Chapter 1

Digital Image Basics

1.1 What is a Digital Image?

To understand what a digital image is, we have to first realize that what we see when we look at a "digital image" is actually a physical image reconstructed from a digital image. The digital image itself is really a data structure within the computer, containing a number or code for each *pixel* or picture element in the image. This code determines the color of that pixel. Each pixel can be thought of as a discrete *sample* of a continuous real image.

It is helpful to think about the common ways that a digital image is created. Some of the main ways are via a digital camera, a page or slide scanner, a 3D rendering program, or a paint or drawing package. The simplest process to understand is the one used by the digital camera.

Figure 1.1 diagrams how a digital image is made with a digital camera. The camera is aimed at a scene in the world, and light from the scene is focused onto the camera's picture plane by the lens (Figure 1.1a). The camera's picture plane contains photosensors arranged in a grid-like array, with one sensor for each pixel in the resulting image (Figure 1.1b). Each sensor emits a voltage proportional to the intensity of the light falling on it, and an analog to digital conversion circuit converts the voltage to a binary code or number suitable for storage in a cell of computer memory. This code is called the pixel's value. The typical storage structure is a 2D array of pixel values, arranged so that the layout of pixel values in memory is organized into a regular grid with row and column numbers corresponding with the row and column numbers of the photosensor reading this pixel's value (Figure 1.1c).

Since each photosensor has a finite area, as indicated by the circles in Figure

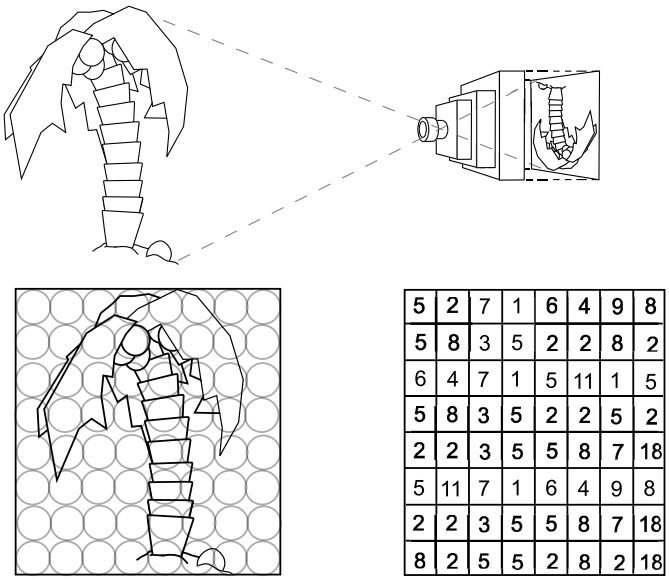

Figure 1.1: Capturing a 2D Continuous Image of a Scene

1.1b, the reading that it makes is a weighted average of the intensity of the light falling over its surface. So, although each pixel is conceptually the sample of a single point on the image plane of the camera, in reality it represents a spread of light over a small area centered at the sample point. The weighting function that is used to describe how the weighted average is obtained over the area of the sample is called a *point spread function*. The exact form of the point spread function is a complex combination of photosensor size and shape, focus of the camera, and photoelectric properties of the photosensor surface. Sampling through a point spread function of a shape that might be encountered in a digital camera is shown in diagram form in Figure 1.2.

A digital image is of little use if it cannot be viewed. To recreate the discretely sampled image from a real continuous scene, there must be as *reconstruction* process to invert the sampling process. This process must convert the discrete image samples back into a continuous image suitable for output on a device like a CRT or LCD for viewing, or a printer or film recorder for hardcopy. This process can also be understood via the notion of the point spread function. Think of each sample (i.e. pixel) in the digital image being passed back through a point spread function that spreads the pixel value out over a small region.

