, your digital camera might take digital images with pixel dimensions of 1600 × 1200. Similarly, your computer screen has a fixed maximum pixel dimensions – e.g., 1024 × 768 or 1400 × 1050.

Digital cameras are advertised as offering a certain number of ***megapixels***. The megapixel value is derived from the maximum pixel dimensions allowable for pictures taken with the camera. For example, suppose the largest picture you can take, in pixel dimensions, for a certain camera is 2048 × 1536. That's a total of 3,145,728 pixels. That makes this camera a 3 megapixel camera (approximately). (Be careful. Camera makers sometimes exaggerate their camera's megapixel values by including such things as "optical zoom," a software method for increasing the number of pixels without really improving the clarity.)

Resolution is defined as the number of pixels in an image file per unit of spatial measure. For example, resolution can be measured in pixels per inch, abbreviated *ppi*. It is assumed that the same number of pixels are used in the horizontal and vertical directions, so a 200 ppi image file will print out using 200 pixels to determine the colors for each inch in both the horizontal and vertical directions.

Resolution of a printer is a matter of how many dots of color it can print over an area. A common measurement is ***dots per inch*** (DPI). For example, an inkjet printer might be able to print a maximum of 1440 DPI. The printer and its software map the pixels in an image file to the dots of color printed. There may be more or fewer pixels per inch than dots printed. You should take a printer's resolution into consideration when you create an image to be printed. There's no point to having more resolution than the printer can accommodate.

Image size is defined as the physical dimensions of an image when it is printed out or displayed on a computer, e.g. in inches (abbreviated ") or centimeters. By this definition, image size is a function of the pixel dimensions and resolution, as follows:

> **Aside**: Unfortunately, not everyone makes a distinction between pixel dimensions and resolution, the term *resolution* commonly being used for both. For example, your computer display's pixel dimensions or the pixel dimensions of an image may be called their *resolution* (with no implication of "per inch"). The confusion goes even further. *Resolution* is sometimes used to refer to bit depth, thus being related to the number of colors that can be represented in an image file. And the documentation for some cameras uses the term *image size* where we would say pixel dimensions. Be alert to these differences in usage. It is the concepts that are important.

For an image with resolution r and pixel dimensions $w \times h$ where w is the width and h is the height, the printed image size will be $a \times b$ as given by

$$a = w/r$$
and
$$b = h/r$$

key equation

For example, if you have an image that is 1600 × 1200 and you choose to print it out at 200 ppi, it will be 8" × 6".

You can also speak of the image size as the image appears on a computer display. For an image with pixel dimensions $h \times w$ and resolution r, the displayed image size is, as before, be $w/r \times h/r$. However, in this case r is the display screen's resolution. For example, if your computer display screen has pixel dimensions of 1400 × 1050 and it is 12" × 9", then the display has a resolution of about 117 ppi. Thus, a 640 × 480 image, when shown at 100% magnification, will be about 5½"× 4". This is because each logical pixel is displayed by one physical pixel on the screen.

For an image with pixel dimensions $w \times h$ where w is the width and h is the height displayed on a computer display with resolution r at 100% magnification, the displayed image size will be $a \times b$ as given by

$$a = w/r$$
and
$$b = h/r$$

(It is assumed that the display resolution is given in pixels per inch or centimeters per inch.)

key equation

The original pixel dimensions of the image file depend on how you created the image. If the image originated as a photograph, its pixel dimensions may have been constrained by the allowable settings in your digital camera or scanner that captured it. The greater the pixel dimensions of the image, the more faithful the image will be to the scene captured. A 300 × 400 image will not be as crisp and detailed as a 900 × 1200 image of the same subject.

You can see the results of pixel dimensions in two ways. First, with more pixels to work with, you can make a larger-sized print and still have sufficient resolution for the printed copy. Usually, you'll want your image to be printed at a resolution of between 100 and 300 ppi, depending on the type of printer you use. (Check the specifications of your printer to see what is recommended.) The printed size is $w/r \times h/r$, so the bigger the r, the smaller the print. For example, if you print your 300 × 400 image out at 100 ppi, it will be 3" × 4". If you print it out at 200 ppi, it will be 1½" × 2".

Supplement on pixel dimensions, resolution, and image size:

hands-on exercise

The second way you see the result of pixel dimensions is in the size of the image on your computer display. Since logical pixels are mapped to physical pixels, the more pixels in the image, the larger the image on the display. A 300 × 400 image will be 1/3 the size of a 900 × 1200 on the display in each dimension. It's true that you can ask the computer to magnify the image for you, but this won't create detail that wasn't captured in

the original photograph. If you magnify the 300 × 400 image by 300% and compare it to an identical image that was originally taken with pixel dimensions of 900 × 1200, the magnified image won't look as clear. It will have jagged edges.

Thus, when you have a choice of the pixel dimensions of your image, you need to consider how you'll be using the image. Are you going to be viewing it from a computer or printing it out? How big to you want it to be, either in print size or on the computer screen. If you're printing it out, what resolution do you want to use for the print? With the answers to these questions in mind, you can choose appropriate pixel dimensions – or what is closest to your needs – based on the choices offered by your camera and the amount of memory you have for storing the image.

There are times when you can't get exactly the pixel dimensions you want. Maybe your camera or scanner has limitations, maybe you didn't take the picture yourself, or maybe you want to crop the picture to cut out just the portion of interest. (***Cropping***, in an image processing program, is simply cutting off part of the picture, discarding the unwanted pixels.) Changing the number of pixels in an image is called ***resampling***. You can increase the pixel dimensions by ***upsampling*** or decrease the dimensions by ***downsampling***, and you may have valid reasons for doing either. But keep in mind that resampling always involves some kind of interpolation, averaging, or estimation, and thus it cannot improve the quality of an image in the sense of making it any more faithful to the picture being represented. The additional pixels created by upsampling are just "estimates" of what the original pixel values would have been if you had originally captured the image at higher pixel dimensions, and pixel values you get from downsampling are just averages of the information you originally captured. In Chapter 3, we'll look more closely at how resampling is done.

The lack of clarity that results from a sampling rate that is too low for the detail of the image is an example of aliasing. You may recall from Chapter 1 that aliasing is a phenomenon where one thing "masquerades" as another. In the case of digital imaging, if the sampling rate is too low, then the image takes on a shape or pattern different what was actually being photographed – blockiness, blurriness, jagged edges, or moiré patterns (which, informally defined, are patterns that are created from two other patterns overlapping each other at an angle). Similarly, a digitized sound might adopt false frequencies heard as false pitches, or digitized video might demonstrate motion not true to the original, like spokes in a bicycle wheel rotating backwards. As explained in Chapter 1, the ***Nyquist theorem*** tells us that aliasing results when the sampling rate is not at least twice the frequency of the highest frequency component of the image (or sound or video) being digitized. To understand this theorem, we need to be able to think of digital images in terms of frequencies, which may seem a little counter-intuitive. It isn't hard to understand frequency with regard to sound, since we're accustomed to thinking of sound as a wave and relating the frequency of the sound wave to the pitch of the sound. But what is frequency in the realm of digital images?

2.3 Frequency in Digital Images

Chapter 1 gives a general discussion of how data can be represented by functions in a variety of domains, and how the representation can be translated from one domain to another without loss of information. Let's look at this more closely in the context of digital images. Our goal is understand how a digital image can be translated from the

spatial domain to the frequency domain, because once we see how this is done, we can understand what the Nyquist theorem means when applied to digital images.

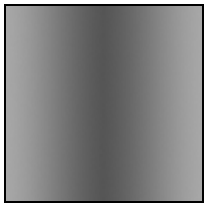


Figure 2.4
An image whose color varies continuously from light gray to dark gray and back again

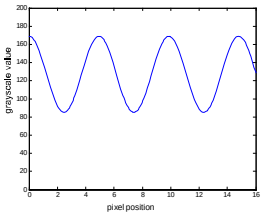


Figure 2.5
Graph of function y=f(x) for one line of color across the image. f is assumed to be continuous and periodic.

To begin, let's think of an image as a function that can be represented in a graph. It's easier to visualize this with a function in one variable, so let's consider just one line of color across an image as a function $y = f(x)$, where x is the position of one point of color. Function f is ***a function over the spatial domain*** in that the x values correspond to points in space. y is the color value at position x.

Figure 2.4 shows an image that is a pattern of grayscale values that vary from lighter gray (a grayscale value of 168) to darker gray (a grayscale value of 85) and back again. (All examples in this section will be based on grayscale images. The observations are easily generalized to RGB images, where each of the red, green, and blue channels has values between 0 and 255. Each channel can be individually translated from the spatial to the frequency domain.) Grayscale values for one horizontal line across the image are graphed as the function $y = f(x)$ in Figure 2.5, assuming that the pattern repeats. Notice that the graph is sinusoidal – the shape of a sine or cosine wave. Because this is such a simple example, it is easy to see the image in terms of frequency. An important concept to remember is that in the realm of digital imaging, *frequency refers to the rate at which color values change*. This image changes at a perfectly regular and continuous rate, resulting in a sinusoidal shape to the graph.

So far we've been treating the color values in an image as a continuous function, but to work with these values in a digital image we need to discretize them. Figure 2.6 is a discretized version of Figure 2.4 in that it has discrete bands of colors and the change from one color to the next is abrupt. Assume that this is a 8 × 8 pixel image, and imagine extending the image horizontally such that the eight-pixel pattern is repeated four times. Then consider just one row of pixels across the image. Figure 2.7 shows how such an image would be graphed in the spatial domain. We can still consider this a waveform, though it is not smooth like a sine or cosine wave. All bitmap images, even those with irregular patterns of pixel values like the picture of the sparrow shown in Figure 2.8, can be viewed as waveforms. Figure 2.9 shows the graph of one row of pixels taken from the sparrow picture.

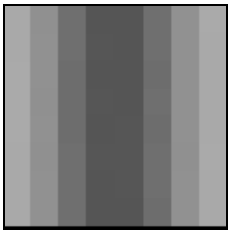


Figure 2.6
Discretized version of gradient

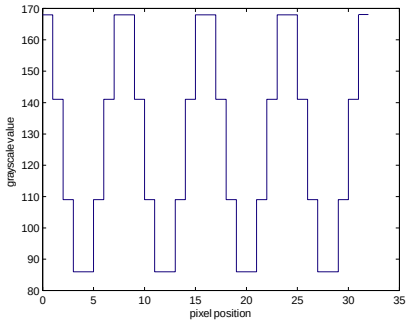


Figure 2.7
Graph of one row of pixels over the spatial domain (Assume the pattern shown in Figure 2.6 is repeated four times in this graph, making a 32 pixel width.)

Figure 2.8 A grayscale bitmap image

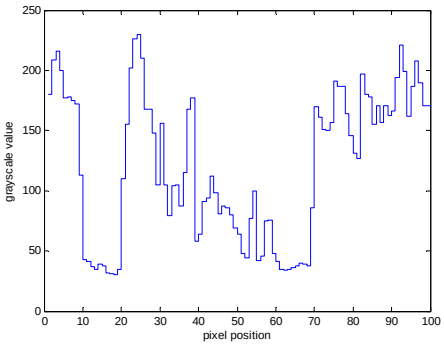


Figure 2.9 Graph of one row of sparrow bitmap over the spatial domain

Extending these observations to two-dimensional images is straightforward. If we assume that the picture in Figure 2.6 is an 8 × 8 pixel bitmap, then the corresponding graph over the spatial domain (Figure 2.10) is a three-dimensional graph where the third dimension is the pixel value at each *(x,y)* position in the image. The graph for the picture of the sparrow is shown in Figure 2.11. When the two-dimensional bitmap images are graphed, you can imagine that each row (and each column) is a complex waveform.

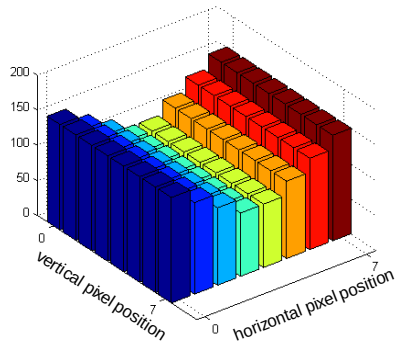


Figure 2.10 Graph of Figure 2.6, assumed to be an 8 × 8 bitmap image

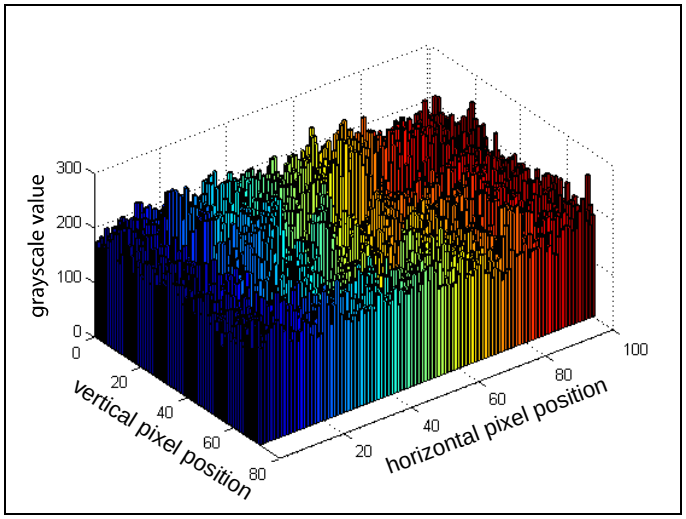


Figure 2.11 Graph of two-dimensional bitmap of sparrow

2.4 The Discrete Cosine Transform

> **Aside**: Fourier theory was developed by mathematician and physical scientist Jean Baptiste Joseph Fourier (1768-1830) and is applicable to a wide range of problems in engineering and digital signal processing.

All of the graphs we've looked at so far have been represented in the spatial domain. To separate out individual frequency components, however, we need to translate the image to the frequency domain. It is known by Fourier theory that any complex periodic waveform can be equated to an infinite sum of simple sinusoidal waves of varying frequencies and amplitudes. To understand what this means, think first about just one horizontal line of image data, picturing it as a complex waveform. The fact that this waveform can be expressed as an infinite sum of simple sinusoidals is expressed in the equation below.

$$f(x) = \sum_{n=0}^{\infty} a_n \cos(n\omega x)$$

Equation 2.1

$f(x)$ is a continuous function over the spatial domain whose graph takes the form of a complex waveform. The equation says that it is equal to an infinite sum of cosine waves

which make up its frequency components. Our convention will be to use ω to represent angular frequency, where $\omega = 2\pi f$ and f is the fundamental frequency of the wave. As n varies, we move through these frequency components, from the fundamental frequency on through multiples of it. a_n is the amplitude for the n^{th} cosine frequency component.

We already showed you a simple example of this in Chapter 1. The figure below shows a sound wave that is the sum of three simple sine waves. (All other frequency components other than those shown have amplitude 0 at all points.)

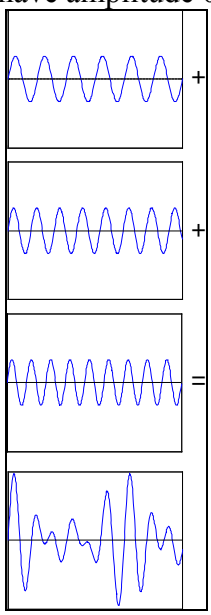


Figure 2.12 Adding frequency components

Let's look at how this translates to the discrete world of digital images. We continue to restrict our discussion to the one-dimensional case – considering a single line of pixels across a digital image, like the one pictured in Figure 2.13. Any row of M pixels can be represented as a sum of the M weighted cosine functions evaluated at discrete points. This is expressed in the following equation:

$$\boldsymbol{f}(r) = \sum_{u=0}^{M-1} \frac{\sqrt{2}C(u)}{\sqrt{M}} \boldsymbol{F}(u) \cos\left(\frac{(2r+1)u\pi}{2M}\right)$$

$$\text{where } C(\delta) = \frac{\sqrt{2}}{2} \text{ if } \delta = 0 \text{ otherwise } C(\delta) = 1$$

Equation 2.2

You can understand Equation 2.2 in this way. ***f****(r)* is a one-dimensional array of M pixel values. Corresponding to Figure 2.13, these values would be [0, 0, 0, 153, 255, 255, 220, 220]. ***F****(u)* is one-dimensional array of coefficients. Each function $\cos\left(\frac{(2r+1)u\pi}{2M}\right)$ is called a ***basis function***. You can also think of each function as a ***frequency component***. The coefficients in ***F****(u)* tell you how much each frequency component is weighted in the sum that produces the pixel values. You can think of this as "how much" each frequency component contributes to the image.

Figure 2.13 A one-dimensional image of eight pixels (enlarged). Pixel outlines are not part of image.

