# VIZA 654 / CPSC 646 – The Digital Image Course Notes

Donald H. House
Visualization Laboratory
College of Architecture
Texas A&M Universtiy

2 Sept 2002

# Contents

# Preface

This book would never have been born without the frustrating, volatile, yet somehow wonderful Visualization Sciences Program at Texas A&M University. This program brings artists, scientists, designers and engineers together in a unified curriculum, dedicated to building a core expertise in all aspects of electronic visualization, from image making, to animation, to the science of computer graphics, to videography. Our Program demands that students not only learn to use the latest high-end software for computer graphics production but that they both learn how it works and to do their own algorithm and software development. Our hope is to turn out researchers, developers, artists and designers who have a real appreciation for not only their own specialty but the broad foundations on which their work is based. Students admitted to the program all have shown real excellence in one of the fields of mathematics, computer science, art or design. They have also shown a proclivity and talent for the other aspects of the field, even if they may not have had much earlier formal training. Thus, there is an abundance of talent, at various levels of development. As a computer scientist teaching in this program, my constant challenge is to stimulate, encourage and develop latent technical talents in the artists while at the same time challenging, stretching and expanding the knowledge of the computer scientists.

The reality of today's world is demanding that images move. Making or admiring a single image is not really where it is at in computer graphics these days. The hot topics are animation, morphs, integration of CG with live action, virtual reality – all with an emphasis on action. It is also true that much of the commercial computer graphics work today is not done by three-dimensional animation, but by image manipulation. How are computers used in movie making? Much of the time they are used to manipulate, process, modify, compose, warp, and otherwise manhandle images. 3D computer animation has its uses, and very powerful ones at that, but this has not diminished the use of computers to simply work on the images themselves.

This book, then, is my attempt to collect together a wide variety of both practical and theoretical information on digital images within the context of Computer Graphics, covering their handling, storage, transmission, manipulation and display. This material has been used successfully for eight years now in a course for our students. It is intended to be taught as a project course, with each subject area accompanied by an assignment that requires the students to actually implement algorithms, or experiment with tools and techniques. Students should have access to computers with a good program development environment, supporting the OpenGL or similar graphics standard. They should also have access to PhotoShop or Gimp, and some means of recording reasonable image hardcopy. An ideal system would have a good quality color printer, a 35mm film recorder, and means of recording sequences of images on videotape; although the course can be run well with just the medium quality color printer and the tools and patience to shoot slides from the screen.

The course should be of interest to students majoring in computer science, the biological and physical sciences, engineering, art, architecture, and design. Recommended background would be computer science through the second course (typically Data Structures), and enough mathematical sophistication to understand the concepts of functions, vector algebra and linear transformations in two dimensions.

It is the common wisdom in the Image Processing community that it to really understand the processing of digital images one needs to understand the transformation of images to the frequency domain using fourier analysis. This relegates the study of images to advanced students, with a solid foundation in complex analysis. This book, however, departs from this path, recognizing that there is a wealth of interesting and useful material regarding digital images that does not require such background, and in fact an algorithmic approach can be taken. Instead of starting with frequency domain analysis, the book reserves this material to the final chapters. It attempts to build a strong intuitive foundation, based on the idea that a digital image is a two-dimensional collection of uniform samples, taken through a point-spread function, from a (hypothetical) continuous two-dimensional image. Image reconstruction for display and/or resampling is done through another point-spread function. Without frequency domain analysis it is not possible to present a mathematically rigorous argument for chosing one point-spread function shape over another or for discovering what sampling rate is adequate to prevent artifacts. However, it is possible to make very strong intuitive arguments for good shapes and rates that will work well while minimizing artifacts. In fact an argument can be made, that the approach taken in this book builds a strong intuitive sense of good practice in working with images, that will greatly enhance the student's appreciation for the deep insights that are gained when digital images are studied using fourier analysis.