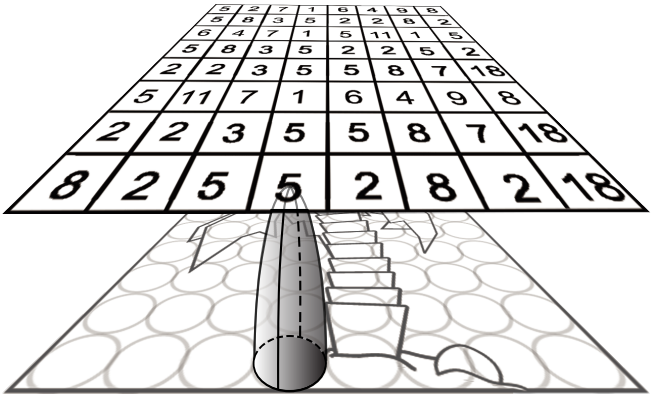

Figure 1.2: Sampling Through a Point-Spread Function

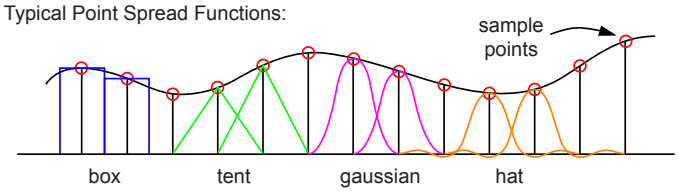

Figure 1.3: Some Typical Point-Spread Functions

1.2 Bitmaps and Pixmaps

1.2.1 Bitmap - the simplest image storage mechanism

A *bitmap* is a simple black and white image, stored as a 2D array of *bits* (ones and zeros). In this representation, each bit represents one *pixel* of the image. Typically, a bit set to zero represents black and a bit set to one represents white. The left side of Figure 1.4shows a simple block letter U laid out on an 8×8 grid. The right side shows the 2-dimensional array of bit values that would correspond to the image, if it were stored as a bitmap. Each row or *scanline* on the image corresponds to a row of the 2D array, and each element of a row corresponds with a pixel on the scanline.

Although our experience with television, the print media, and computers leads us to feel that the natural organization of an image is as a 2D grid of dots or pixels, this notion is simply a product of our experience. In fact, although images are displayed as 2D grids, most image storage media are not organized in this way. For example, the computer's memory is organized into a long linear array

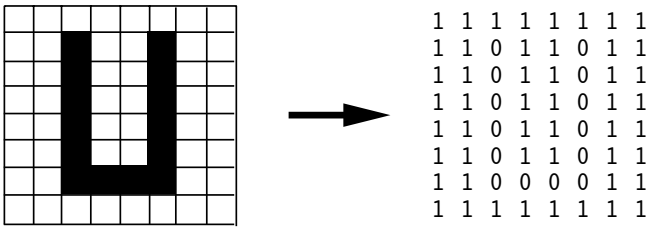

Figure 1.4: Image of Black Block Letter U and Corresponding Bitmap

of addressable *bytes* (8 bit groups) of storage. Thus, somewhere in the memory of a typical computer, the block letter U of Figure 1.4 might be represented as the following string of contiguous bytes:

11111111	11011011	11011011	11011011	11011011	11011011	11000011	11111111

Since the memory is addressable only at the byte level, the color of each pixel (black or white) must be extracted from the byte holding the pixel's value. And, since the memory is addressed as a linear array, rather than as a 2D array, a computation must be made to determine which byte in the representation contains the pixel that we wish to examine, and which bit in that byte corresponds with the pixel.

The procedure `print_bitmap()` in Figure 1.5 will print the contents of the image stored in the array named `bitmap`. We assume that the image represented by `bitmap` contains exactly `width * height` pixels, organized into `height` scanlines, each of length `width`. In other words, the number of pixels vertically along the image is `height`, and the number of pixels horizontally across the image is `width`. The `print_bitmap()` procedure assumes that each scanline in memory is padded out to a multiple of 8 bits (pixels), so that it exactly fits into an integer number of bytes. The variable `w` gives the width of a scanline in bytes.

Another issue is that the representation of groups of pixels in terms of lists of ones and zeros is extremely difficult for humans to deal with cognitively. To convince yourself of this, try looking at a group of two or more bytes of information, remembering what you see, and then writing down the numbers from memory. To make the handling of this binary encoded information more manageable, it is convenient to think of each group of 4 bits as encoding a hexadecimal number. The hexadecimal numbers are the numbers written using a base of 16, as opposed to the usual decimal numbers that use base 10, or the binary numbers of the computer that use base 2. Since 16 is the 4th power of 2, each hexadecimal digit can be represented exactly by a unique pattern of 4 binary digits. These patterns are given in table Table 1.1, and because of their regular organization they can be easily memorized. With the device

```
void print_bitmap(unsigned char *bitmap, int width, int height){

  int w = (width + 7) / 8;  // number of bytes per scanline

  int row;                // scanline number (row)
  int col;                 // pixel number on scanline (column)
  int byte;               // byte number within bitmap array
  int bit;                  // bit number within byte
  int value;             // value of bit (0 or 1)

  for(row = 0; row < height; row++){  // loop for each scanline
    for(col = 0; col < width; col++){   // loop for each pixel on line
      byte = row * w + col / 8;
      bit = 7 - col % 8;
      value = bitmap[byte] >> bit & 1;  // isolate bit
      printf("%1d", value);
    }
    printf("\n");
  }
}
```