For $M = 8$, the basis functions are those given below. They are shown as cosine functions (made continuous) and then as 8×8 bitmap images whose color values change in accordance with the given basis function. As the values of the cosine function decreases, the pixels get darker because 1 represents white and -1 represents black.

$$\cos\left(\frac{(2r+1)0\pi}{16}\right) = \cos(0)$$

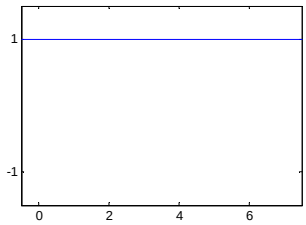


Figure 2.14 Basis Function 0

$$\cos\left(\frac{(2r+1)\pi}{16}\right)$$

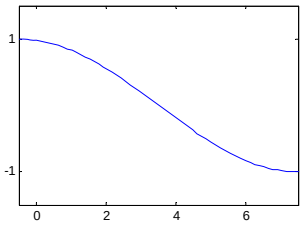


Figure 2.15 Basis Function 1

$$\cos\left(\frac{(2r+1)2\pi}{16}\right)$$

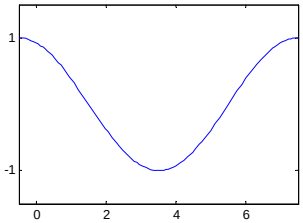


Figure 2.16 Basis Function 2

$$\cos\left(\frac{(2r+1)3\pi}{16}\right)$$

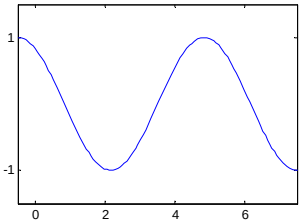


Figure 2.17 Basis Function 3

$$\cos\left(\frac{(2r+1)4\pi}{16}\right)$$

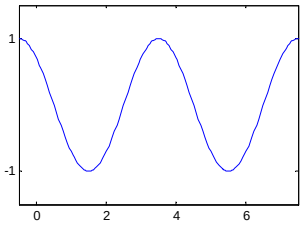


Figure 2.18 Basis Function 4

$$\cos\left(\frac{(2r+1)5\pi}{16}\right)$$

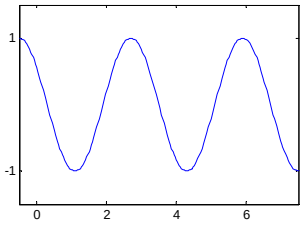


Figure 2.19 Basis Function 5

$$\cos\left(\frac{(2r+1)6\pi}{16}\right)$$

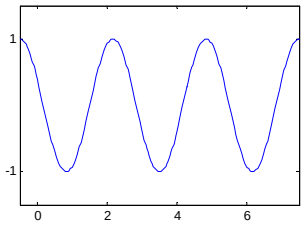


Figure 2.20 Basis Function 6

$$\cos\left(\frac{(2r+1)7\pi}{16}\right)$$

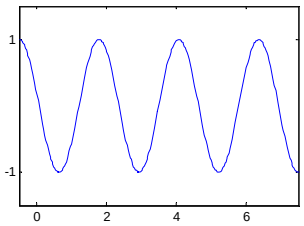


Figure 2.21 Basis Function 7

The pictures of grayscale bars correspond to the sinusoidal graphs as follows: Evaluate the basis function at $x = i$ for $0 <= i <= 7$ from left to right. Each of the resulting values corresponds to a pixel in the line of pixels to the right of the basis function, where the grayscale values are scaled in the range of -1 (black) to 1 (white). Thus, frequency component $\cos\left(\frac{(2r+1)\pi}{16}\right)$ (basis function 1) corresponds to a sequence of eight pixels that go from white to black.

Equation 2.2 states only that the coefficients ***F(u) exist*** , but it doesn't tell you how to compute them. This is where the ***discrete cosine transform*** (***DCT***) comes in. In the one-dimensional case, the discrete cosine transform is stated as follows:

$$\boldsymbol{F}(u) = \sum_{r=0}^{M-1} \frac{2C(u)}{\sqrt{M}} \boldsymbol{f}(r) \cos\left(\frac{(2r+1)u\pi}{2M}\right)$$

$$\text{where } C(\delta) = \frac{\sqrt{2}}{2} \text{ if } \delta = 0 \text{ otherwise } C(\delta) = 1$$

Equation 2.3

This equation tells how to transform an image from the spatial domain – which gives color or grayscale values – to the frequency domain – which gives coefficients by which the frequency components should be multiplied. For example, consider the row of eight pixels shown in Figure 2.13. The corresponding grayscale values are [0, 0, 0, 153, 255, 255, 220, 220]. This array represents the image in the spatial domain. If you compute a value ***F(u)*** for $0 \le u \le M-1$ using Equation 2.3, you get the array of values [389.97, −280.13, −93.54, 83.38, 54.09, −20.51, −19.80, −16.34]. You have applied the DCT, yielding an array that represents the pixels in the frequency domain.

What this tells you is that the line of pixels is a linear combination of frequency components – i.e., the basis functions multiplied by the coefficients in **F** and a constant and added together, as follows:

$$\boldsymbol{f}(r) = \frac{389.97}{\sqrt{M}}\cos(0) + \frac{\sqrt{2}}{\sqrt{M}}\left(-280.13\cos\left(\frac{(2r+1)\pi}{2M}\right) - 93.54\cos\left(\frac{(2r+1)2\pi}{2M}\right) + \right.$$
$$83.38\cos\left(\frac{(2r+1)3\pi}{2M}\right) + 54.09\cos\left(\frac{(2r+1)4\pi}{2M}\right) - 20.51\cos\left(\frac{(2r+1)5\pi}{2M}\right) -$$
$$\left. 19.80\cos\left(\frac{(2r+1)6\pi}{2M}\right) - 16.34\cos\left(\frac{(2r+1)7\pi}{2M}\right)\right)$$

You can understand this visually through Figure 2.22.

Supplements on discrete cosine transform:

interactive tutorial

programming exercise

worksheet

worksheet

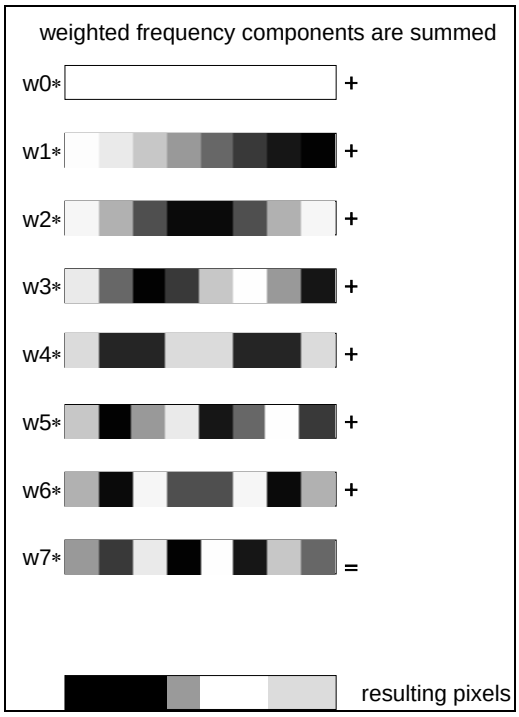


Figure 2.22

Having a negative coefficient for a frequency components amounts to adding the inverted waveform, as shown in Figure 2.23.

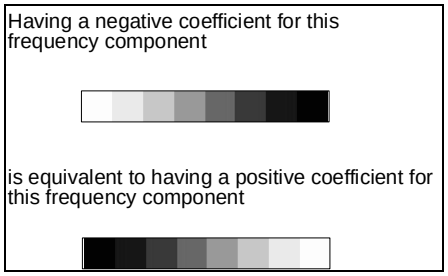


Figure 2.23 A frequency component with a negative coefficient

As a matter of terminology, you should note that the first element ***F(0)*** – is called the ***DC component***. For a periodic function represented in the frequency domain, the DC component is a scaled average value of the waveform. You can see this in the one-dimensional case.

$$\boldsymbol{F}(0) = \sqrt{\frac{2}{M}}\left(\begin{array}{l}\cos(0)\boldsymbol{f}(0) + \cos(0)\boldsymbol{f}(1) + \cos(0)\boldsymbol{f}(2) + \cos(0)\boldsymbol{f}(3) + \\ \cos(0)\boldsymbol{f}(4) + \cos(0)\boldsymbol{f}(5) + \cos(0)\boldsymbol{f}(6) + \cos(0)\boldsymbol{f}(7)\end{array}\right) =$$

$$\sqrt{\frac{2}{M}}\left(\begin{array}{l}\boldsymbol{f}(0) + \boldsymbol{f}(1) + \boldsymbol{f}(2) + \boldsymbol{f}(3) + \\ \boldsymbol{f}(4) + \boldsymbol{f}(5) + \boldsymbol{f}(6) + \boldsymbol{f}(7)\end{array}\right)$$

Equation 2.4

All the other components (***F(1)*** through ***F(M−1)***) are called ***AC components***. The names were derived from an analogy with electrical systems, the DC component being

comparable to direct current and the AC component being comparable to alternative current.

We've been looking at the one-dimensional case to keep things simple, but to represent images, we need two dimensions. The two-dimensional DCT is expressed as follows:

Let $\boldsymbol{f}(r,s)$ be the pixel value at row r and column s of a bitmap. $\boldsymbol{F}(u,v)$ is the coefficient of the frequency component at *(u,v),* where $0 \le r,u \le M-1$ and $0 \le s,v \le N-1$.

$$\boldsymbol{F}(u,v) = \sum_{r=0}^{M-1} \sum_{s=0}^{N-1} \frac{2C(u)C(v)}{\sqrt{MN}} \boldsymbol{f}(r,s) \cos\left(\frac{(2r+1)u\pi}{2M}\right) \cos\left(\frac{(2s+1)v\pi}{2N}\right)$$

$$\text{where } C(\delta) = \frac{\sqrt{2}}{2} \text{ if } \delta = 0 \text{ otherwise } C(\delta) = 1$$

Equation 2.5

key equation

(Matrices are assumed to be treated in row-major order – row-by-row rather than column-by-column.) Equation 2.5 serves as an effective procedure for computing the coefficients of the frequency components from a bitmap image. (Note that since *C(u)* and *C(v)* do not depend on the indices for the summations, you can move the factor $\frac{2C(u)C(v)}{\sqrt{MN}}$ outside the nested summation. You'll sometimes see the equation written in this alternative form.)

You can think of the DCT in two equivalent ways. The first way is to think of the DCT as taking ***a function over the spatial domain*** – function $\boldsymbol{f}(r,s)$ – and returning a function ***a function over the frequency domain*** – function $\boldsymbol{F}(u,v)$. Equivalently, you can think of the DCT as taking a bitmap image in the form of a matrix of color values – $\boldsymbol{f}(r,s)$ – and returning the frequency components of the bitmap in the form of a matrix of coefficients – $\boldsymbol{F}(u,v)$. The coefficients give the amplitudes of the frequency components.

Rather than being applied to a full $M \times N$ image, the DCT is generally applied to 8 × 8 pixel subblocks (for example, as a key step in JPEG compression), so our discussion will be limited to images in these dimensions. An enlarged 8 × 8 pixel image is shown in Figure 2.24, and its corresponding bitmap values (matrix ***f***) and amplitudes of the frequency components computed by the DCT (matrix ***F***) are given in Table 2.1 and Table 2.2, respectively.

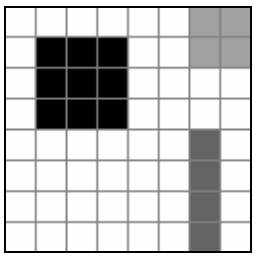


Figure 2.24 8 × 8 bitmap image. Pixels outlines are not part of image.

255	255	255	255	255	255	159	159
255	0	0	0	255	255	159	159
255	0	0	0	255	255	255	255
255	0	0	0	255	255	255	255
255	255	255	255	255	255	100	255
255	255	255	255	255	255	100	255
255	255	255	255	255	255	100	255
255	255	255	255	255	255	100	255

Table 2.1 Color values for image in Figure 2.24

1628	-61	39	234	173	-128	171	22
-205	-163	74	222	1	74	-30	111
81	150	-95	-82	-42	-11	-6	-53
188	231	-135	-188	-53	-36	-2	-103
96	71	-42	-78	-32	2	-17	-32
25	-42	25	3	-14	27	-26	19
70	15	-6	-51	-17	7	-16	-13
94	72	-38	-87	-18	-14	-4	-40

Table 2.2 Amplitudes of frequency components for image in Figure 2.24

The discrete cosine transform is invertible. By this we mean that given the amplitudes of the frequency components as ***F***, we can get color values for the bitmap image as ***f***. This relationship is described in the following equation:

Let $\boldsymbol{F}(u,v)$ is the coefficient of the frequency component at *(u,v)*.
$\boldsymbol{f}(r,s)$ is the pixel value at row *r* and column *s* of a bitmap, where $0 \le r,u \le M-1$ and $0 \le s,v \le N-1$.

$$\boldsymbol{f}(r,s) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \frac{2C(u)C(v)}{\sqrt{MN}} \boldsymbol{F}(u,v) \cos\left(\frac{(2r+1)u\pi}{2M}\right) \cos\left(\frac{(2s+1)v\pi}{2N}\right)$$

$$\text{where } C(\delta) = \frac{\sqrt{2}}{2} \text{ if } \delta = 0 \text{ otherwise } C(\delta) = 1$$

key equation

Equation 2.6

In essence, the equation states that the bitmap is equal to a sum of *M*N* weighted frequency components.

The DCT basis functions for an 8 × 8 image are pictured in Figure 2.25. Each "block" at position (u,v) of the matrix is in fact a graph of the function $\cos\left(\frac{(2r+1)u\pi}{16}\right)\cos\left(\frac{(2s+1)v\pi}{16}\right)$ at discrete positions $r = [0\ 7]$ and $s = [0\ 7]$. ***F(0,0)***, the DC component, is in the upper left corner. The values yielded by the function are pictured as grayscale values in these positions. The thing for you to understand is that any 8 × 8 pixel block of grayscale values – like the one pictured in Figure 2.24 – can be recast as a sum of frequency components pictured in Figure 2.25. You just have to know "how much" of each frequency component to add in. That is, you need to know the coefficient by which to multiply each frequency component, precisely what the DCT gives you in ***F(u,v)*** (as in Table 2.2). In the case of color images represented in RGB color mode, the DCT can be done on each of the color components individually. Thus,

the process is the same as what we described for grayscale images, except that you have three 8 × 8 blocks of data on which to do the DCT – one for red, one for green, and one for blue.

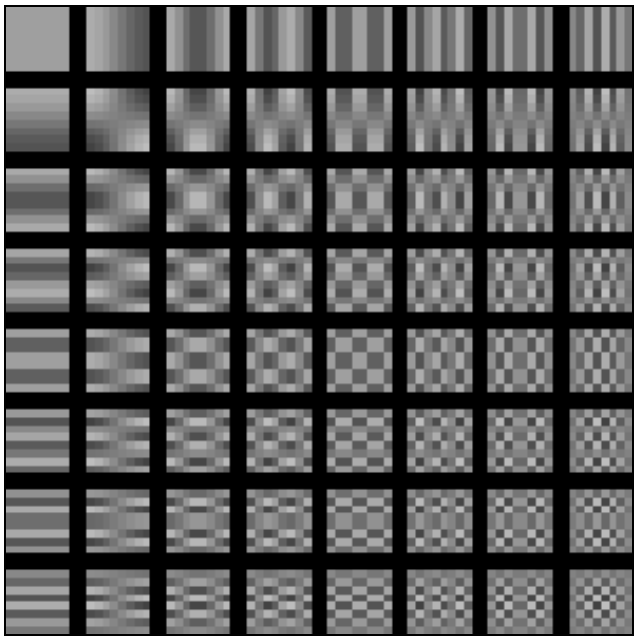


Figure 2.25 Base frequencies for discrete cosine transform

Let's complete this explanation with one more example, this time a two-dimensional image. Consider the top leftmost 8 × 8 pixel area of the sparrow picture in Figure 2.8. Figure 2.26 shows a close-up of this area, part of the sidewalk in the background. Figure 2.27 shows the graph of the pixel data over the spatial domain. Figure 2.28 captures the same information about the digital image, but represented in the frequency domain. This graph was generated by the DCT. The DC component is the largest. It may be difficult for you to see at this scale, but there are other non-zero frequency components – e.g., at positions (1,1) and (1,3) (with array indices starting at 0).

Figure 2.26 Close-up of 8 × 8 pixel area (sidewalk) from sparrow picture

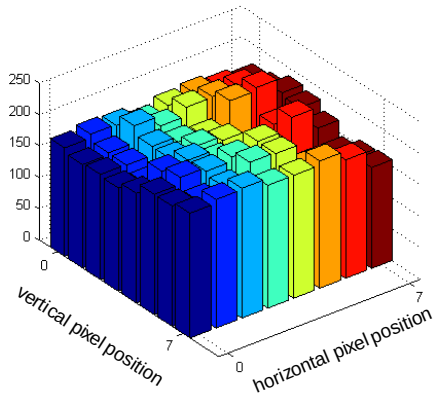


Figure 2.27
Graph of pixel values over the spatial domain

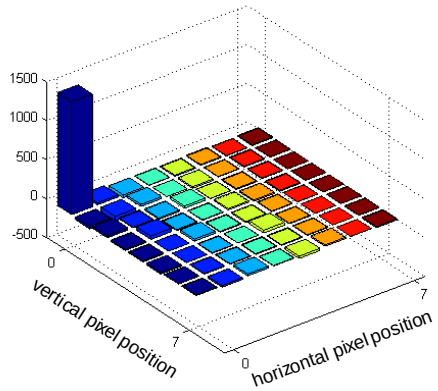


Figure 2.28
Graph of amplitudes over the frequency domain

2.5 Aliasing

2.5.1 Blurriness and Blockiness

Supplements on aliasing in sampling:

interactive tutorial

worksheet

The previous section showed that one way to understand frequency in the context of digital images is to think of the color values in the image bitmap as defining a surface. This surface forms a two-dimensional wave. A complex waveform such as the surface shown in Figure 2.11 can be decomposed into regular sinusoidal waves of various frequencies and amplitudes. In reverse, these sinusoidal waves can be summed to yield the complex waveform. This is the context in which the Nyquist theorem can be understood. Once we have represented a bitmap in the frequency domain, we can determine its highest frequency component. If the sampling rate is not at least twice the frequency of the highest frequency component, aliasing will occur.

Let's try to get an intuitive understanding of the phenomenon of aliasing in digital images. Look again at the picture in Figure 2.1 and think of it as a real-world scene that you're going to photograph. Consider just the horizontal dimension of the image. Imagine that this picture has been divided into sampling areas so that only 15 samples are taken across a row – an unrealistically low sampling rate, but it serves to make the point. If the color changes even just one time within one of the sample areas, then the two colors in that area cannot both be represented by the sample. This implies that the image reconstructed from the sample will not be a perfect reproduction of the original scene, as you can see in Figure 2.2. Mathematically speaking, the spatial frequencies of the original scene will be aliased to lower frequencies in the digital photograph. Visually, we perceive that when all the colors in a sampling area are averaged to one color, the reconstructed image looks blocky and the edges of objects are jagged. This observation seems to indicate that you'd need a very high sampling rate – i.e., large pixel dimensions – to capture a real-world scene with complete fidelity. Hypothetically, that's true for most scenes. Fortunately, however, the human eye isn't going to notice a little loss of detail. The pixel dimensions offered by most digital cameras these days provide more than enough detail for very crisp, clear images.