The course, as it stands, would make an excellent first course in Computer Graphics in a Computer Science program. It can be placed in the curriculum so that it preceeds a traditional course in 3D Modeling and Graphics. If this course were made a prerequisite for the 3D course it would be possible to make the 3D course much more focused on the numerous issues in 3D modeling and graphics that sometimes get lost in a one semester stand-alone course. It would also be possible to design the curriculum so that both courses function independently, which has certain administrative advantages. 3D graphics demands a sophisticated knowledge of data-structures and algorithms, and thus is best taught as an advanced computer science elective. In this course, on the other hand, the algorithms are considerably easier to implement and understand. Yet, the visual results can be as interesting and compelling as those obtained in a 3D graphics course, and the potential audience for the course can be much broader.

# Chapter 1

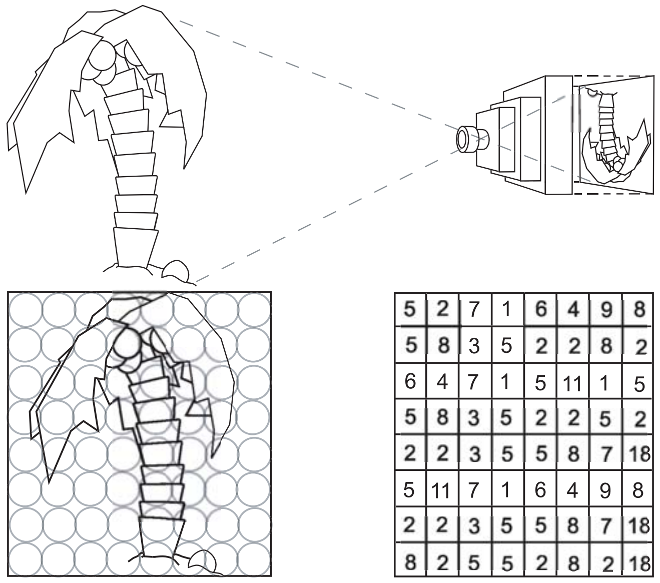
# Digital Image Basics

## 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).



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 7 | 1 | 6 | 4 | 9 | 8 |
| 5 | 8 | 3 | 5 | 2 | 2 | 8 | 2 |
| 6 | 4 | 7 | 1 | 5 | 11 | 1 | 5 |
| 5 | 8 | 3 | 5 | 2 | 2 | 5 | 2 |
| 2 | 2 | 3 | 5 | 5 | 8 | 7 | 18 |
| 5 | 11 | 7 | 1 | 6 | 4 | 9 | 8 |
| 2 | 2 | 3 | 5 | 5 | 8 | 7 | 18 |
| 8 | 2 | 5 | 5 | 2 | 8 | 2 | 18 |

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

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

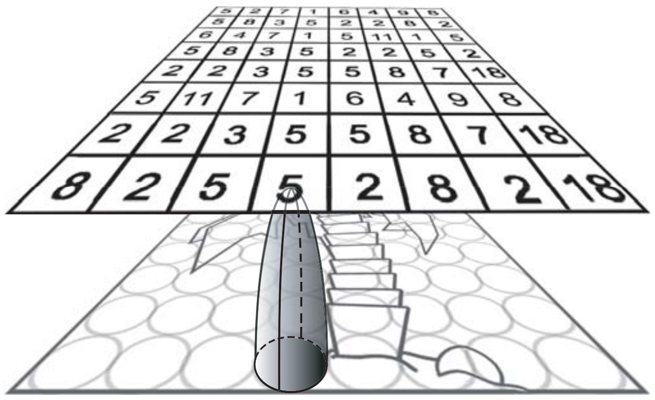


Figure 1.2: Sampling Through a Point-Spread Function

pixel value out over a small region.