Figure 1.5: Procedure to Print the Contents of a Bitmap

of hexadecimal notation, we can now display the internal representation of the block letter U, by representing each 8-bit byte by two hexadecimal digits. This reduces the display to:

FF	DB	DB	DB	DB	DB	C3	FF

1.2.2 Pixmap - Representing Grey Levels or Color

If the pixels of an image can be arbitrary grey tones, rather than simply black or white, we could allocate enough space in memory to store a real number, rather than a single bit, for each pixel. Then arbitrary levels of grey could be represented as a 2D array of real numbers, say between 0 and 1, with pixel color varying smoothly from black at 0.0 through mid-grey at 0.5 to white at 1.0. However, this scheme would be very inefficient, since *floating point* numbers (the computer equivalent of real numbers) typically take 32 or more bits to store. Thus image size would grow 32 times from that needed to store a simple bitmap. The *pixmap* is an efficient alternative to the idea of using a full floating point number for each pixel. The main idea is that we can take advantage of the eye's finite ability to discriminate levels of grey.

Table 1.1: Hexadecimal Notation

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Table 1.2: Combinations of Bits

Bits	# of Combinations	Combinations
1	$2^1 = 2$	0, 1
2	$2^2 = 4$	00, 01, 10, 11
3	$2^3 = 8$	000, 001, 010, 011, 100, 101, 110, 111
...	...	...
8	$2^8 = 256$	00000000, 00000001, ... , 11111110, 11111111

It is a simple mathematical fact that in a group of n bits, the number of distinct combinations of 1's and 0's is 2^n. In other words, n bits of storage will allow us to represent and discriminate among exactly 2^n different values or pieces of information. This relationship is shown in tabular form in Table 1.2. If, in our image representation, we use 1 byte (8 bits) to represent each pixel, then we can represent up to 256 different grey levels. This turns out to be enough to "fool" the eye of most people. If these 256 different grey levels are drawn as vertical lines across a computer screen, people will think that they are seeing a smoothly varying *grey scale*.

The structure of a pixmap, then, is a 2D array of pixel values, with each pixel's value stored as a group of 2 or more bits. To conform to byte boundaries, the number of bits used is typically 8, 16, 24 or 32 bits per pixel, although any size is possible. If we think of the bits within a byte as representing a binary number, we can store grey levels between 0 and 255 in 8 bits. We can easily convert the pixel value in each byte to a grey level between 0.0 and 1.0 by dividing the pixel value by the maximum grey value of 255.

Assuming that we have a pixmap storing grey levels in eight bits per pixel, the procedure `print_greymap()` in Figure 1.6 will print the contents of the image stored in the array named `greymap`. We assume that the image represented by `greymap` contains exactly `width * height` pixels, organized into `height` scanlines, each of length `width`.

```
void print_greymap(unsigned char *greymap, int width, int height){

  int row;              // scanline number (row)
  int col;               // pixel number on scanline (column)
  int value;           // value of pixel (0 to 255)

  for(row = 0; row < height; row++){  // loop for each scanline
    for(col = 0; col < width; col++){   // loop for each pixel on line
      value = greymap[row * width + col];  // fetch pixel value
      printf("%5.3f ", value / 255.0);
    }
    printf("\n");
  }
}
```

Figure 1.6: Procedure to Print the Contents of an 8 bit/pixel Greylevel Pixmap

1.3 The RGB Color Space

If we want to store color images, we need a scheme of color representation that will allow us to represent color in a pattern of bits (just like we represented grey levels as patterns of bits). Fortunately, many such representations exist, and the most common one used for image storage is the *RGB* or Red-Green-Blue system. This takes advantage of the fact that we can "fool" the human eye into "seeing" most of the colors that we can recognize perceptually by superimposing 3 lights colored red, green and blue. The level or intensity of each of the three lights determines the color that we perceive.