2.5.2 Moiré Patterns

Another interesting example of aliasing, called the ***moiré effect*** or ***moiré pattern***, can occur when there is a pattern in the image being photographed, and the sampling rate for the digital image is not high enough to capture the frequency of the pattern. If the pattern is not sampled at a rate that is at least twice the rate of repetition of the pattern, then a different pattern will result in the reconstructed image. In the image shown in Figure 2.29, the color changes at a perfectly regular rate, with a pattern that repeats five times in the horizontal direction. What would happen if we sampled this image five times, at regularly spaced intervals? Depending on where the sampling started, the resulting image would be either all black or all white. If we sample more than ten times, however – more than twice per repetition of the pattern – we will be able to reconstruct the image faithfully. This is just a simple application of the Nyquist theorem.

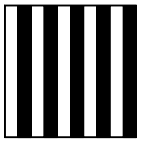


Figure 2.29 A simple image with vertical stripes

More visually interesting moiré effects can result when the original pattern is more complex and the pattern is tilted at an angle with respect to the sampling. Imagine the image that would result from tilting the original striped picture and then sampling in the horizontal and vertical directions, as shown in Figure 2.30. The red grid shows the sampling blocks. Assume that if more than ½ a sampling block is filled with black from the original striped image, then that block becomes black. Otherwise, it is white. The pattern in the reconstructed image is distorted in a moiré effect. Once you know what the moiré effect is, you start seeing it all around you. You can see it any time one pattern is overlaid on another – like the shimmering effect of a sheer curtain folded back on itself, or the swirls resulting from looking through a screen at a closely woven wicker chair.

Figure 2.30 Sampling that can result in a moiré pattern

Moiré patterns can result both when a digital photograph is taken and when a picture is scanned in to create a digital image, because both these processes involve choosing a sampling rate. Figure 2.31 shows a digital photograph of a computer bag where a moiré pattern is evident, resulting from the sampling rate and original pattern being "out of synch." Figure 2.32 shows a close-up of the moiré pattern. Figure 2.33 shows a close-up of the pattern as it should look. If you get a moiré effect when you take a digital photograph, you can try tilting the camera at a different angle or changing the

focus slightly to get rid of it. This will change the sampling orientation or sampling precision with respect to the pattern.

Figure 2.31 Moiré pattern in digital photograph

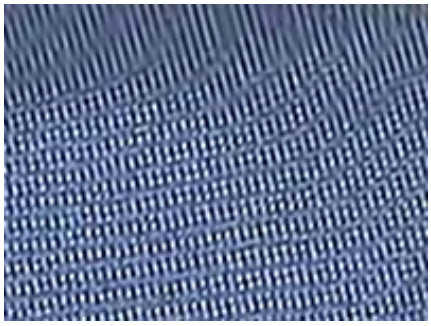


Figure 2.32 Close-up of moiré pattern

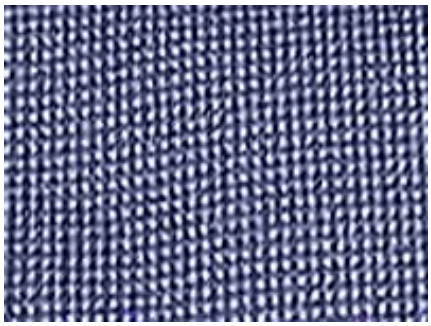


Figure 2.33 True pattern

Moiré patterns occur in digital photography because it is based on discrete samples. If the samples are taken "off beat" from a detailed pattern in the subject being photographed, an alias of the original pattern results. But this does not fully explain the source of the problem. Sometimes aliasing in digital images manifests itself as small areas of incorrect colors or artificial auras around objects, which can be referred to as ***color aliasing***, ***moiré fringes***, ***false coloration***, or ***phantom colors***. To understand what causes this phenomenon, you have to know a little about how color is perceived and recorded in a digital camera.

When a photograph is taken with a traditional analog camera, film that is covered with silver-laden crystals is exposed to light. There are three layers on photographic film, one sensitive to red, one to green, and one to blue light (assuming the use of RGB color). At each point across a continuous plane, all three color components are sensed

simultaneously. The degree to which the silver atoms gather together measures the amount of light to which the film is exposed.

We have seen that one of the primary differences between analog and digital photography is that analog photography measures the incident light continuously across the focal plane, while digital photography samples it only at discrete points. Another difference is that it is more difficult in a digital camera to sense all three color components – red, green, and blue – at each sample point. These constraints on sampling color and the use of interpolation to "fill in the blanks" in digital sampling can lead to color aliasing. Let's look at this more closely.

Many current digital cameras use ***charge-coupled device*** (***CCD***) technology to sense light and thereby color. (CMOS – complementary metal-oxide semiconductor – is an alternative technology for digital photography, but we won't discuss that here.) A CCD consists of a two-dimensional array of photosites. Each photosite corresponds to one sample – i.e., one pixel in the digital image. The number of photosites determines the limits of a camera's resolution. To sense red, green, or blue at a discrete point, the sensor at that photosite is covered with a red, green, or blue color filter. But the question is: Should all three color components be sensed simultaneously at each photosite, should they be sensed at different moments when the picture is taken, or should only one color component per photosite be sensed?

There are a variety of CCD designs in current technology, each with its own advantages and disadvantages. (1) The incident light can be divided into three beams. Three sensors are used at each photosite, each covered with a filter that allows only red, green, or blue to be sensed. This is an expensive solution and creates a bulkier camera. (2) The sensor can be rotated when the picture is taken so that it takes in information about red, green, and blue light in succession. The disadvantage of this method is that the three colors are not sensed at precisely the same moment, so the subject being photographed needs to be still. (3) A more recently-developed technology (Foveon X3) uses silicon for the sensors in a method called *vertical stacking*. Because different depths of silicon absorb different wavelengths of light, all three color components can be detected at one photosite. This technology is gaining popularity. (4) A less-expensive method of color detection uses an array like the one shown in Figure 2.34 to detect only one color component at each photosite. Interpolation is then used to derive the other two other color components based on information from neighboring sites. It is the interpolation that can lead to color aliasing.

G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G

Figure 2.34 Bayer color filter array

In the 4 × 4 array shown in the figure, the letter in each block indicates which color is to be detected at each site. The pattern shown here is called a ***Bayer color filter array***, or simply a ***Bayer filter***. (It's also possible to use a cyan-magenta-yellow

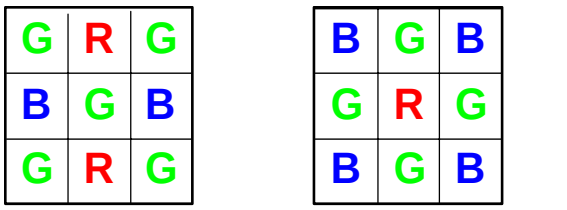
combination.) You'll notice that there are twice as many green sensors compared to blue or red. This is because the human eye is more sensitive to green and can see more fine-grained changes in green light. The array shown in the figure is just a small portion of what would be on a CCD. Each block in the array represents a photosite, and each photosite has a filter on it that determines which color is sensed at that site.

The interpolation algorithm for deriving the two missing color channels at each photosite is called ***demosaicing***. A variety of demosaicing algorithms have been devised. A simple ***nearest neighbor algorithm*** determines a missing color *c* for a photosite based on the colors of the nearest neighbors that have the color *c*. For the algorithm given below, assuming a CCD array of dimensions $m \times n$, the nearest neighbors of photosite *(i,j)* are sites $(i-1, j-1), (i-1, j), (i-1, j+1), (i, j-1), (i, j+1), (i+1, j-1), (i+1, j), (i+1, j+1)$ where $0 \le i \le m$ and $0 \le j \le n$ (disregarding boundary areas where neighbors may not exist). The nearest neighbor algorithm is given as Algorithm 2.1.

```
algorithm nearest_neighbor
{
  for each photosite (i,j) where the photosite detects color cx {
    for each cy ∈ {red,green,blue} such that cy ≠ cx {
      S = the set of nearest neighbors of site (i,j) that have color cy
      set the color value for cy at site (i,j) equal to the average of the color values of cy at
the sites in S
  }
}
```

Algorithm 2.1 Nearest neighbor algorithm

With this algorithm, there may be either two or four neighbors involved in the averaging, as shown in Figure 2.35a and Figure 2.35b.



a. Determining R or B from the center G photosite entails an average of two neighboring sites.

b. Determining B from the center R photosite entails an average of four neighboring sites diagonal from R.

Figure 2.35 Photosite interpolation

This nearest neighbor algorithm can be fine-tuned to take into account the rate at which colors are changing in either the vertical or horizontal direction, giving more weight to small changes in color as the averaging is done. Other standard interpolation methods – linear, cubic, cubic spline, etc. – can also be applied, and the region of nearest neighbors can be larger than 3×3 or a shape other than square.

The result of the interpolation algorithm is that even though only one sensor for one color channel is used at each photosite, the other two channels can be derived to yield full RGB color. This method works quite well and is used in many digital cameras with

CCD sensors. However, interpolation by its nature cannot give a perfect reproduction of the scene being photographed, and occasionally color aliasing results from the process, detected as moiré patterns, streaks, or spots of color not present in the original scene. A simple example will show how this can happen. Imagine that you photograph a white line, and that line goes precisely across the sensors in the CCD as shown in Figure 2.36a. If there is only black on either side of the line, then averaging the neighboring pixels to get the color channels not sensed at the photosites covered by the line always gives an average of 0. That is, no other color information is added to what is sensed at the photosites covered by the line, so each photosite records whatever color is sensed there. The result is the line shown in Figure 2.36b. Generally speaking, when a small area color can be detected by only a few photosites, the neighboring pixels don't provide enough information so that the true color of this area can be determined by interpolation. This situation can produce spots, streaks, or fringes of aliased color.

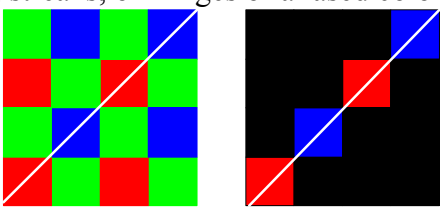


Figure 2.36 A situation that can result in color aliasing

Some cameras use descreening or anti-aliasing filters over their lenses – effectively blurring an image slightly to reduce the color aliasing or moiré effect. The filters remove high frequency detail in the image that can lead to color aliasing, but this sacrifices a little of the image's clarity. Camera manufacturers make this choice in the design of their cameras – to include the anti-aliasing filter or not – and many high quality cameras do not have the filters because the manufacturers assume that sharp focus is more important to most photographers when weighed against occasional instances of color aliasing or moiré patterns.

2.5.3 The "Jaggies"

You may have heard the term ***aliasing*** used to describe the "jaggies" along lines or edges that are drawn at an angle across a computer screen. This type of aliasing occurs during rendering rather than sampling and results from the finite resolution of computer displays. A line as an abstract geometric object is made up of points, and since there are infinitely many points on a plane, you can think of a line as being infinitely narrow. A line on a computer screen, on the other hand, is made up of discrete units – pixels. If we assume these pixels are square or rectangular and lined up parallel to the sides of the display screen, then there is no way to align them perfectly when a line is drawn at an angle on the computer screen. This situation is illustrated in Figure 2.37. We assume for purposes of illustration that pixels are non-overlapping rectangles or squares that cover the display device entirely.

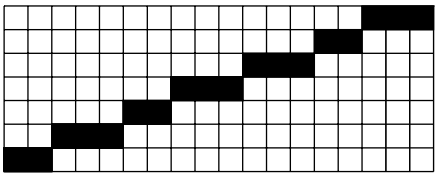


Figure 2.37 Aliasing of a one-pixel-wide line

Supplements on aliasing in rendering:

interactive tutorial

worksheet

When you draw a line in a draw or a paint program using a line tool, you click on one endpoint and then another, and the line is drawn between the two points. So that the line can be drawn on the computer display, the pixels that are colored to form the line must be determined. This requires a line-drawing algorithm such as the one given in Algorithm 2.2. Beginning at one of the line's endpoints, this algorithm moves horizontally across the display device, one pixel at a time. Given column number *x0*, the algorithm finds the integer *y0* such that (*x0, y0*) is the point closest to the line. Pixel (*x0, y0*) is then colored. The results might look like Figure 2.37. The figure and the algorithm demonstrate how a one-pixel wide line would be drawn. It is not the only algorithm for the purpose, and it does not deal with lines that are two or more pixels wide.

```
algorithm draw_line
/*Input:  x0, y0, x1, and y1, coordinates of the line's endpoints (all integers) c, color of
the line.
Output:  Line is drawn on display.*/
{
/*Note:  We include data types because they are important to understanding the
algorithm's execution. */
   int dx, dy, num_steps, i
   float x_increment, y_increment, x, y
   dx = x1 – x0
   dy = y1 – y0
   if (absolute_value(dx) > absolute_value(dy) then  num_steps = absolute_value(dx)
   else  num_steps = absolute_value(dy)
   x_increment = float(dx) / float (num_steps)
   y_increment = float (dy) / float (num_steps)
   x = x0
   y = y0
/*round(x) rounds to the closest integer.*/
   draw(round(x), round(y), c)
   for i = 0 to num_steps−1 {
      x = x + x_increment
      y = y + y_increment
      draw(round(x), round(y), c)
   }}
```

Algorithm 2.2 Algorithm for drawing a line

Supplements on aliasing in line drawing:

programming exercise

interactive tutorial

To test your understanding, think about how Algorithm 2.2 would be generalized to lines of any width. Figure 2.38 shows a line that is two pixels wide going from point

(8,1) to point (2,15). The ideal line is drawn as a dashed line between the two endpoints. To render the line in a black and white bitmap image, the line-drawing algorithm must determine which pixels are intersected by the two-pixel-wide area. The result is the line drawn in Figure 2.39. Because it is easy to visualize, we use the assumption that a pixel is colored black if at least half its area is covered by the two-pixel line. Other line-drawing algorithms may operate differently.

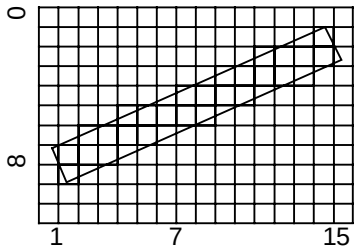


Figure 2.38 Drawing a line two pixels wide

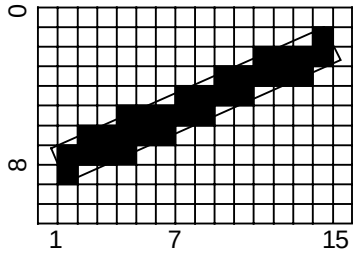


Figure 2.39 Line two pixels wide, aliased

Anti-aliasing is a technique for reducing the jaggedness of lines or edges caused by aliasing. The idea is to color a pixel with a shade of its designated color in proportion to the amount of the pixel that is covered by the line or edge. In this example, consider the two-pixel-wide line shown in Figure 2.38. If a pixel is completely covered by the line, it is colored black. If it is half covered, it is colored a shade of gray half way between black and white, and so forth. Using gradations of colors softens the edges. The edges are not sharp, as they would be ideally, but making the lines and edges perfectly straight is impossible due to the finite resolution of the image and the display device. Anti-aliasing helps to compensate for the jagged effect that results from imperfect resolution. The anti-aliased version of our example line is shown in Figure 2.40, enlarged. At normal scale, this line looks smoother than the aliased version.

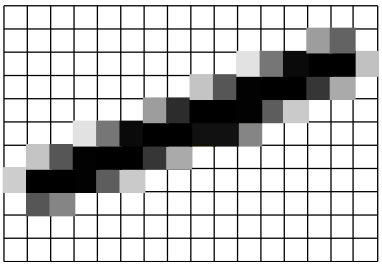


Figure 2.40 Line two pixels wide, anti-aliased

Bitmaps are just one way to represent digital images. Another way is by means of vector graphics. Rather than storing an image bit-by-bit, a vector graphic file stores a

description of the geometric shapes and colors in an image. Thus, a line can be described by means of its endpoints, a square by the length a side and a registration point, a circle by its radius and center point, etc. Vector graphics suffer less from aliasing problems than do bitmap images in that vector graphics images can be resized without loss of resolution. Let's look at the difference.

When you draw a line with a line tool in a bitmap image, the pixel values are computed for the line and stored in the image pixel by pixel. Say that the image is later enlarged by increasing the number of pixels. This process is called ***upsampling***. A simple algorithm for upsampling a bitmap image, making it twice as big, is to make four pixels out of each one, duplicating the color of the original pixel. Obviously, such an algorithm will accentuate any aliasing from the original image. Figure 2.41 shows a one-pixel wide line from a black and white bitmap image at three different enlargements, the second twice as large and the third three times as large as the first. The jagged edges in the original line look even blockier as the line is enlarged. Other more refined algorithms for upsampling can soften the blocky edges somewhat, but none completely eliminate the aliasing. Algorithms for upsampling are discussed in more detail in Chapter 3.

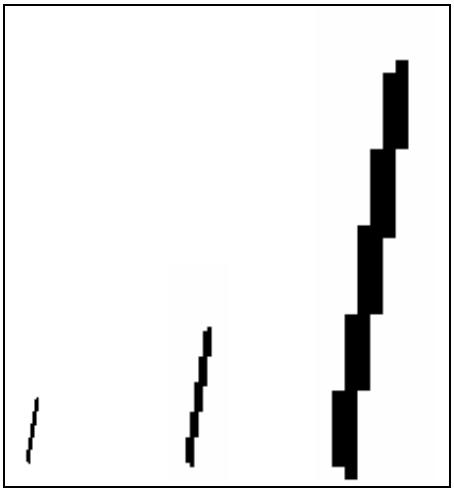


Figure 2.41 Aliasing of bitmap line resulting from enlargement

Aliasing in rendering can occur in both bitmap and vector graphics. In either type of digital image, the smoothness of lines, edges, and curves is limited by the display device. There is a difference, however, in the two types of digital imaging, and vector graphics has an advantage over bitmap images with respect to aliasing. Aliasing in a bitmap image becomes even worse if the image is resized. Vector graphic images, on the other hand, have only the degree of aliasing caused by the display device on which the image is shown, and this is very small and hardly noticeable.