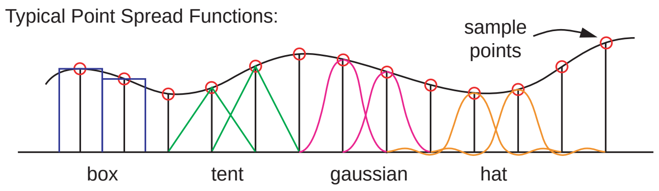


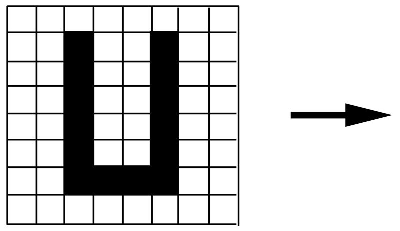
Figure 1.3: Some Typical Point-Spread Functions

## 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 $8 \times 8$ 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



```
1 1 1 1 1 1 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1
1 1 0 0 0 0 1 1
1 1 1 1 1 1 1 1
```

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

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 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 num-

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

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

  int y;                 // scanline number
  int x;                 // pixel number on scanline
  int byte;              // byte number within bitmap array
  int bit;               // bit number within byte
  int value;             // value of bit (0 or 1)

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

Figure 1.5: Procedure to Print the Contents of a Bitmap

bers 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 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

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 |

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.

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

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 |

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 y;              // scanline number
  int x;              // pixel number on scanline
  int value;          // value of pixel (0 to 255)

  for(y = 0; y < height; y++){  // loop for each scanline
    for(x = 0; x < width; x++){   // loop for each pixel on line
      value = greymap[y * width + x];  // 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.

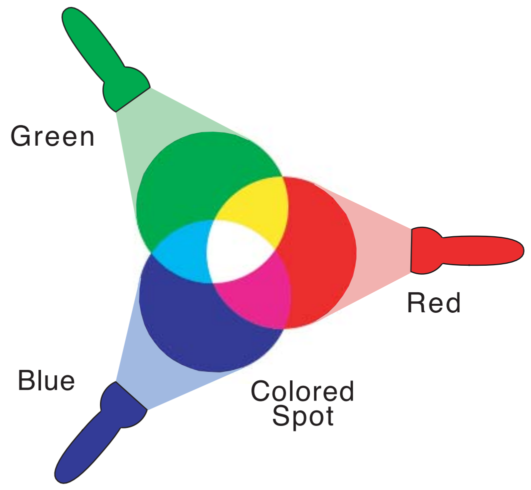


Figure 1.7: Additive Color Mixing for the Red-Green-Blue System

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!

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

| | | |
|---|---|---|
| $(1,0,0)$=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

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

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

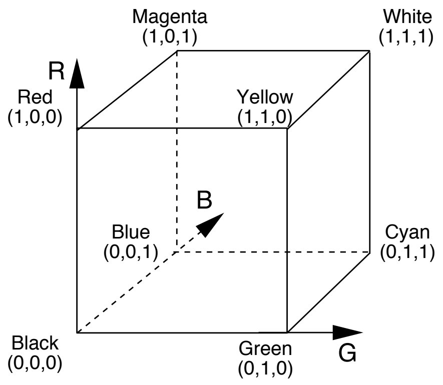


Figure 1.10: RGB Color Cube

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

  for(y = 0; y < height; y++){  // loop for each scanline
    for(x = 0; x < width; x++){   // loop for each pixel on line
      value = pixmap[y * width + x];  // 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

Chapter 2

# Simple Image File Formats

## 2.1 Introduction

The purpose of this lecture is to acquaint you with the simplest ideas in image file format design, and to get you ready for this week's assignment - which is to write a program to read, display, and write a file in the PPM *rawbits* format.

Image file storage is obviously an important issue. A TV resolution greyscale image has about 1/3 million pixels – so a full color RGB image will contain $3 \times 1/3 = 1$ million bytes of color information. Now, at 1,800 frames (or images) per minute in a computer animation, we can expect to use up nearly 2 gigabytes of disk storage for each minute of animation we produce! Fortunately, we can do somewhat better than this using various file compression techniques, but disk storage space remains a crucial issue. Related to the space issue is the speed of access issue – that is, the bigger an image file, the longer it takes to read, write and display.

But, for now let us start with looking at the simplest of formats, before moving on to compression schemes and other issues.