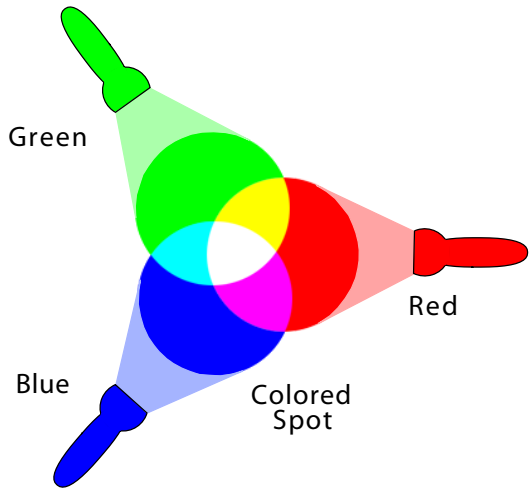

Figure 1.7: Additive Color Mixing for the Red-Green-Blue System

If we think of red, green, and blue levels as varying from 0 (off) to 1 (full brightness), then a color can be represented as a red, green, blue triple. Some example color representations using this on/off scheme are shown in Figure 1.8. It is interesting and somewhat surprising that yellow is made by combining red and green!

$(1,0,0)$=red	$(0,1,0)$=green	$(0,0,1)$=blue
$(1,1,0)$=yellow	$(0,1,1)$=cyan	$(1,0,1)$=magenta
$(0,0,0)$=black	$(1,1,1)$=white	$(0.5,0.5,0.5)$=grey

Figure 1.8: Example Colors Encoded as RGB Triples

Now, we can extend this idea by allowing a group of bits to represent one pixel. We can assign some of these bits to the red level, some to green, and some to blue, using a binary encoding scheme like we used to store grey level. For

example, if we have only 8 bits per pixel, we might use three for the red level, 3 for green, and 2 for blue (since our eye discriminates blue much more weakly than red or green). Figure 1.9 shows how a muted green color could be stored using this kind of scheme. The value actually stored is hexadecimal 59, which is then shown in binary broken into red, green and blue binary fields. Each of these binary numbers is divided by the maximum unsigned number possible in the designated number of bits, and finally shown represented as a (RGB) triple of color primary values, each on a scale of $0 - 1$.

On a high end graphics computer, it is not unusual to allocate 24 bits per pixel for color representation, allowing 8 bits for each of the red, green and blue components. This is more than enough to allow for perceptually smooth color gradations, and fits nicely into a computer whose memory is organized into 8-bit bytes. If you read specifications for computer displays or use graphics software, you will have noticed that many of these systems use red, green, and blue levels between 0-255. These are obviously systems that use an 8-bit per color primary representation.

$$59_{16} = \underset{\text{R}}{\boxed{010}}\,\underset{\text{G}}{\boxed{110}}\,\underset{\text{B}}{\boxed{01}} = (2/7, 6/7, 1/3) = (0.286, 0.757, 0.333)$$

Figure 1.9: 8-Bit Encoding of a Muted Green

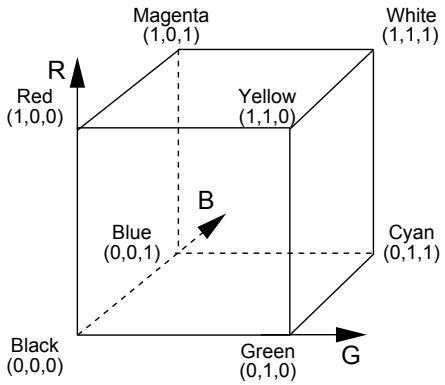

Figure 1.10: RGB Color Cube

Since the RGB system organizes color into three *primaries*, and allows us to scale each primary independently, we can think of all of the colors that are represented by the system as being organized in the shape of a cube, as shown in Figure 1.10. We call this the RGB color cube, or the RGB color space (when we add coordinate axes to measure R, G and B levels). Note that the corners of the RGB color cube represent pure black and pure white, the three primaries red, green and blue, and the 3 secondary colors yellow, cyan and magenta. The

diagonal from the black corner to the white corner represents all of the grey levels. Other locations within the cube correspond with all of the other colors that can be displayed.

A pixmap storing RGB levels using eight bits per primary, with an additional eight bits per pixel reserved, is called an RGBA (or Red, Green, Blue, Alpha) pixmap. The procedure `print_pixmap()` in Figure 1.11 will print the contents of the RGBA image stored in the array named `pixmap`. We assume that the image represented by `pixmap` contains exactly `width * height` pixels, organized into `height` scanlines, each of length `width`.

```
void print_pixmap(unsigned int *pixmap, int width, int height){

  int row;                       // scanline number (row)
  int col;                        // pixel number on scanline (column)
  unsigned int value;    // pixel as fetched from pixmap
  int r, g, b;                   // RGB values of pixel (0 to 255)

  for(row = 0; row < height; row++){  // loop for each scanline
    for(col = 0; col < width; col++){   // loop for each pixel on line
      value = pixmap[row * width + col];  // fetch pixel value
      r = value >> 24;
      g = (value >> 16) & 0xFF;
      b = (value >> 8) & 0xFF;
      printf("(%5.3f,%5.3f,%5.3f) ",
             r / 255.0, g / 255.0, b / 255.0);
    }
    printf("\n");
  }
}
```