Changing the size of an image is handled differently in the case of vector graphics. Since vector graphic files are not created from samples and not stored as individual pixel values, *upsampling* has no meaning in vector graphics. A vector graphic image *can* be resized for display or printing, but the resizing is done by recomputation of geometric shapes on the basis of their mathematical properties. In a vector graphic image, a line is stored by its endpoints along with its color and an indication that it is a line object. When the image is displayed, the line is rendered by a line drawing algorithm at a size relative to the image dimensions that have been chosen for the image at that

moment. Whenever the user requests that the image be resized, the pixels that are colored to create the line are recomputed using the line-drawing algorithm. This means that the jaggedness of the line will never be worse than what results from the resolution of the display device. In fact, to notice the aliasing at all when you work in a vector graphic drawing program, you need to turn off the anti-aliasing option when you view the image. Figure 2.42 shows a vector graphic line at increasing enlargements, with the "view with anti-aliasing" option turned off. The small amount of aliasing is due entirely to the resolution of the display device. The important thing to notice is that as the line gets larger, the aliasing doesn't increase. These observations would hold true for more complex vector graphic shapes as well. (When the lines are printed out on a printer with good resolution, the aliasing generally doesn't show up at all because the printer's resolution is high enough to create a smooth edge.)

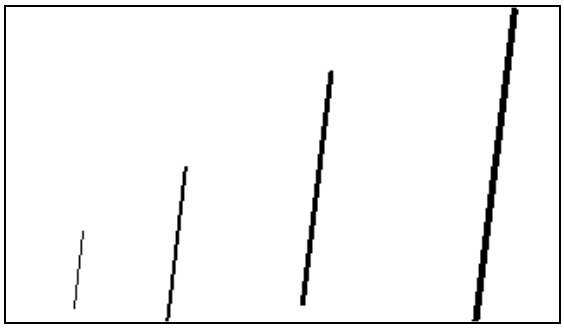


Figure 2.42 Aliasing does not increase when a vector graphic line is enlarged

In summary, vector graphics and bitmap imaging both have their own advantages. Vector graphics imaging is suitable for pictures that have solid colors and well-defined edges and shapes. Bitmap imaging is suitable for continuous tone pictures like photographs. The type of images you work with depends on your purpose.

2.6 Color

2.6.1 Color Perception and Representation

Color is both a physical and psychological phenomenon. Physically, color is composed of electromagnetic waves. For humans, the wavelengths of visible colors fall between approximately 370 and 780 nanometers, as shown in Figure 2.43 and Figure 2.44. (A nanometer, abbreviated *nm* in the figure, is 10^{-9} meters.) These waves fall upon the color receptors of the eyes, and in a way not completely understood, the human brain translates the interaction between the waves and the eyes as color perception.

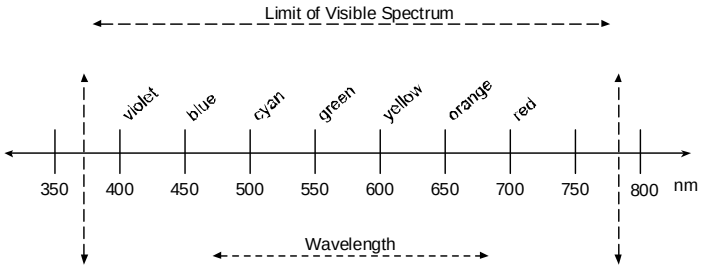


Figure 2.43 The visible spectrum

Although it is possible to create pure color composed of a single wavelength – for example, by means of a laser – the colors we see around us are almost always produced by a combination of wavelengths. The green of a book cover, for example, may look like a pure green to you, but a spectrograph will show that it is not. A spectrograph breaks up a color into its component wavelengths, producing a spectral density function P(λ). A spectral density function shows the contributions of the wavelengths λ to a given perceived color as λ varies across the visible spectrum.

Spectral density functions are one mathematical way to represent colors, but not a very convenient way for computers. One problem is that two colors that are perceived to be identical may, on analysis, produce different spectral density curves. Said the other way around, more than one spectral density curve can represent two colors that look the same. If we want to use a spectral density curve to tell a computer to present a particular shade of green, which "green" spectral density curve is the best one to use?

Aside: The differences between the power, energy, luminance, and brightness of a light can be confusing. Two colored lights can be of the same wavelength but of different power. A light's *power*, or *energy per unit time*, is a physical property *not* defined by human perception. Power is related to brightness in that if two lights have the same wavelength but the first has greater power, then the first will appear brighter than the second.. ***Brightness*** is a matter of subjective perception and has no precise mathematical definition. ***Luminance*** has a mathematical definition that relates a light's wavelength and power to how bright it is perceived to be. Interestingly, lights of equal power but different wavelengths do not appear equally bright. The brightest wavelengths are about 550 nm. Brightness decreases from there as you moved to longer or shorter wavelengths. In general, the more luminant something is, the brighter it appears to be. However, keep in mind that brightness is in the eye of the beholder – a matter of human perception – while luminance has a precise definition that factors in power, wavelength, and the average human observer's sensitivity to that wavelength.

It is possible to represent a color by means of a simpler spectral density graph. (This is basically how color representation is done in the HSV and HLS color models, as will be explained below.) That is, each color in the spectrum can be characterized by a unique graph that has a simple shape, as illustrated in Figure 2.45. The graph for each color gives the color's ***dominant wavelength***, equivalent to the ***hue***; its ***saturation*** (i.e., color purity); and its ***luminance***. The dominant wavelength is the wavelength at the spike in the graph. The area beneath the curve indicates the luminance *L*. (This "curve"

is a rectangular area with a rectangular spike.) Saturation *S* is the ratio of the area of the spike to the total area. More precisely with regard to Figure 2.45

$$L = (d - a)e + (f - e)(c - b)$$

Equation 2.7

$$S = \frac{(f - e)(c - b)}{L}$$

Equation 2.8

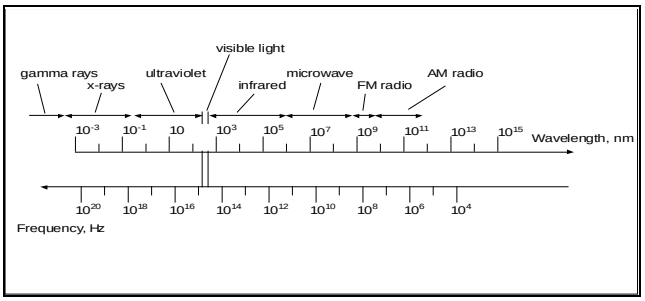


Figure 2.44 Electromagnetic spectrum

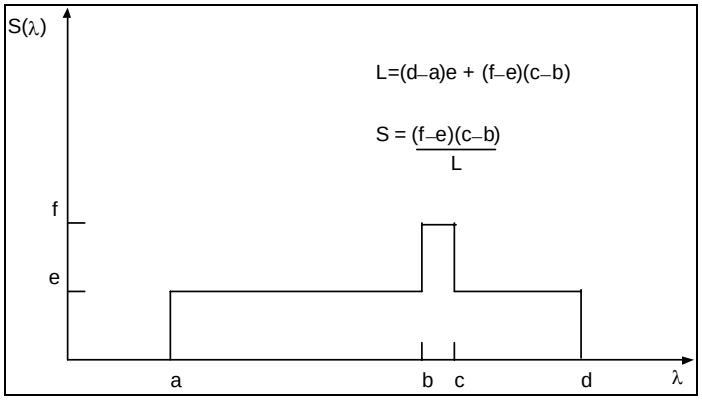


Figure 2.45 Spectral density graph showing hue, saturation, and lightness

This representation has intuitive appeal, since it is natural to consider first a color's essential hue and then consider varying shades and tones. However, the dimensions of hue, saturation, and brightness do not correspond very well to the way computer monitors are engineered. (We use the terms *monitor* and *display* interchangeably to refer to the viewing screen of a computer.) An alternative way to look at a color is as a combination of three primaries. Cathode ray tube (CRT) monitors, for example, display colored light through a combination of red, green and blue phosphors that light up at varying intensities when excited by an electron beam. Similarly, liquid crystal display (LCD) panels display color with neighboring pixels of red, green, and blue that are either lit up or masked by the liquid crystals.

So what is the best way to model color for a computer? There is no simple answer, since different models have advantages in different situations. In the discussion that follows, we will look at ***color models*** mathematically and find a graphical way to compare their expressiveness.

2.6.2 RGB Color Model

One method to create a wide range of colors is by varying combinations of three primary colors. Three colors are primary with respect to each other if no one of them can be created as a combination of the other two. Red, green, and blue are good choices as primary colors because the cones of the eyes – the colors receptors – are especially sensitive to these hues.

$$C = rR + gG + bB$$

Equation 2.9

where *r*, *g*, and *b* indicate the relative amounts of red, green, and blue energy respectively. *R*, *G*, and *B* are constant values based on the wavelengths chosen for the red, green and blue components. The values r, g, and b are referred to as the values of the RGB ***color components*** (also called ***color channels*** in application programs).

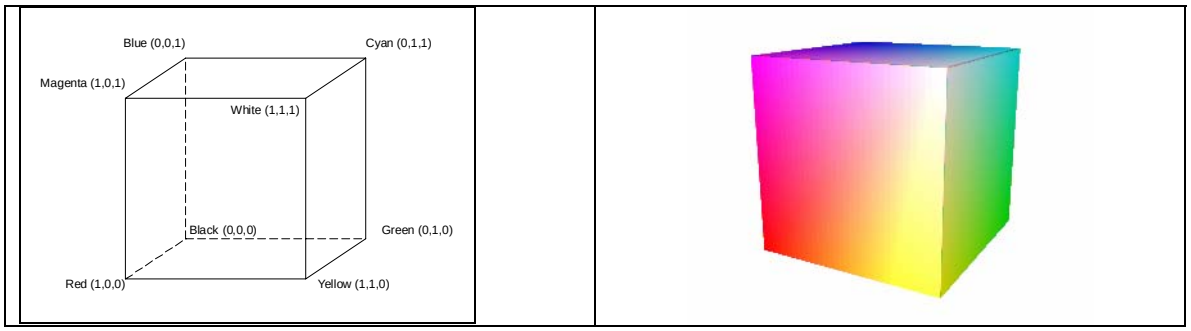


Figure 2.46 RGB color cube

The color space for the RGB color model is easy to depict graphically. Let R, G, and G correspond to three axes in three-dimensional space. We will normalize the relative amounts of red, green, and blue in a color so that each value varies between 0 and 1. This color space is shown in Figure 2.46. The origin (0, 0, 0) of the RGB color cube corresponds to black. White is the value (1, 1, 1). The remaining corners of the cube correspond to red, green, blue, and their complementary colors – cyan, magenta, and yellow, respectively. Others colors are created at values between 0 and 1 for each of the components. For example, (1, 0.65, 0.15) is light orange, and (0.26, 0.37, 0.96) is a shade of blue. Shades of gray have equal proportions of red, green, and blue and lie along the line between (0, 0, 0) and (1, 1, 1). Notice that if you decrease each of the values for light orange in (1, 0.65, 0.15) but keep them in the same proportion to each other, you are in effect decreasing the luminance of the color, which is like adding in more black. The color moves from a light orange to a muddy brown. You can't increase the luminance of this color and maintain the proportions, because one of the components is already 1, the maximum value. The color is at 100% luminance. On the other hand, the color (0.32, 0.48, 0.39) is a shade of green not at full brightness. You can multiply each component by 2 to get (0.64, 0.96, 0.78), a much lighter shade of green.

You may want to note that in mathematical depictions of the RGB color model, it is convenient to allow the three color components to range between 0 and 1. However, the corresponding RGB color mode in image processing programs is more likely to have values ranging between 0 and 255, since each of the three components is captured in eight bits. What is important is the relative amounts of each component, and how large these amounts are with respect to the maximum possible values. For example, the light

orange described as (1, 0.65, 0.15) above would become (255, 166, 38) in an RGB mode with maximum values of 255.

It's interesting to note that grayscale values fall along the RGB cube's diagonal from (0,0,0) to (1,1,1). All grayscale values have equal amounts of R, G, and B. When an image is converted from RGB color to grayscale in an image processing program, the the equation can be used for the conversion of each pixel value. This equation reflects the fact that the human eye is most sensitive to green and least sensitive to blue.

key equation

Let an RGB color pixel be given by (R,G,B), where R, G, and B are the red, green, and blue color components, respectively. Then the corresponding grayscale value is given by (L,L,L), where

$$L = 0.30R + 0.59G + 0.11B$$

Equation 2.10

Since all three color components are equal in a gray pixel, only one of the three values needs to be stored. Thus a 24-bit RGB pixel can be stored as an 8-bit grayscale pixel.

2.6.3 CMY Color Model

Like the RGB color model, the ***CMY color model*** divides a color into three primaries, but using a subtractive rather than an additive color creation process. The CMY model can be depicted in a unit cube similar to the RGB model. The difference is that the origin of the cube is white rather than black, and the value for each component indicates how much red, green and blue are subtracted out, effectively combining the color components cyan, magenta, and yellow, their respective complements.

Assuming that each of the three RGB (or CMY) components is a value between 0 and 1, the corresponding CMY components can be computed as follows:

key equation

For a pixel represented in RGB color, the red, green, and blue color components are, respectively, R, G, and B. Then the equivalent C, M, and Y color components are given by

$$C = 1 - R$$

$$M = 1 - G$$

$$Y = 1 - B$$

Similarly, RGB values can be computed from CMY values with

$$R = 1 - C$$

$$G = 1 - M$$

$$B = 1 - Y$$

(The values can be given in the range of [0 255] or normalized to [0 1].)

The CMY model, used in professional four-color printed processes, indicates how much cyan, magenta, and yellow ink is combined to create color. Theoretically, the maximum amount of cyan, magenta, and yellow ink should combine to produce black, but in fact they produce a dark muddy brown. In practice, the four-color printing process used in professional presses adds a fourth component, a pure black ink, for greater clarity

and contrast. The amount of *K*, or black, can be taken as the smallest of the *C*, *M*, and *Y* components in the original CMY model. Thus the CMYK model is defined as follows:

key equation

For a pixel represented in the CMY color model, the cyan, magenta, and yellow color components are, respectively, *C*, *M*, and *Y*. Let *K* be the minimum of *C*, *M*, and *Y*. Then the equivalent color components in the CMYK model, $C_{new}, M_{new}, Y_{new}$, and *K* are given by

$$K = \min(C, M, Y)$$
$$C_{new} = C - K$$
$$M_{new} = M - K$$
$$Y_{new} = Y - K$$

(The definition above theoretically gives the values for CMYK. However, in practice, other values are used due to the way in which colored inks and paper interact.)

2.6.4 HSV and HLS Color Models

Instead of representing color by three primary color components, it is possible to speak of a color in terms of its hue (i.e., the essential color), its lightness (or value or luminance), and its saturation (i.e., the purity of the color). Both the ***HSV color model*** (also called HSB) and the ***HLS model*** represent color in this manner. Geometrically, the HSV color space is a distortion of the RGB space into a kind of three-dimensional diamond called a *hexacone*. Picture turning the RGB cube around and tilting it so that you are looking straight into the origin (white/black) with the remaining corners visible – two on the top, two on the bottom, one on the left, and one on the right, as shown in Figure 2.47. Imagine this as a flat, two-dimensional hexagon where the center white/black point is connected by a line to each vertex, as shown in Figure 2.48. The six primary colors and their complements are at the outside vertices of this shape. Now imagine expanding this into three dimensions again by pulling down on the center point. You have created the HSV color space, as shown in Figure 2.49.

To see how this shape captures the HSV color model, draw an imaginary circle that touches all the vertices of the hexacone's base. The hue is represented by a position on this circle given in degrees, from 0 to 360, with red conventionally set at 0. As the hue values increase, you move counterclockwise through yellow, green, cyan, etc. Saturation is a function of the color's distance from the central axis (i.e., the value axis). The farther a color is from this axis, the more saturated the color. The value axis lies from the black point of the hexacone through the center of the circle, with values ranging from 0 for black to 1 for white, where 0 is at the tip and 1 is on the surface of the hexacone. For example, (58°, 0.88, 0.93) is a bright yellow.

The HLS color model is essentially the same. To create the HLS color space from the HSV space (and hence from RGB), go through the same steps illustrated in Figure 2.47, Figure 2.48, and Figure 2.49. Then take a mirror image of the shape in Figure 2.49 and connect it to the top, as in Figure 2.50. Hue and saturation are given as before, but now lightness varies from 0 at the black tip to 1 at the white tip of the double cones.

The distortion of the RGB color space to either HSV or HLS is a non-linear transformation. In other words, to translate from RGB to HSV, you can't simply multiply each of the R, G, and B components by some coefficient. Algorithm 2.3 shows how to translate RGB to HSV. Algorithm 2.4 goes from RGB to HLS. The inverse algorithms are left as an exercise.