## 2.2 PPM file format

The *PPM*, or *Portable Pixmap*, format was devised to be an intermediate format for use in developing file format conversion systems. Most of you know that there are numerous image file formats, with names like *GIF*, *Targa*, *RLA*, *SGI*, *PICT*, *RLE*, *RLB*, etc. Converting images from one format to another is one of the common tasks in visualization work, since different software packages and hardware units require different file formats. If there were $N$ different file formats, and we wanted to be able to convert any one of these formats into any of the other formats, we would have to have $N \times (N-1)$ conversion programs – or about $N^2$. The PPM idea is that we have one format that any other format can be converted into and then write $N$ programs to convert all formats into PPM and then $N$ more programs to convert PPM files into any format. In this way, we need only $2 \times N$ programs – a huge savings if $N$ is a large number (and it is!).

The PPM format is not intended to be an archival format, so it does not need to be too storage efficient. Thus, it is one of the simplest formats. Nevertheless, it will still serve to illustrate features common to many image file formats.

Most file formats are variants of the organization shown in Figure 2.1. The file will typically contain some indication of the file type, a block of header or control information, and the image description data. The header block contains descriptive information necessary to interpret the data in the image data block. The image data block is usually an encoding of the pixmap or bitmap that describes the image. Some formats are fancier, some are extremely complex, but this is the basic layout. Also, most (but not all) formats have some kind of identifier – called the *magic number* – at the start, that identifies the file type. Often the magic number is not a number at all, but is a string of characters. But in any case, that is what it is called.

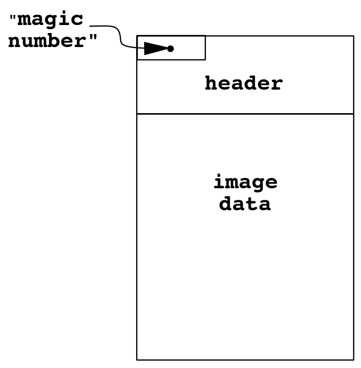


Figure 2.1: Typical Image File Layout

In the PPM format, the magic number is either the ASCII character string `"P1"`, `"P2"`, `"P3"`, `"P4"`, `"P5"`, or `"P6"` depending upon the storage method used. `"P1"` and `"P4"` indicate that the image data is in a bitmap.

These files are called PBM (portable bitmap) files. `"P2"` and `"P5"` are used to indicate greyscale images or PGM (portable greymap) files. `"P3"` and `"P6"` are used to indicate full color PPM (portable pixmap) files. The lower numbers – `"P1"`, `"P2"`, `"P3"` – indicate that the image data is stored as ASCII characters; i.e., all numbers are stored as character strings. This is a real space waster but has the advantage that you can read the file in a text editor. The higher numbers – `"P4"`, `"P5"`, `"P6"` – indicate that image data is stored in a binary encoding – affectionately known as Portable Pixmap *rawbits* format. In our study of the PPM format, we will look only at `"P6"` type files.

### 2.2.1 PPM header block

The header for a PPM file consists of the information shown in Figure 2.2, stored as ASCII characters in consecutive bytes in the file. The image width and height determine the length of a scanline, and the number of scanlines. The maximum color value cannot exceed 255 (8 bits of color information) but may be less, if less than 8 bits of color information per primary are available. In the header, all *white-space* (blanks, carriage returns, newlines, tabs, etc.) is ignored, so the program that writes the file can freely intersperse spaces and line breaks. Exceptions to this are that following an end-of-line character (decimal 10 or hexadecimal 0A) in the PPM header, the character `#` indicates the start of a text comment, and another end-of-line character ends the comment. Also, the maximum color value at the end of the header must be terminated by a single white-space character (typically an end-of-line).

```
P6                     -- magic number
# comment              -- comment lines begin with #
# another comment      -- any number of comment lines
200 300                -- image width & height
255                    -- max color value
```

Figure 2.2: PPM Rawbits Header Block Layout

The PPM P6 data block begins with the first pixel of the top scanline of the image (upper lefthand corner), and pixel data is stored in scanline order from left to right in 3 byte chunks giving the R, G, B values for each pixel, encoded as binary numbers. There is no separator between scanlines, and none is needed as the image width given in the header block exactly determines the number of pixels per scanline. Figure 2.3a shows a red cube on a mid-grey background, and Figure 2.3b gives the first several lines of a hexadecimal *dump* (text display) of the contents of the PPM file describing the image. Each line of this dump has the hexadecimal byte count on the left, followed by 16 bytes of hexadecimal information from the file, and ends with the same 16 bytes of information displayed in ASCII (non-printing characters are displayed using the character `!`). Except for the first line of the dump, which contains the file header information, the ASCII information is meaningless, since the image data in the file is binary encoded. A line in the dump containing only a `*` indicates a sequence of lines all containing exactly the same information as the line above.