Figure 1.11: Procedure to Print the RGB Values in a 32 Bit/Pixel RGBA Pixmap

Note that the code in Figure 1.11 is written to work correctly on a "Big Endian" architecture processor. If you are working on a Windows machine, or any Intel-like computing architecture, your machine will be "Little Endian". This will affect how colors are displayed. Please refer to the *Homework Nuts and Bolts'* section of the next chapter of these notes (on Simple Image File Formats) for a fuller explanation of this issue.

1.4 Other Ways to Organize RGBA Pixmaps

The method of storing red, green, blue and alpha values by packing them into a single `unsigned int` (which is a 32 bit structure) as shown in the example Figure 1.11, is only one way to organize the storage of RGBA pixels. Another way would be to define a pixel to be an array of four unsigned characters, or a struct whose elements are four unsigned characters. You can do this using a type definition in C. For example

```
typedef unsigned char RGBApixel[4];
```

defines the type `RGBApixel` to be an array of four `unsigned char`. If you create a variable `RGBApixel pixel`, then `pixel[0]` will store the red value, `pixel[1]` will store the green value, `pixel[2]` will store the blue value, and `pixel[3]` will store the alpha value for the pixel.

Alternatively, we could use the form

```
typedef struct rgbapixel{
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
} RGBApixel;
```

which defines the type `RGBApixel` to be a struct with four elements, each an `unsigned char`. Using this definition, if you create a variable `RGBApixel pixel`, then `pixel.r` will store the red value, `pixel.g` will store the green value, `pixel.b` will store the blue value, and `pixel.a` will store the alpha value.

In addition, it is a bit cumbersome to have to compute the index `row * width + col` into the pixmap array every time you want to reference an individual pixel at row `row` and column `col`. It would be much easier if we could use a two-dimensional array notation, so that we could index the pixmap array directly using `row` and `col` as array indices. Unfortunately, it is not possible to do this directly in C if the width of the array (i.e. the number of columns) is not known at compile time. When dealing with digital images this it is usually the case that when you are writing your program you will not know the exact image dimensions. This will usually be determined at runtime. Fortunately we can get around this problem by being clever about how we allocate the data structure to store the pixmap.

The trick that we can use is that C actually implements an array using pointer arithmetic. The name of an array is a pointer to the first cell in the block of

memory holding the array. Then, the use of an array index causes the index, times the number of bytes in a cell of the array, to be added to the address of the first cell in the array. So, if we have an array `int array[10];`, then, remembering that an integer takes four bytes in memory, `array[0]` stands for the address of the first cell in the array: `array + 0 * 4`, and `array[2]` stands for the address of the third cell in the array: `array + 2 * 4`.

Here is the construction that we can use. Assuming one of the `typedefs` above for `RGBApixel`, we declare our pixmap variable to be a pointer to a pointer to an `RGBApixel`. Now, assuming that our pixmap has `height` rows, and `width` columns, we allocate an array of `height` pointers to `RGBApixel`, then allocate an array of `width * height` `RGBApixel`, and finally assign to each element `i` of the first array a pointer to the beginning of the `i`'th scanline in the large array. The code for doing this would be:

```
RGBApixel **pixmap;
pixmap = new RGBApixel*[height];
pixmap[0] = new RGBApixel[width * height];
for(int i = 1; i < height; i++)
    pixmap[i] = pixmap[i - 1] + width;
```

The final data structure would look like the diagram shown in Figure 1.12, which shows the layout that would be obtained with `height = 5`, and `width = 6`. Once this allocation is done, all references to the pixmap at row `row` and column `col` would be in the form `pixmap[row][col]`. There is no longer any need to do an explicit calculation of the position of the pixel in the array, as `pixmap[row]` is the address of the cell of the small array corresponding to `row`, and when we index off of this cell by `[col]`, we get the offset to the specified column in the specified row.

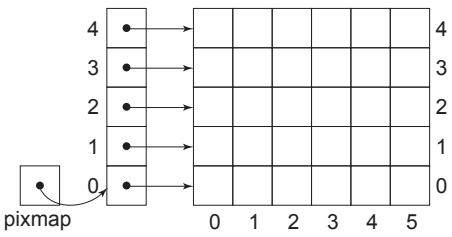

Figure 1.12: Pixmap Organized as a 2D Array