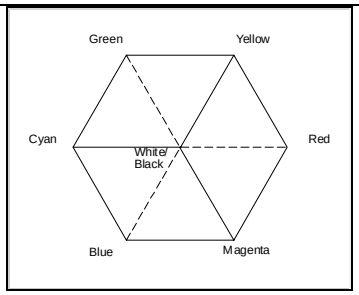


Figure 2.47 RGB color cube viewed from the top

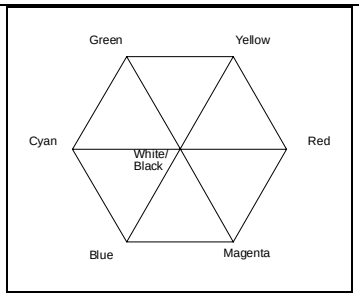


Figure 2.48 RGB color cube collapsed to 2D

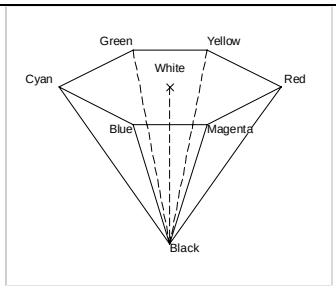


Figure 2.49 HSV color space, a hexacone

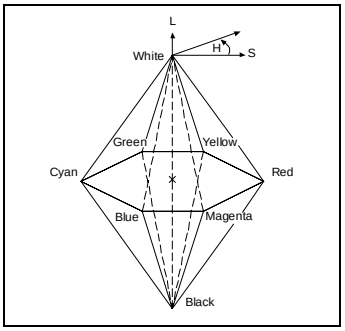


Figure 2.50 HLS Color Space

```
algorithm RGB_to_HSV
/*   Input:  r, g, and b, each real numbers in the range [0…1].
Output:  h, a real number in the range of [0…360), except if s = 0, in which case h is
undefined.  s and v are real numbers in the range of [0…1].*/
{
  max = maximum(r,g,b)
  min = minimum(r,g,b)
```

```
  v = max
  if max ≠ 0 then s = (max – min)/max
  else s = 0
  if s == 0 then h = undefined
  else {
    diff = max –  min
    if r == max then h = (g – b) / diff
    else if g == max then h = 2 + (b – r) / diff
    else if b == max then h = 4 + (r – g) / diff
    h = h * 60
    if h < 0 then h = h + 360
  }
}
```

Algorithm 2.3
RGB to HSV

```
algorithm RGB_to_HLS
/*  Input  r, g, and b, each real numbers in the range [0…1]
Output:  h, a real number in the range of [0…360], except if s = 0, in which case h is
undefined.. t and  s  are real numbers in the range of [0…1].*/
{
  max = maximum(r,g,b)
  min = minimum(r,g,b)
  t = average(max, min)
  if max == min then s = 0
  else {
    sum = max + min
    diff = max – min
    if t ≤ 0.5 then s = diff / sum
    else s = diff / (2 – max + min)
    r_temp = (max – r) / diff
    g_temp = (max – g) / diff
    b_temp = (max – b) / diff
    if r == max then h = b_temp – g_temp
    else if g == max then h = 2 + r_temp – b_temp
    else if b == max then h = 4 + g_temp – r_temp
    h = h * 60
    if h < 0 then h = h + 360
  }
}
```

Algorithm 2.4 RGB to HLS

Supplements on color model conversions:

programming exercise

2.6.5 Luminance and Chrominance Color Models

Another way to specify a color is to capture all the luminance information in one value and put the color (i.e., ***chrominance***) information in the other two values. The YIQ model is one example that takes this approach.

The YIQ model is a simple translation of the RGB model, separating out the information in a way that is more efficient for television broadcasting. In the early days of color television, both black and white and color signals had to be transmitted because not all consumers had color television sets. It was convenient to consolidate all of the "black and white" information – which is luminance – in one of the three components and capture all the color information in the other two. That way, the same transmission worked for both kinds of consumers. A linear transformation of the values makes this possible. Specifically,

key equation

For a pixel represented in RGB color, let the red, green, and blue color components be, respectively, R, G, and B. Then the equivalent Y, I, and Q color components in the YIQ color model are given by

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

(Note that the values in the transformation matrix depend upon the particular choice of primaries for the RGB model.)

Equation 2.11

Y is the luminance component, and I and Q are chrominance. The inverse of the matrix above is used to convert from YIQ to RGB. The coefficients in the matrix are based on primary colors of red, green, and blue that are appropriate for the standard National Television System Committee (NTSC) RGB phosphor.

YIQ is the model used in U.S. commercial television broadcasting. Isolating luminance in one of the three terms has a further advantage, aside from its advantage in color/black and white broadcasting. Perhaps surprisingly, human vision is more sensitive to differences in luminance than differences in color. Therefore, it makes more sense to give a more finely-nuanced representation of the luminance component than of the chrominance. In practical terms, this means that we don't need as many bits – and therefore as much bandwidth – for the transmission of the I and Q components relative to the Y component. It would not be possible to make this savings in bandwidth using the RGB model because in RGB the luminance is not a separate element but instead is implicit in the combination of the three components.

The YUV color model, originally used in the European PAL analog video standard, is also based upon luminance and chrominance. The YCbCr model is closely related to the YUV, with its chrominance values scaled and shifted. YCbCr is used in JPEG and MPEG compression. These compression techniques benefit from the separation of luminance from chrominance since some chrominance information can be sacrificed during compression without visible loss of quality in photographic images.

This is called *chroma subsampling*. Chapters 3 and 6 will explain chroma subsampling in more detail.

2.6.6 CIE XYZ and Color Gamuts

The RGB color model has the advantage of relating well to how the human eye perceives color and how a computer monitor can be engineered to display color, in combinations of red, green, and blue light. Its main disadvantage is that there exist visible colors that cannot be represented with positive values for each of the red, green, and blue components.

It may seem that the obvious way to generate all possible colors is to combine all possible intensities of red, green, and, blue light – or at least enough of these values at discrete intervals to give us millions of choices. For example, we could vary the intensity of the red light through 256 evenly-spaced increments, and the same for green and blue. This would give us 256*256*256=16,777,216 colors. That must cover the range of possible colors, right? Wrong. In fact, there exist colors outside the range of those we can create in RGB, colors that we *cannot* capture with any combination of red, green, and blue. But how does anyone know this?

We know this by an experiment called *color matching*. In this experiment, human subjects are asked to compare pure colors projected onto one side of a screen to composite colors projected beside them. The pure colors are created by single wavelength light. The composite colors are created by a combination of red, green, and blue light, and the amounts of the three components are called the *tristimulus values*. For each pure color presented to the human observer, these colors ranging through all of the visible spectrum, the observer is asked to adjust the relative intensities of the red, green, and blue components in the composite color until the match is as close as possible. It turns out that there are pure colors in the visible spectrum that cannot be reproduced by positive amounts of red, green, and blue light. In some cases, it is necessary to "subtract out" some of the red, green, or blue in the combined beams to match the pure color. (Effectively, this can be done by adding red, green, or blue to the pure color until the two light samples match.) This is true no matter what three visible primary colors are chosen. No three visible primaries can be linearly combined to produce all colors in the visible spectrum.

> Aside: There is no fixed shade of red, green, and blue that must be used for RGB. The only requirement is that they be primary colors relative to each other in that no two can be combined in any manner to create the third. The actual wavelengths of *R*, *G*, and *B* chosen for particular monitors depend on the characteristics of the display itself. For example, for a CRT monitor, the persistence of the phosphors in the monitor will determine in part the choice of *R*, *G*, and *B*. *Persistence* refers to the length of time the phosphors continue to glow after excitation by an electron beam.

The implication of this experiment is that no computer monitor that bases its color display on combinations of red, green, and blue light can display all visible colors. The range of colors that a given monitor can display is called its ***color gamut***. Since computer monitors may vary in their choice of basic red, green, and blue primaries, two computer monitors based on RGB color can still have different gamuts. By similar reasoning, the gamut of a color system based on the CMYK model will vary from one

based on RGB. In practical terms, this means that there will be colors that you can represent on your computer monitor but you cannot print, and vice versa.

It would be useful to have a mathematical model that captures all visible colors. From this model, we could create a color space in which all other color models could be compared. The first step in the direction of a standard color model that represents all visible colors was called CIE XYZ, devised in 1931 by the Commission Internationale de l'Eclairage. You can understand how the ***CIE color model*** was devised by looking graphically at the results of the color matching experiment. Consider the graph in Figure 2.51. The *x*-axis shows the wavelength, λ, ranging through the colors of the visible spectrum. The *y*-axis shows the relative amounts of red, green, and blue light energy that the "average" observer combines to match the pure light sample. (Units are unimportant. It is the relative values that matter.) Notice that in some cases, red has to be "subtracted" from the composite light (i.e., added to the pure sample) in order to achieve a match.

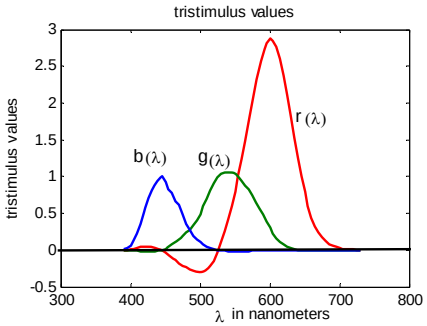


Figure 2.51 Color matching functions

Mathematically, the amount of red light energy needed to create the perceived pure spectral red at wavelength λ is a function of the wavelength, given by r(λ), and similarly for green (the function g(λ)) and blue (the function b(λ)). Let C(λ) be the color the average observer perceives at wavelength λ. Then C(λ) is given by a linear combination of these three components, that is,

$$C(\lambda) = r(\lambda)R + g(\lambda)G + b(\lambda)B$$

Equation 2.12.

Here, R refers to pure spectral red light at a fixed wavelength, and similarly for G and B.

The CIE model is based on the observation that, although there are no three visible primary colors that can be combined in *positive* amounts to create all colors in the visible spectrum, it is possible to use three "virtual" primaries to do so. These primaries – called *X*, *Y*, and *Z* – are purely theoretical rather than physical entities. While they do not correspond to wavelengths of visible light, they provide a mathematical way to describe colors that exist in the visible spectrum. Expressing the color matching functions in terms of *X*, *Y*, and *Z* produces the graphs in Figure 2.52. We can see that *X*, *Y*, and *Z* are chosen so that all three functions remain positive over the wavelengths of the visible spectrum. We now have the equation

$$C(\lambda) = x(\lambda)X + y(\lambda)Y + z(\lambda)Z$$

Equation 2.13

to represent all visible colors.

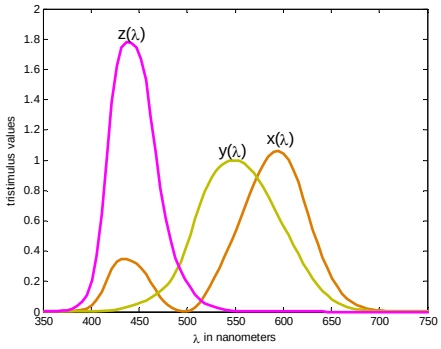


Figure 2.52 XYZ color matching functions

We are still working toward finding a graphical space in which to compare color gamuts, and the CIE color model is taking us there. The dotted line in Figure 2.54 graphs the values of *x*, y, and *z* for all perceived colors $C(\lambda)$ as λ varies across the visible spectrum. By the choice of *X, Y*, and *Z*, all values *x*, *y*, and *z* lie in the positive octant. Clearly, not all visible colors are contained within the RGB gamut.

Aside: As is the case with any set of primaries, there is not just one "right" set of values for X, Y, and Z, but X, Y, and Z are in fact fixed by the CIE standard. Of course they must be primaries with respect to each other, and we want them to be combinable in positive amounts to produce all colors in the visible spectrum. It was also found that by a proper choice of Y, function y(λ) could be modeled such that its shape is the same as *the luminous efficiency function*. The luminous efficiency function is the eye's measured response to monochromatic light of fixed energy at different wavelengths. With Y chosen in this fashion, y(λ) is always equal to the overall luminance of the light in C(λ).

To see the different colors and where the gamuts do or do not overlap, it would be easier to picture this in two dimensions. To simplify things further, it is convenient to normalize the values of x(λ), y(λ), and z(λ) so that they sum to 1. That is, the three colors combine to unit energy. Furthermore, the normalized values show each component's fractional contribution to the color's overall energy. Thus we define

$$x'(\lambda) = \frac{x(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)},\ y'(\lambda) = \frac{y(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)},\ z'(\lambda) = \frac{z(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)}$$

Equation 2.14

In this way, any two of the color components give us the third one. For example,

$$x'(\lambda) = 1 - y'(\lambda) - z'(\lambda)$$

Equation 2.15

x′(λ), y′(λ), and z′(λ) are called the *chromaticity values*. Figure 2.53 shows where the chromaticity values fall within the CIE three-dimensional space. Let s(λ) be a parametric function defined as follows:

$$s(\lambda) = (x'(\lambda), y'(\lambda), z'(\lambda))$$

Equation 2.16

Because we have stipulated that $x'(\lambda)+y'(\lambda)+z'(\lambda)=1$, this function must lie on the $X+Y+Z=1$ plane. We also know that the values for *X*, *Y*, and *Z* are always positive for this function because it was defined that way. Thus, we need only look at the area in the positive octant where $X+Y+Z=1$, represented by the triangle in the figure. The curve traced on this plane shows the values of s(λ) for the pure spectral colors in the

Supplements on XYZ color and the CIE chromaticity diagram:

worksheet

programming exercise

visible spectrum. These are fully saturated colors at unit energy. The colors in the interior of this curve on th $X + Y + Z = 1$ plane are still at unit energy, but not fully-saturated.

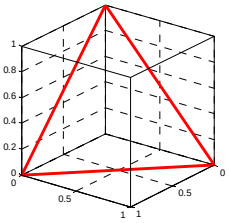


Figure 2.53 X+Y+Z = 1 plane

Not all visible colors are represented in the CIE chromaticity diagram. Color perceptions that depend in part on the luminance of a color are absent from the diagram – brown, for example, which is an orange-red of low luminance.

To picture the curve projected onto the X+Y+Z = 1 plane, imagine standing right in the middle of the X, Y, and Z axes, at the origin, and looking up at the CIE color space, which you imagine casting a shadow (from either side, up on down) onto the X+Y+Z=1 plane. This is precisely s(λ). In Figure 2.54, s(λ) is the finely-dotted line, the X+Y+Z=1 plane is a triangular drawn with solid lines, and the projection of s(λ) onto the X+Y+Z=1 plane is a horseshoe-shaped coarsely-dotted line, which forms the perimeter of the cone seen in Figure 2.55. The cone shown in Figure 2.55 represents the visible colors in CIE space. (Actually, this cone extends even beyond the X+Y+Z=1, as there is no specific maximum energy.) The horseshoe-shaped outline from Figure 2.54 is projected onto the X-Y plane. The 2D projection on the XY plane is called the CIE Chromaticity Diagram (Figure 2.56). In this two-dimensional diagram, we have a space in which to compare the gamuts of varying color models. However, we have said that color is inherently a three-dimensional phenomenon in that it requires three values for its specification. We must have dropped some information in this two-dimensional depiction. The information left out here is energy. Recall that we have normalized the chromaticity functions so that they combine for "unit energy." Unit energy is just some fixed level, the only one we consider as we compare gamuts within the CIE diagram, but it's sufficient for the comparison.

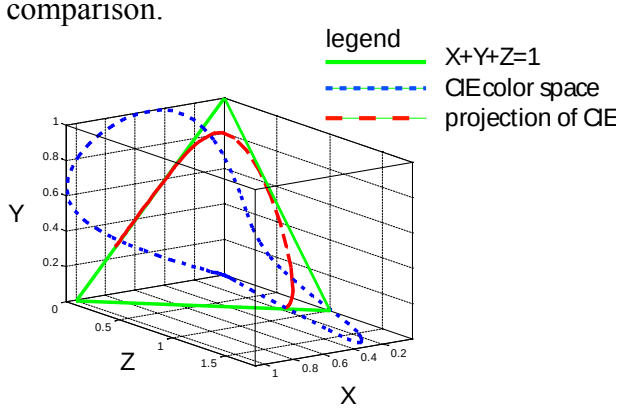


Figure 2.54 Visible color spectrum projected onto the X + Y + Z = 1 plane

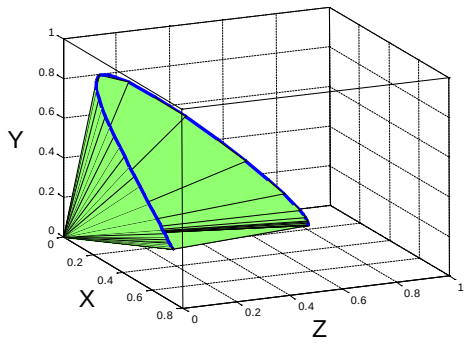


Figure 2.55 Visible colors in CIE color space

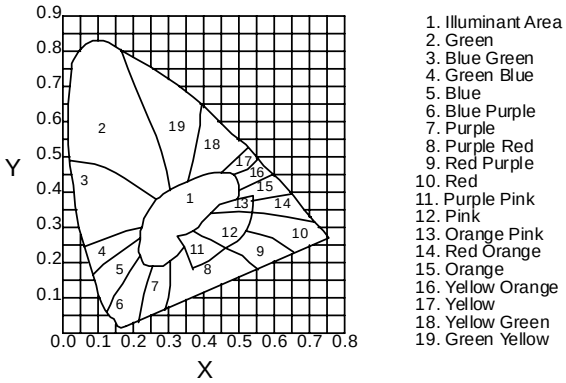


Figure 2.56 CIE chromaticity diagram

Figure 2.57 shows the gamuts for the RGB color vs. the CMYK color space. Note that for each, we must be assuming particular wavelengths for RGB and CMY. Thus, the RGB gamut could be the gamut for a specific computer monitor, given the wavelengths that it uses for its pure red, green, and blue. For any given choice of R, G, and B, these primary colors can be located in the CIE chromaticity diagram by the *x* and *y* values given below. For example, a reasonable choice or R, G, and B would be located at these positions in the CIE diagram:

	R	*G*	*B*
x	0.64	0.30	0.15
y	0.33	0.60	0.06

According to these values, the color red lies at 0.64 along the horizontal axis in the CIE diagram and 0.33 along the vertical axis.

The gamut for RGB color is larger than the CMYK gamut. However, neither color space is entirely contained within the other, which means that there are colors that you can display on the computer monitor that cannot be printed, and vice versa. In practice this is not a big problem. The range of colors for each color model is great and the variations from one color to the next are sufficiently detailed, and usually media creators do not require an exact reproduction of their chosen colors from one display device to the next. However, where exact fidelity is aesthetically or commercially desirable, users need to be aware of the limits of color gamuts.

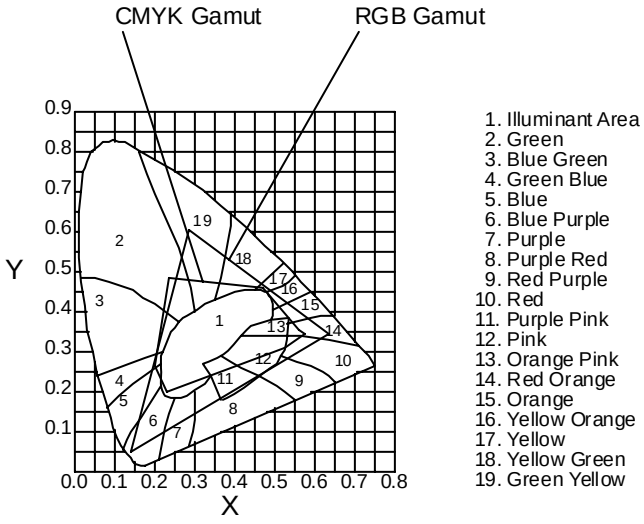


Figure 2.57 RGB vs. CMYK gamuts

CIE diagram is helpful in illustrating color concepts and relationships. In Figure 2.58, point *C* is called ***illuminant C***, or fully saturated white. Points along the edge of the curved horseshoe shape are fully saturated spectral colors. Colors along the straight line base are purples and magentas, colors that cannot be produced by single-wavelength spectral light nor isolated from daylight spectrographically. The line joining any two colors in the graph corresponds to the colors that can be created by a combination of these colors. To find the dominant wavelength of a color that lies at point *A*, it is possible to create a line between points *C* and *A* and extend it to the nearest point on the perimeter of the horseshoe shape. For example, the dominant wavelength of point *A* in Figure 2.58 is approximately 550, and the color is a shade of green. If a point is closest to the base of the horseshoe, it is called a ***nonspectral color***. *D* in Figure 2.58 is an example. Points like *D* have no spectral dominant wavelength. However, if we draw a line from *D* through *C* and find where it crosses the horseshoe shape (at point *E*), we find the complementary dominant wavelength of a nonspectral point. The ratio of a point's distance from the perimeter as opposed to its distance from illuminant *C* indicates how saturated the color is. For example, point *A* is closer to illuminant *C* than to point *B*. Thus, it is a pastel or light green.

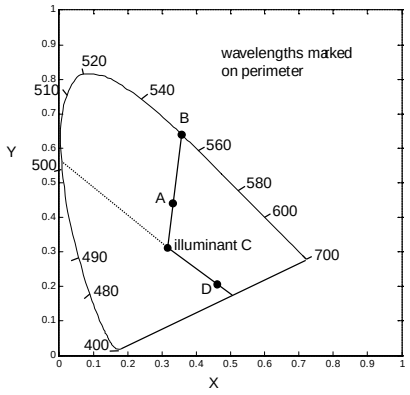


Figure 2.58 Illuminant C

A third advantage of the CIE Chromaticity Diagram is that it gives us a way to standardize color representation. Say that you have a color and you want to tell someone else exactly what that color is. It is possible, with the right instruments, to measure the x, *y*, and *z* values of your color and communicate these to someone else as a precise specification. In the reverse, other instruments allow you to re-create a color specified in *x*, *y*, and *z* coordinates.

The conversion from CIE-XYZ to RGB is a simple linear transformation. The coefficients in the conversion matrix are dependent on the wavelengths chosen for the primaries for the RGB model and well as the definition of white. As noted previously, the RGB values can vary from one computer monitor to another. A reasonable transformation matrix is given below.

key equation

For a pixel represented in XYZ color, let the values for the three color components be *X*, *Y*, and *Z*. Then the equivalent *R*, *G*, and *B* color components in the RGB color model are given by

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.24 & -1.54 & -0.50 \\ -0.97 & 1.88 & 0.04 \\ 0.06 & -0.20 & 1.06 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Conversion from RGB to CIE-XYZ uses the inverse of the above matrix.

2.6.7 CIE L*U*V*, CIE L*a*b*, and Perceptual Uniformity

The CIE XYZ model has three main advantages: it is device-independent; it provides a way to represent all colors visible to humans; and the representation is based upon spectrophotometric measurements of color. The RGB and CMYK color models, on the other hand, are not device independent. They begin with an arbitrary choice of complementary primary colors, from which all other colors in their gamuts are derived. Different computer monitors or printers can use different values for R, G, and B, and thus their gamuts are not necessarily identical. The RGB and CMYK models are not comprehensive, either. No matter the choice of primary colors in either model, there will exist colors visible to humans that cannot be represented.

The development of the CIE XYZ color model overcame these disadvantages, but there was still room for improvement. A remaining disadvantage of the CIE XYZ model is that it is not perceptually uniform. In a ***perceptually uniform color space***, the distance between two points is directly proportional to the perceived difference between the two colors. A color space that is perceptually uniform is easier to work with at an intuitive level, since colors (as they look to the human eye) change at a rate that is proportional to changes in the values representing the colors.

It is possible for a color model to be perceptually uniform in one dimension but not perceptually uniform in its three dimensions taken together. For example, in the HSV color model, hues change at a steady rate as you rotate from 0 to 360 degrees around the plane denoting hue. Similarly, brightness varies steadily up and down the brightness

axis. However, equidistant movement through the three combined planes do not result in equal perceived color changes.

The Commission Internationale de l'Eclairage continued to refine its color model and by 1976 produced the CIE L*U*V* and CIE L*a*b* models. CIE L*a*b* is a subtractive color model in which the L* axis gives brightness values varying from 0 to 100, the *a* axis moves from red (positive values) to green (negative values), and the *b* axis moves from yellow (positive values) to blue (negative values). CIE L*U*V is an additive color model similarly constructed to achieve perceptual uniformity, but found to be less convenient in practical usage.

Because the CIE L*a*b* model can represent all visible colors, it is generally used as the intermediate color model in conversions between other color models, as described in Section 2.4.

2.6.8 Color Management Systems

As careful as you might be in choosing colors for a digital image, it is difficult to ensure that the colors you choose will be exactly the colors that others see when your picture is placed on the web or printed in hard copy. There are a number of reasons for this. As we have seen, the gamut of colors representable by an RGB monitor is not identical to the gamut printable in a CMYK color processing system, so you may lose color fidelity in going from the electronic to the hard copy format. Even if you remain in RGB color and display your picture only on computers, you can't be sure that the RGB space of one monitor is the same as the RGB space of another. The color space of your own monitor can even change over time as your computer ages as the screen's ability to glow and show color diminishes. There are perceptual and environmental influences as well. Colors look different depending on the ambient lighting, the incidental surrounding colors like background patterns on your computer screen, and the reflection of your own clothing on the computer monitor.

Most of the time, the shifts in a picture's colors from one environment to the next are subtle and insignificant. However, artists may have aesthetic reasons for seeking color consistency, or an advertising agency may have commercial ones. In cases where color consistency is important, a color management system is needed. Different hardware devices and application programs – e.g., scanners, computer monitors, printers, and image processing programs – each have their own color spaces and basic color settings. A ***color management system*** communicates the assumptions about color spaces, settings for primary colors, and the mapping from color values to physical representations in pixels and ink from one device to another. Using the CIE color space as a universal, device-independent language, a color management system serves as a translator that communicates color settings from one device or software program to another.

Color management involves five steps: calibrating your monitor; characterizing your monitor's color profile; creating an individual image's color profile that includes choices for color model and rendering intent; saving the color profile with the image; and reproducing the image's color on another device or application program on the basis of the source and destination profiles.

Monitor calibration and characterization can be done with specialized hardware or software. The operating system of your computer or your image processing program

probably both have a color management system that allows you to calibrate and profile your monitor, or it is possible to use more precise hardware-based systems. Calibration should be done before the monitor is characterized. For good results, it should be done in "typical" lighting conditions, on a monitor that has been "warmed up," and with a background screen of neutral gray. Generally this step begins with the choice of a basic ICC (International Color Consortium) profile, to which changes can be made. The first adjustment is setting the monitor's ***white point*** as measured in degrees Kelvin. Daylight is 6500° Kelvin – generally the default hardware setting – while warm white is 5000°. While usually you'll want to keep the white point set to the hardware default, in some cases you may want to adjust it to reflect the conditions in which you expect your images to be viewed. Calibration also involves gamma correction, or adjustment of the midtone brightness. When the adjustments have been made, the new monitor profile is saved. If precise color reproduction is important to your work, it may be necessary to recalibrate your monitor periodically as image specifications or working environments change.

Once your monitor has been properly calibrated, a ***color management policy*** can be created and saved with each individual image. The image's color management policy is based upon the monitor profile, to which special settings are added for a particular image or group of images that determine how color is translated from one space to another. Typically, your color management system will provide a set of predefined standard color management policies that are sufficient for most images. For example, sRGB is a color management policy devised to be suitable for the typical PC monitor and accepted as a standard in many monitors, scanners, and printers. In addition to these standard policies, it is also possible to create a profile that customizes the settings for ***rendering intent***, dot gain, and the color management engine to be used. The color management engine determines how pixel values in the image file convert to the voltage values applied to – and thus the colors of – the pixels on the computer screen. The rendering intent determines how a color in one color space will be adjusted when it is out of the gamut of the color space to which it is being converted, which could happen, for example, when moving from RGB to CMYK color space. One rendering intent might seek to preserve colors in the manner that they are perceived by the human eye relative to other colors; another might sacrifice color similarities for the saturation, or vividness, of the overall image. ***Dot gain*** is a matter of the way in which wet ink spreads as it is applied to paper, and how this may affect the appearance of an image. The color management policy created for an image can be saved and shared with other images. It is also embedded in individual image file and is used to communicate color, expressed in terms of the device-independent CIE color space, from one device and application program to another.

Most of us who work with digital image processing don't need to worry very much about color management systems. Your computer monitor has default settings and is initially calibrated such that you may not find it necessary to recalibrate it, and the default color management policy in your image processing program may be sufficient for your purposes. However, it is good to know that these tools exist so that you know where to turn if precise color management becomes important.

2.7 Vector Graphics

2.7.1 Geometric Objects in Vector Graphics

Section 2.2 discusses how data is represented in bitmaps, which are suitable for photographic images with continuously varying colors. We now turn to ***vector graphics***, an image file format suitable for pictures with areas of solid, clearly-separated colors – cartoon images, logos, and the like. As opposed to being "painted" pixel by pixel, a vector graphic image is "drawn" object by object in terms of each object's geometric shape.

Supplements on the mathematics of curve-drawing:

interactive tutorial

worksheet

Although there are many file formats for vector graphics – *.fh* (Freehand), *.ai* (Adobe Illustrator), *.wmf* (Windows metafile), *.eps* (encapsulated Postcript), etc. – they are all similar in that they contain the parameters to mathematical formulas defining how shapes are drawn. Lines can be specified by their endpoints, squares by the length of a side, rectangles by the length of two sides, circles by their radius, and so forth. However, not everything you might want to draw can be pieced together by circles, rectangles, and lines. What if you want to draw a curved flower vase, a winding path disappearing into the woods, a tree branch, or a plate of spaghetti? Curves are the basis of many of the interesting and complex shapes you might want to draw.

Consider how a curve is rendered on a computer display. If you were working in a drawing program that offered no curve-drawing tool, what would you do? You'd have to create a curve as a sequence of short line segments connected at their endpoints. If the line segments were short enough, the curve would look fairly smooth. But this would be a very tedious way to draw a curve, and the curve would be difficult to modify. A better way would be to select just a few points and ask a curve-drawing tool to smooth out the curve defined by these points. This is how curve tools work in drawing programs. In the next section, we look behind the scenes at the mathematical basis for curve-drawing tools (sometimes called *pen tools*). These tools make your work easy. You simply choose a few points, and a curve is created for you. But how are these curve-drawing tools themselves implemented?

In the sections that follow, we will present the mathematics underlying the implementation of curves in vector graphics applications. A side note about terminology: If you've encountered the term *spline* in your work in digital media, you may wonder what the difference is between a curve and a spline. The word *spline* was borrowed from the vocabulary of draftspersons. Before computers were used for CAD design of airplanes, automobiles, and the like, drafters used a flexible metal strip called a *spline* to trace out smooth curves along designated points on the drafting table. Metal splines provided smooth, continuous curves that now can be modeled mathematically on computers. Some sources make a distinction between curves and splines, reserving the latter term for curves that are interpolated (rather than approximated) through points and that have a high degree of continuity (as in natural cubic splines and B-splines). Other sources use the terms *curve* and *spline* interchangeably. We will distinguish between Hermite and Bézier curves on the one hand and natural cubic splines on the other following the terminology used in most sources on vector graphics. We restrict our discussion to Hermite and Bézier curves.

2.7.2 Specifying Curves with Polynomials and Parametric Equations

Specifying a curve as a sequence of piecewise linear segments is tedious and inefficient. It's much handier simply to choose a few points and ask the drawing program to fit the curve to these. But for this to happen, the drawing program needs to use something more powerful than linear equations to define the curve. In the discussion that follows, we show that ***parametric cubic polynomial functions*** are an excellent formulation for curves. We will step you through the basic mathematics you need in order to understand curve creation, including a review of polynomials, functions, and parametric equations. With this background, you'll be able to understand how the pen tool works in programs such as Illustrator or GIMP.

An ***n^{th} degree polynomial*** is a function that takes the form

$$a_n t^n + a_{n-1} t^{n-1} + a_{n-2} t^{n-2} + ... + a_1 t + a_0$$

Equation 2.17

where $a_n \neq 0$ and $a_0, a_1, ..., a_n$ are the coefficients of the polynomial. Cubic polynomials (i.e., 3rd degree polynomials, where the highest power is 3), are a good way to represent curves. They offer sufficient detail for approximating the shape of curves while still being efficient and easy to manipulate.

The polynomial functions used to describe curves can be expressed in one of three ways: as explicit functions, as implicit functions, or as parametric representations. Both explicit functions and implicit functions are non-parametric forms. You will see that parametric representations are most convenient for curves. Before looking at a variety of parametric cubic polynomials that are used to represent curves, let's review the difference between parametric and non-parametric forms.

The ***explicit form of a function*** with one independent variable is

$$y = f(x)$$

Equation 2.18

For example, the explicit form of the equation for a half-circle is

$$y = \sqrt{r^2 - x^2}$$

Equation 2.19

where *r* is the radius of the circle – a constant for any given circle. A function such as this represents a one-dimensional function in that it has only one independent variable – in this case, *x*. Note that a one-dimensional function can be graphed as a shape in two-dimensional space. In the discussion that follows, we will use one-dimensional functions. The observations are easily generalized to two-dimensional functions graphable in 3D space.

The ***implicit form of the function*** equivalent to Equation 2.18 is

$$f(x, y) = 0$$

Equation 2.20

For example, the implicit form of the equation for a circle is

$$x^2 + y^2 - r^2 = 0$$

Equation 2.21

Both the explicit and the implicit forms given above are ***non-parametric representations***, and neither one is particularly convenient for representing curves. In the case of the explicit form, each value for *x* gives only one value for *y*, and this makes it impossible to

represent ellipses and circles. Furthermore, representing curves with vertical tangents is difficult because the slope of such a tangent would be infinity. In the case of the implicit functional form, it is difficult to specify just part of a circle. To overcome these and other problems, the parametric functional representation is used to define curves.

The ***parametric representation of a function*** takes the general form

$$\boldsymbol{P}(t) = (x(t), y(t))$$

Equation 2.22

for t varying within some given range. Generally, the range of t is taken to be between 0 and 1, but other ranges are possible. (The range can always be normalized to between 0 and 1 if needed.) $\boldsymbol{P}$ effectively gives us the points on the Cartesian plane which constitute the curve, with the points' positions given by an equation for each of the variables in the function, as in

$$x = x(t) \text{ and } y = y(t)$$

Equation 2.23

As mentioned above, cubic polynomials are used in the parametric representation for curves. This means that for a curve, the x and y positions of the points in the curve are given by two parametric equations as follows:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

Equation 2.24

and

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

Equation 2.25

It is sometimes convenient to represent the parametric equations in matrix form. This gives us:

$$\boldsymbol{P}(t) = \begin{bmatrix} x(t) & y(t) \end{bmatrix} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix} \quad 0 \le t \le 1$$

Equation 2.26

or, in short,

$$\boldsymbol{P} = \boldsymbol{T} * \boldsymbol{C}$$

Equation 2.27

$$\text{where } \boldsymbol{T} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \text{ and } \boldsymbol{C} = \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix} = \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix}$$

Equation 2.28

So let's think about where this is leading us. We define ***control points*** to be points that are chosen to define a curve. We want to be able to define a curve by selecting a number of control points on the computer display. In general, $n+1$ control points make it possible to model a curve with an n-degree polynomial. Cubic polynomials have been shown to be good for modeling curves in vector graphics, so our n will be 3, yielding a 3rd degree polynomial like the one in Equation 2.24. What we

need now is an algorithm that can translate the control points into coefficients a_x, b_x, c_x, d_x, a_y, b_y, c_y, and d_y, which are then encoded in the vector graphics file. That is, we want to get matrix $\boldsymbol{C}$. When the curve is drawn, the coefficients are used in the parametric equation, and *t* is varied along small discrete intervals to construct the curve.

Given a set of points, what's the *right* or *best* way to connect them into a curve? As you may have guessed, there's no single right way at all. A number of different methods have been devised for translating points into curves, each with its own advantages and disadvantages depending on the application or work environment. Curve-generating algorithms can be divided into two main categories: interpolation algorithms and approximation algorithms. An ***interpolation algorithm*** takes a set of control points and creates a curve that runs directly through all of the points. An ***approximation algorithm*** creates a curve that does not necessarily pass through all the control points. Approximation algorithms are sometimes preferable in that they allow the user to move a single control point and alter just one part of the curve without affecting the rest of it. This is called ***local control***. Hermite curves and natural cubic splines are based on interpolation algorithms. Bézier curves are based on an approximation algorithm. We'll focus on Bézier curves here since they are the basis for curve-drawing tools in commonly-used vector graphics environments.

The general strategy for deriving the coefficients for the parametric equations from the control points is this: To derive an n^{th} degree parametric equation, you must find $n+1$ coefficients, so you need $n+1$ equations in $n+1$ unknowns. In the case of a cubic polynomial, you need four equations in four unknowns. These equations are formulated from constraints on the control points. For Bézier curves, for example, two of the control points are constrained to form tangents to the curve. Let's see how this can be derived.

2.7.3 Bézier Curves

Bézier curves are curves that can be approximated with any number of control points. However, it simplifies the discussion to think of the curve in segments defined by four control points. We divide these control points into two types: two endpoints, $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$; and two interior control points, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. Bézier curves are uniform cubic curves in the sense that it is assumed that the control points are evenly distributed along the range of parameter *t*. Thus the control points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ correspond to the values of *t* = 0,1/3, 2/3, and 1. The curve is defined by the constraints that $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$ are endpoints, the line from $\boldsymbol{p}_0$ to $\boldsymbol{p}_1$ is a tangent to one part of the curve, and the line from $\boldsymbol{p}_3$ to $\boldsymbol{p}_2$ is a tangent to another part of the curve. Thus, $\boldsymbol{P}'(0)$ is slope of the line segment from $\boldsymbol{p}_0$ to $\boldsymbol{p}_1$, yielding

$$3*\boldsymbol{a}*0^2 + 2*\boldsymbol{b}*0 + \boldsymbol{c} = (\boldsymbol{p}_1 - \boldsymbol{p}_0)/(1/3 - 0)$$
$$\therefore \boldsymbol{c} = 3(\boldsymbol{p}_1 - \boldsymbol{p}_0)$$

Equation 2.29

Similarly, $\boldsymbol{P}'(1)$ is the slope of the line segment from $\boldsymbol{p}_2$ to $\boldsymbol{p}_3$, yielding

$$3*\boldsymbol{a}*1^2 + 2*\boldsymbol{b}*1 + \boldsymbol{c} = (\boldsymbol{p}_3 - \boldsymbol{p}_2)/(1 - 2/3)$$
$$\therefore 3\boldsymbol{a} + 2\boldsymbol{b} + \boldsymbol{c} = 3(\boldsymbol{p}_3 - \boldsymbol{p}_2)$$

Equation 2.30

The constraints that $\boldsymbol{p_0}$ is the first point on the curve and $\boldsymbol{p_3}$ is the last are stated as

$$\boldsymbol{a} * 0^3 + \boldsymbol{b} * 0^2 + \boldsymbol{c} * 0 + \boldsymbol{d} = \boldsymbol{p_0}$$

$$\therefore \boldsymbol{d} = \boldsymbol{p_0}$$

Equation 2.31

$$\boldsymbol{a} * 1^3 + \boldsymbol{b} * 1^2 + \boldsymbol{c} * 1 + \boldsymbol{d} = \boldsymbol{p}_3$$

$$\therefore \boldsymbol{a} + \boldsymbol{b} + \boldsymbol{c} + \boldsymbol{d} = \boldsymbol{p}_3$$

Equation 2.32

These four constraint equations in matrix form are

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} * \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_3 \\ 3*(\boldsymbol{p}_1 - \boldsymbol{p}_0) \\ 3*(\boldsymbol{p}_3 - \boldsymbol{p}_2) \end{bmatrix}$$

Equation 2.33

Solving for $\boldsymbol{C}$, the coefficient vector, we get the form $\boldsymbol{C} = \boldsymbol{A^{-1}} * \boldsymbol{G}_B$

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} * \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_3 \\ 3*(\boldsymbol{p}_1 - \boldsymbol{p}_0) \\ 3*(\boldsymbol{p}_3 - \boldsymbol{p}_2) \end{bmatrix}$$

Equation 2.34

Thus

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_3 \\ 3*(\boldsymbol{p}_1 - \boldsymbol{p}_0) \\ 3*(\boldsymbol{p}_3 - \boldsymbol{p}_2) \end{bmatrix}$$

Equation 2.35

Simplifying, we get the following key equation:

Let $\boldsymbol{P}(t) = (x(t), y(t))$ denote the parametric representation of a curve where (x, y) are the pixel positions across the curve, $x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$ and $y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$. Let the coefficients of $x(t)$ and $y(t)$ be given by $C = \begin{bmatrix} \boldsymbol{a} \\ \boldsymbol{b} \\ \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix} = \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix}$. Then a

key equation

Bézier curve is defined by control points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ and the equation

$$\boldsymbol{C} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \end{bmatrix}$$

Equation 2.36

To create an even simpler form of Equation 2.36, let

$$\boldsymbol{M} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

and

$$\boldsymbol{G} = \begin{bmatrix} \boldsymbol{p}_0 \\ \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \end{bmatrix}$$

This yields

$$\boldsymbol{C} = \boldsymbol{M} * \boldsymbol{G}$$

Equation 2.37

$\boldsymbol{M}$ is called the *basis matrix*. $\boldsymbol{G}$ is called the *geometry matrix*. The basis matrix and geometry matrix together characterize a type of curve drawn by means of particular control points based on constraints on these points – in this case, in the manner used to define a Bézier curve.

With $\boldsymbol{M}$ and $\boldsymbol{G}$, we can derive another convenient representation for Bezier curves, in terms of the curve's *blending functions*, as they are called. Recall that $\boldsymbol{T} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$. A Bézier curve is defined by a cubic polynomial equation.

$$\boldsymbol{P}(t) = \boldsymbol{T} * \boldsymbol{M} * \boldsymbol{G}$$

Equation 2.38

The blending functions are given by $\boldsymbol{T} * \boldsymbol{M}$. That is,

$$\boldsymbol{P}(t) = (\boldsymbol{T} * \boldsymbol{M}) * \boldsymbol{G} = (1-t)^3 \boldsymbol{p}_0 + 3t(1-t)^2 \boldsymbol{p}_1 + 3t^2(1-t)\boldsymbol{p}_2 + t^3 \boldsymbol{p}_3$$

Equation 2.39

(You should verify this by doing the multiplication and simplifying $\boldsymbol{T} * \boldsymbol{M}$.) The multipliers $(1-t)^3$, $3t(1-t)^2$, $3t^2(1-t)$, and t^3 are the ***blending functions*** for Bézier curves and can be viewed as "weights" for each of the four control points. They are also referred to as ***Bernstein polynomials***.

A more general formulation of Bézier curves allows for polynomials of any degree n and describes the curves in terms of the blending functions. It is as follows:

$$\boldsymbol{P}(t) = \sum_{k=0}^{n} \boldsymbol{p}_k blending_{k,n}(t) \text{ for } 0 \le t \le 1$$

Equation 2.40

In the equation, $blending_{k,n}(t)$ refers to the blending functions, where n is the degree of the polynomial used to define the Bézier curve, and k refers to the "weight" for the k^{th} term in the polynomial. $blending_{k,n}(t)$ is defined as follows:

$$blending_{k,n}(t) = C(n,k)t^k(1-t)^{n-k}$$

Equation 2.41

and $C(n,k)$ is defined in turn as

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

Equation 2.42

In the case of the cubic polynomials that we have examined above, $n = 3$, and the blending functions are

$$\begin{aligned} blending_{0,3} &= (1-t)^3 \\ blending_{1,3} &= 3t(1-t)^2 \\ blending_{2,3} &= 3t^2(1-t) \\ blending_{3,3} &= t^3 \end{aligned}$$

Equation 2.43

One of the easiest ways to picture a cubic Bézier curve is algorithmically. The ***de Casteljau algorithm*** shows how a Bézier curve is constructed recursively from the four control points. Consider points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ shown in Figure 2.59. Let $\overline{\mathbf{p}_m, \mathbf{p}_n}$ denote a line segment between $\boldsymbol{p}_m$ and $\boldsymbol{p}_n$. First we find the midpoint $\boldsymbol{p}_{01}$ of $\overline{\mathbf{p}_0, \mathbf{p}_1}$, the midpoint $\boldsymbol{p}_{12}$ of $\overline{\mathbf{p}_1, \mathbf{p}_2}$, and the midpoint $\boldsymbol{p}_{23}$ of $\overline{\mathbf{p}_2, \mathbf{p}_3}$. We then find the midpoint $\mathbf{p}_{012}$ of $\overline{\mathbf{p}_{01}, \mathbf{p}_{12}}$ and the midpoint $\boldsymbol{p}_{123}$ of $\overline{\mathbf{p}_{12}, \mathbf{p}_{23}}$. Finally, we draw $\overline{\mathbf{p}_{012}, \mathbf{p}_{123}}$ and find its midpoint $\boldsymbol{p}_{0123}$. This point will be on the Bézier curve. The same procedure is repeated, based on initial points $\boldsymbol{p}_0$, $\boldsymbol{p}_{01}$, $\boldsymbol{p}_{012}$, and $\boldsymbol{p}_{0123}$ on one side and $\boldsymbol{p}_{0123}$, $\boldsymbol{p}_{123}$, $\boldsymbol{p}_{23}$, $\boldsymbol{p}_3$ on the other. This goes on recursively, theoretically down to infinite detail – or, more practically, down to pixel resolution – thereby tracing out the Bézier curve. Analyzing this recursive procedure mathematically is another method for deriving the Bernstein polynomials.

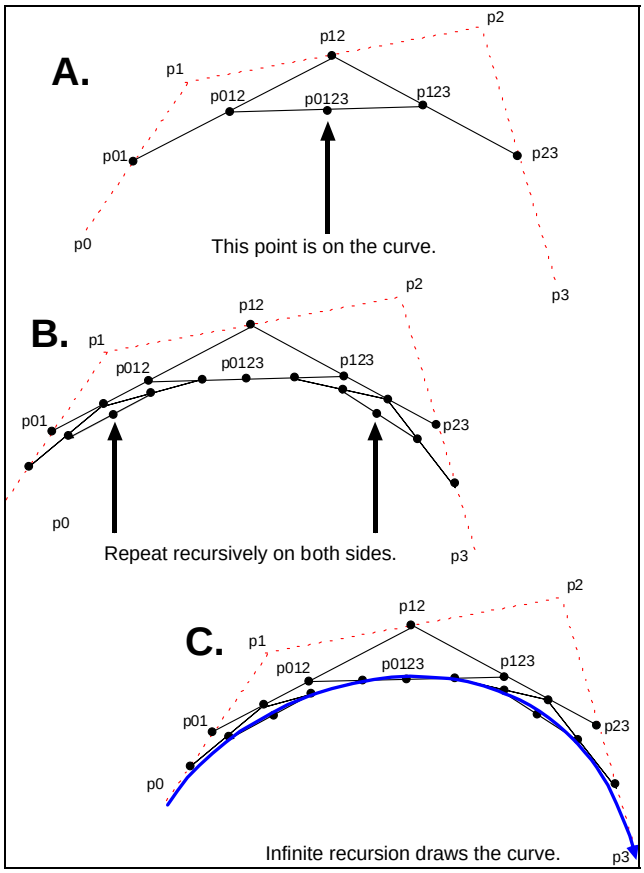


Figure 2.59 De Casteljau algorithm for drawing a Bézier curve

To conclude this section, it may be helpful to return to an intuitive understanding of Bézier curves by considering their properties and looking at some examples as they commonly appear in drawing programs. Bézier curves have the following characteristics:

- Bézier curves can be based on *n*-degree polynomials of degrees higher than three. However, most drawing programs create Bézier curves from cubic polynomials, and thus the curves are defined by four control points. The number of control points is always one more than the degree of the polynomial.
- A Bézier curve with control points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ passes through the two endpoints $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$ and is approximated through the two interior control points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. The lines between $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ and between $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ are tangent to the curve.
- Bézier curves are fairly easy to control and understand in a drawing program because the curve will always lie within the convex hull formed by the four points. The convex hull defined by a set $\boldsymbol{S}$ of points is the polygon formed by a subset of the points such that all the points in $\boldsymbol{S}$ are either inside or on an edge of the polygon.
- The mathematics of moving, scaling, or rotating a Bézier curve is easy. If the operation is applied to the control points, then the curve will be appropriately transformed. (Note that transformations such as moving, scaling, and rotating are called ***affine transformations***. An affine transformation is one that preserves colinearity and ratios of distances among points.)

Figure 2.59 shows the stepwise creation of a Bézier curve as it is done using the pen tool in programs such as Illustrator or GIMP. We'll have to adjust our definition of control points in order to describe how curves are drawn in these application programs. The control points defined in the Bézier equation above are $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$. We need to identify another point that we'll call $\boldsymbol{p}_x$. To make a curve defined by the mathematical control points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$, you click on four physical control points $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_x$ as follows:

- Click at point $\boldsymbol{p}_0$, hold the mouse down and drag to point $\boldsymbol{p}_1$.
- Release the mouse button.
- Move the cursor to your desired endpoint $\boldsymbol{p}_3$, click, and pull to create a "handle" that ends at $\boldsymbol{p}_x$.
- Release the mouse button.

As you pull from $\boldsymbol{p}_3$ to $\boldsymbol{p}_x$, a handle extends equidistant from $\boldsymbol{p}_3$ but in the opposite direction from $\boldsymbol{p}_x$, ending in control point $\boldsymbol{p}_2$. Notice that both $\boldsymbol{p}_0$ and $\boldsymbol{p}_3$ will have "handles" that can be pulled. The handles appear and disappear and you select and deselect endpoints. After you create the initial curve, you can modify the curve's shape by pulling on the handles. The length of the line segments $\overline{\mathbf{p}_0,\mathbf{p}_1}$ and $\overline{\mathbf{p}_2,\mathbf{p}_3}$ define how strongly the curve is pulled toward $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, respectively. Their orientations determine the direction of the curves on each side.

Supplement on pen tool and curve drawing:

hands-on worksheet

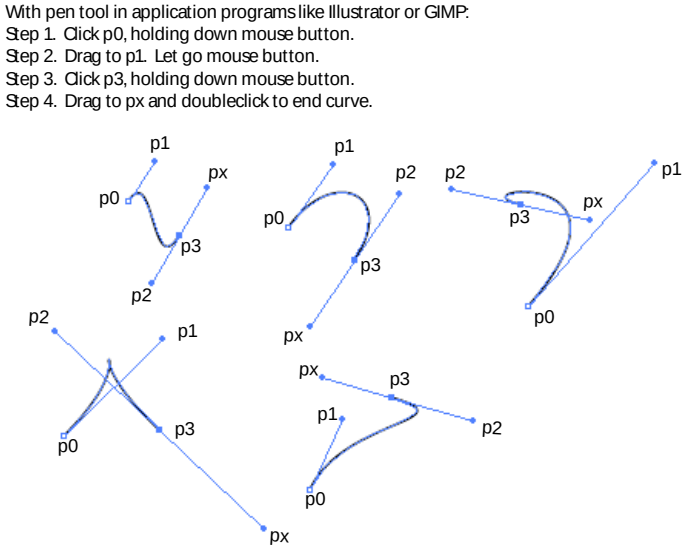


Figure 2.60 Examples of Bézier curves drawn in, for example, Illustrator or GIMP.

2.8 Algorithmic Art and Procedural Modeling

So far, we've talked about two types of digital images: bitmap graphics and vector graphics. Bitmaps are created by means of either digital photography or a paint program. Vector graphic images are created by vector drawing programs. Paint and vector drawing programs allow you to work with powerful tools at a high level of

abstraction. However, if you're a computer programmer, you can also generate bitmaps and graphical images directly by writing programs at a lower level of abstraction. To "hand-generate" and work with bitmaps, you need to be adept at array manipulation and bit operations. To work at a low level of abstraction with vector graphics, you need to understand the mathematics and algorithms for line drawing, shading, and geometric transformations such as rotation, displacement, and scaling. (These are topics covered in graphics courses.) In this book, we emphasize your work with high-level tools for photographic, sound, and video processing; vector drawing; and multimedia programming. However, we try to give you the knowledge to descend into your image and sound files at a low level of abstraction when you want to have more creative control.

There is a third type of digital image which we will call ***algorithmic art***. (It is also referred to as *procedural modeling*.) In algorithmic art, you create a digital image by writing a computer program based on some mathematical computation or unique type of algorithm, but the focus is different from the type of work we've described so far. In vector graphics, (assuming that you're working at a low level of abstraction) you write programs to generate lines, three-dimensional images, colors, and shading. To do this, you first picture and then mathematically model the objects that you are trying to create. In algorithmic art, on the other hand, your focus is the mathematics or algorithm rather than on some preconceived objects, and you create an image by associating pixels with the results of the calculation. The image emerges naturally as a manifestation of the mathematical properties of the calculation or algorithm.

Supplement on algorithmic art:

interactive tutorial

One of the best examples of algorithmic art is fractal generation. A ***fractal*** is a graphical image characterized by a recursively-repeating structure. This means that if you look at the image at the macro-level – the entire structure – you'll see a certain geometric pattern, and then when you zoom in on a smaller scale, you'll see that same pattern again. For fractals, this self-embedded structure can be repeated infinitely.

Fractals exist in natural phenomena. A fern has the structure of a fractal in that the shape of one frond of a fern is repeated in each of the small side-leafs of the frond. Similarly, a cauliflower's shape is repeated in each sub-cluster down to several levels of detail.

Figure 2.61 Natural fractal structures

You can create a fractal with a recursive program that draws the same shape down to some base level. This is an example of algorithmic art because the self-replicating structure of the image results from the recursive nature of the algorithm. The ***Koch snowflake***, named for Swedish mathematician Helge von Koch, is an example of a fractal structure that can be created by means of a recursive program. Here's an explanation of how you can draw one by hand. First, draw an equilateral triangle. Then draw another triangle of the same size, rotate it 180°, and align the two triangles at their center points. You've created a six-pointed star. Now, consider each point of the star as a separate equilateral triangle, and do the same thing for each of these. That is, create another triangle of the same size as the point of the star, rotate it, and align it with the first. You've just created six more stars, one for each point of the original star. For each of these stars, do the same thing that you did with the first star. You can repeat this to whatever level of detail you like (or until you can't add any more detail because of the resolution of your picture). If you fill in all the triangles with a single color, you have a Koch snowflake.

Supplement on Koch snowflake:

programming exercise

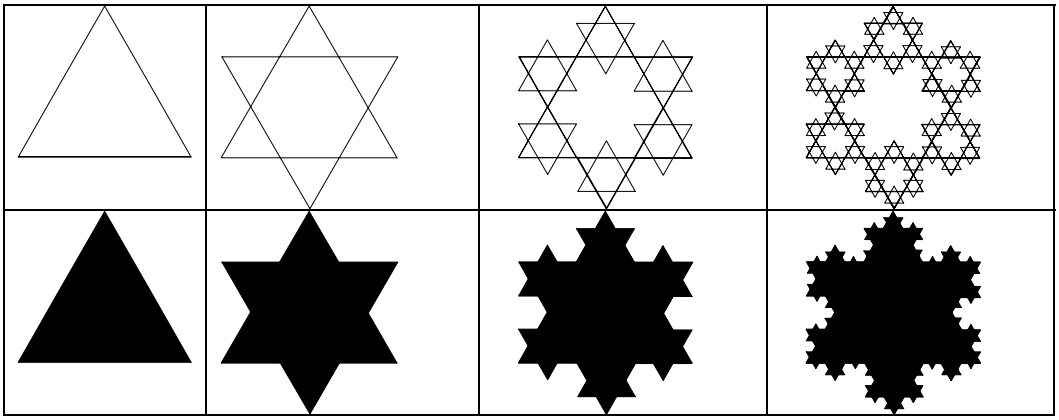


Figure 2.62 Koch's snowflake, a recursively-defined fractal

Another recursively-defined fractal structure is ***Sierpinski's gasket***, which is built in the following manner: Begin with an equilateral triangle with sides that are one unit in length. The triangle is colored black (or whatever solid color you like). Create a white triangle inside of the first one by connecting the midpoints of the sides of the original triangle. Now you have three small black triangles and one white one, each with sides ½ the length of the original. Subdivide each of the black triangles similarly. Keep doing this to the level of detail your resolution allows. This can easily be described by a recursive algorithm.

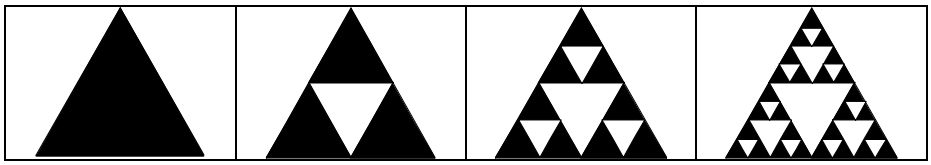


Figure 2.63 Sierpinski's gasket

Another very intriguing type of fractal was discovered by Benoit Mandelbrot in the 1970's. Mandelbrot's method is interesting in that the mathematical computation does not explicitly set out to create a fractal of infinitely-repeating shapes; instead, a beautiful and complex geometry emerges from the nature of the computation itself.

Benoit Mandelbrot was born in Poland in 1924 and divided his academic and professional career in mathematics between France and the United States. Admired for his eclectic, interdisciplinary approach to research, Mandelbrot worked in disciplines spanning statistics, graphics, linguistics, physics, and economics. He coined the word "fractals" to describe both shapes and behaviors that are self-similar at varying levels of detail. His extensive work in fractals defined a new geometry where mathematics meets aesthetics.

The Mandelbrot fractal computation is based on a simple iterative equation.

A Mandelbrot fractal can be computed by the iterative equation

$$f(z) = z^2 + c$$

where z and c are complex numbers. the output of the i^{th} computation is the input z to the $i+1^{st}$ computation.

key equation

Equation 2.44

Complex numbers have a real and an imaginary number component. For c, we will call these components c_r and c_i, and for z the components are z_r and z_i.

$$c = c_r + c_i i \text{ and } z = z_r + z_i i \text{ where } i \text{ is } \sqrt{-1}$$

Equation 2.45

To create a Mandelbrot fractal, you relate values of c on the complex number plane to pixel positions on the image bitmap, and you use these as initial values in computations that determine the color of each pixel. c_r corresponds to the horizontal axis on the plane, and c_i corresponds to the vertical axis. The range of −2.0 to 2.0 in the horizontal direction and −1.5 to 1.5 in the vertical direction works well for the complex number

plane. Let's assume that we're going to create a fractal bitmap that has dimensions of 1024 × 768 pixels. Given these respective dimensions for the two planes, each pixel position (*x,y*) is mapped proportionately to a complex number (c_r, c_i).

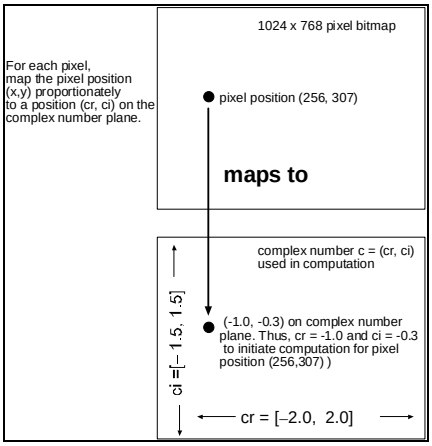


Figure 2.64 Mapping from pixel plane to complex number plane for Mandelbrot fractal

Supplement on Mandebrot and Julia fractals:

programming exercise

interactive tutorial

To compute a pixel's color, you iteratively compute $f(z) = z^2 + c$, where *c* initially is the complex number corresponding to the pixel's position, and *z* is initially 0. After the first computation, the output of the i^{th} computation is the input *z* to the $i+1^{st}$ computation. If, after some given maximum number of iterations, the computation has not revealed itself as being unbounded, then the pixel is painted black. Otherwise, the pixel is painted a color related to how many iterations were performed before it was discovered that the computation was unbounded. The result is a Mandelbrot fractal like the ones shown in Figure 2.65 .

Algorithm 2.5 describes the Mandelbrot fractal calculation. We have not shown in this pseudo-code how complex numbers are handled, nor have we been specific about the termination condition. For details on this, see the related programming assignment.

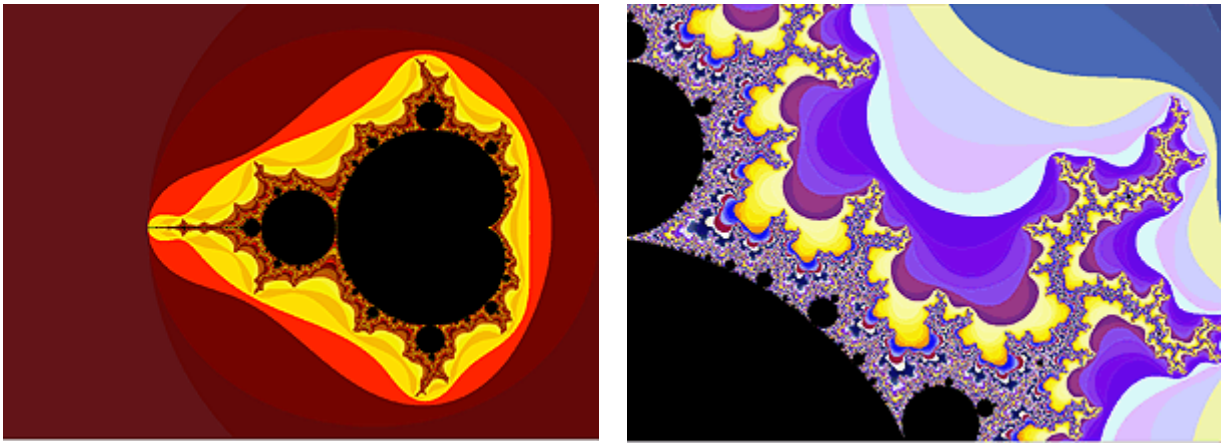


Figure 2.65 Mandelbrot fractals, with two different color maps

```
algorithm mandelbrot_fractal
/*Input:  Horizontal and vertical ranges of the complex number plane.
Resolution of the bitmap for the fractal.
Color map.
Output:  A bitmap for a fractal.*/
{
/*constant MAX is the maximum number of iterations*/
  for each pixel in the bitmap {
    map pixel coordinates (x,y) to complex number plane coordinates (cr, ci)
    num_iterations = 0
    z = 0
    while num_iterations <= MAX and not_unbounded* {
      z =z^2 + c
      num_iterations = num_iterations + 1
    }
/*map_color(x,y) uses the programmer's chosen color map to determine the color of each
pixel based on how many iterations are done before the computation is found to be
unbounded*/
    if num_iterations = MAX then  color(x,y) = BLACK
    else      color(x,y) = map_color(num_iterations)
  }
}
/*We have not explained not_unbounded here.  For an explanation of the termination
condition and computations using complex numbers, see the programming assignment
related to this section.*/
```

Algorithm 2.5

A variation of the Mandelbrot fractal, called the ***Julia fractal***, can be created by relating *z* rather than *c* to the pixel's position and appropriately selecting a value for *c*, which remains constant for all pixels. Values for *c* that create interesting-looking Julia fractals can be determined experimentally, by trial and error. Each different constant *c* creates a fractal of a different shape.

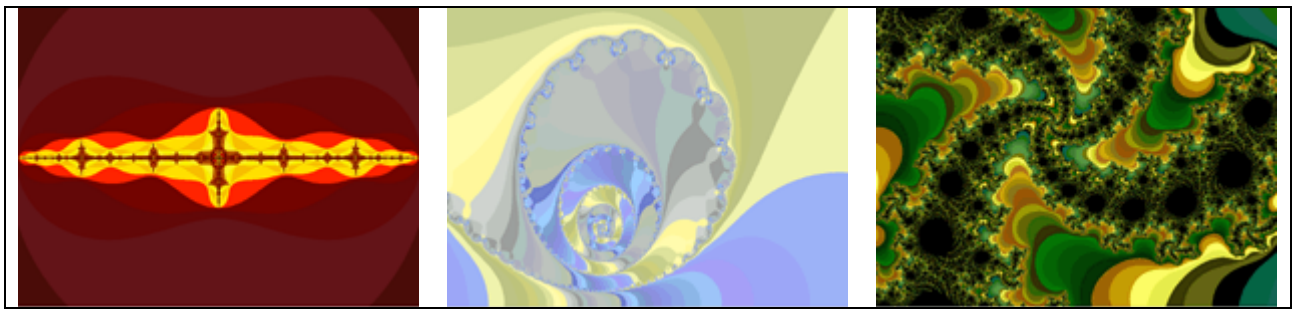


Figure 2.66 Three Julia fractals using different starting values for *c* and different color maps

2.9 Vocabulary

AC component
affine transformations

algorithmic art
aliasing
anti-aliasing
basis functions for DCT (base frequency components)
Bayer color filter array (Bayer filter)
Bernstein polynomial
Bézier curve
bit depth
bitmap and pixmap
bitmap images (raster graphics)
blending function
brightness
charge-coupled device (CCD)
chrominance
CIE color models (CIE-XYZ, LUV, and Lab)
CMY color model (and CMYK)
color aliasing (false coloration, phantom colors, moiré fringes)
color components (color channels)
color gamut
color management policy
color management system
color models
control points
cropping
DC component
de Casteljau algorithm
demosaicing
discrete cosine transform (DCT)
dominant wavelength
dot gain
dots per inch (DPI)
explicit and implicit form of a function
fractal
    Julia fractal
    Mandelbrot fractal

function over the spatial domain
function over the frequency domain
HLS color model
HSV color model
hue
illuminant C
image size
interpolation and approximation algorithms for curve-fitting
Julia fractal
Koch snowflake (Koch star)

local control of curve
megapixels (specification for a digital camera)
moiré effect, moiré pattern
monitor calibration and characterization
nearest neighbor algorithm for demosaicing
non-spectral colors
n^{th} degree polynomial
Nyquist theorem
parametric and non-parametric representations of a function
parametric cubic polynomial function
perceptually uniform color space
pixel
    logical pixel
    physical pixel
pixel dimensions
procedural modeling
quantization
quantization error
resampling
    upsampling
    downsampling
rendering intent
RGB color model
resolution
sampling
sampling error
saturation
Sierpinski's gasket
tangent vector
vector graphics
white point

2.10 Exercises

1. Pixel dimensions, resolution, and image size, hands-on exercise, online
2. DCT, interactive tutorial, worksheet, programming exercise, and mathematical modeling exercise, online.
3. Aliasing in sampling, interactive tutorial and worksheet, online
4. Aliasing in rendering, interactive tutorial and worksheet, online
5. Line drawing, interactive tutorial and programming exercise, online
6. Color model conversions, programming exercise, online
7. XYZ color and the CIE chromaticity diagram, mathematical modeling exercise and programming exercise, online
8. Curve drawing, interactive tutorial, worksheet, and hands-on exercise, online
9. Algorithmic art, interactive tutorial, online
10. Koch snowflakes, programming exercise, online
11. Mandelbrot and Julia fractals, interactive tutorial and programming exercise, online

12. a. What type of values would you expect for the DCT of the enlarged 8 pixel × 8 pixel image below (i.e., where do you expect non-zero values)? The grayscale values are given in the matrix. Explain your answer.

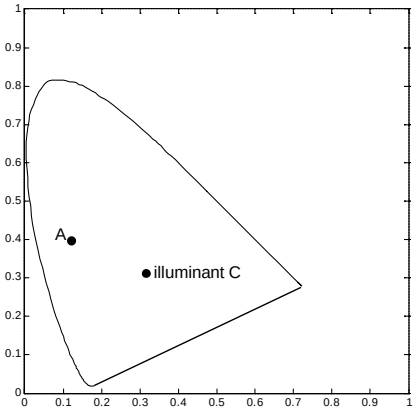
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255
0	255	155	155	255	255	200	255

b. Compute the values. You can use computer help (e.g., write a program, use MATLAB, etc.)

13. Say that your 1-CCD camera detects the following RGB values in a 3 pixel × 3 pixel area. What value would it record for the three pixels that are in boldface, assuming the nearest-neighbor algorithm is used? (Give the R, G, and B values for these pixels.)

G = 240	R = 255	G = 239	R = 244	G = 236
B = 238	**G = 229**	**B = 224**	G = 230	B = 222
G = 244	R = 255	G = 238	**R = 250**	G = 236
B = 230	G = 226	B = 222	G = 232	B = 228
G = 244	R = 255	G = 238	R = 250	G = 236

14. Answer the following questions based on the diagram below.



a. What color is point A?
b. How would you find the dominant wavelength of this color?
c. What does the line segment between A and C represent?
d. How would you determine the saturation of the color at point A?

2.11 Applications

1. Examine the specifications on your digital camera (or the one you would like to have), answering the following questions:
   - Does your camera allow you to choose various pixel dimension settings? What settings are offered?
   - How many megapixels does the camera offer? What does this mean, and how does it relate to pixel dimensions?
   - What type of storage medium does the camera use? How many pictures can you take on your storage medium? This will depend on the size of the storage medium and the size of the pictures. Explain.
   - How can you get your pictures from the camera to your computer?
   - What technology does your camera use for detecting color values?
2. What are the pixel dimensions of your computer display? Can you change them? If so, change the pixel dimensions and describe what do you observe about the images on your display? Explain.
3. Examine the specifications of your pinter (or the one you would like to have). Is it inkjet? Laser? Some other type? What is its resolution? How does this effect the choices you make for initial pixel dimensions and final resolution of digital images that you want to print?
4. For the following exercise, we assume you have access to a digital image processing program (e.g., Photoshop or GIMP), a paint program (e.g. PaintShop), and/or a drawing program (e.g. Corel Draw). Open these programs and examine the color models that they offer you. Which of the color models described in this chapter are offered in the application program? Are there other color models available in the application program that are not discussed in this chapter? Examine the interface – sometimes called a "color chooser." Look at the Help of your application program. How is color be saved by the application program? If you specify color in HLS rather than RGB, for example, using the color chooser, is the internal representation changing, or just the values you use to specify the color in the interface?
5. Do some experiments with the color chooser of your image processing, paint, or draw program in RGB mode to see if you can understand what is meant by the statement "the RGB color space is not perceptually uniform."

Additional exercises or applications may be found at the book or author's websites.

2.12 References

2.12.1 Print Publications

Briggs, John. *Fractals: The Patterns of Chaos.* New York: Simon & Schuster/A Touchstone Book, 1992.

Foley, James D., Steven K. Feiner, John F. Hughes, and Andries Van Dam. *Computer Graphics: Principles and Practice*. 2nd ed. Boston: Addison-Wesley, 1996.

Hargittai, Istvan, and Clifford A. Pickover. *Spiral Symmetry*. Singapore: World Scientific, 1992.

Hearn, Donald, and M. Pauline Baker. *Computer Graphics with OpenGL*. 3rd ed. Upper Saddle River, NJ: Pearson/Prentice-Hall, 2003.

Hill, F. S., Jr. *Computer Graphics Using Open GL*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2001.

Hofstadter, Douglas R. *Gödel, Escher, Bach: An Eternal Golden Braid.* New York: Basic Books, 1979. (Reprinted with a new preface in 1999.)

Livio, Mario. *The Golden Ratio: The Story of Phi, The World's Most Astonishing Number.* New York: Broadway Books, 2002.

Pickover, Clifford. *Chaos and Fractals: A Computer Graphical Journey*. Elsevier, 1998.

Pickover, Clifford. *The Pattern Book: Fractals, Art, and Nature*. Singapore: World Scientific, 1995.

Pokorny, Cornel. *Computer Graphics: An Object-Oriented Approach to the Art and Science*. Franklin Beedle and Associates, 1994.

RAO, K. R., and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications.* San Diego: Academic Press, 1990.

2.12.2 Websites

MPI Programs with Graphics – Mandelbrot Rendering. http://www.cs.appstate.edu/~can/classes/5530/mpiman/node35.html. Accessed 4/25/05.