## 2.3 Homework Nuts and Bolts

### 2.3.1 OpenGL and PPM

Display of an image using the OpenGL library is done most easily using the procedure `glDrawPixels()`, that takes an image pixmap and displays it in the graphics window. Its calling sequence is

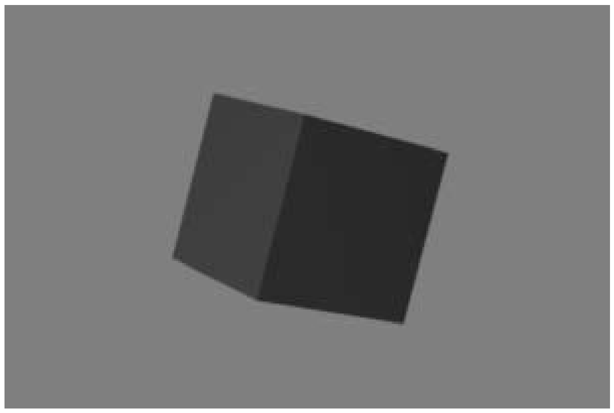
```
glRasterPos2i(0, 0);
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, Pixmap);
```

The call `glRasterPos2i(0, 0)` assures that the image will be drawn in the window starting at the lower left corner (pixel 0 on scanline 0). The `width` and `height` parameters to `glDrawPixels()` specify the width, in pixels, of a scanline, and the height of the image, in number of scanlines. The `GL_RGBA` parameter indicates that the pixels are stored as RGBA quadruples, with color primaries stored in the order red, green, blue, alpha, and the `GL_UNSIGNED_BYTE` parameter indicates that each color primary is stored in a single byte and treated as an unsigned number between zero and 255. Finally, the parameter `Pixmap` is a pointer to an array of integers (`unsigned int *Pixmap`) that is used to store the image's pixmap. Since an `unsigned integer` is 32 bits long, each element of the `Pixmap` array has room for four bytes of information, exactly what is required to store one pixel.

Please make special note of the following complications:

1. By default, OpenGL wants the image raster stored from the bottom scanline of the image to the top scanline, whereas PPM stores the image from the top scanline to the bottom.

2. Each 32 bit `int` in the pixmap stores 1 pixel value in the order R, G, B, $\alpha$. $\alpha$ is an opacity value that you can set to 0 for now. Each of R, G, B, and $\alpha$ take 1 byte.



a) red cube on a midgrey (0.5 0.5 0.5) background

```
000000  5036 0a33 3030 2032 3030 0a32 3535 0a7f  P6!300 200!255!!
000010  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
009870  886d 6d92 5959 8a69 6984 7676 817d 7d80  !mm!YY!ii!vv!}}!
009880  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
009bf0  7f80 7e7e 9d41 41b5 0909 b211 11a9 2424  !!~~!AA!!!!!!!$$
009c00  9f3b 3b94 5454 8b68 6886 7272 827b 7b80  !;;!TT!hh!rr!{{!
009c10  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
009f70  7f7f 7f7f 7f82 7979 a72b 2bb9 0000 ba00  !!!!!!yy!++!!!!!
009f80  00ba 0000 b901 01b7 0606 b20f 0faf 1d1d  !!!!!!!!!!!!!!!!
009f90  a532 3297 4d4d 8d62 6286 7272 827a 7a80  !22!MM!bb!rr!zz!
009fa0  7e7e 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  ~~!!!!!!!!!!!!!!
009fb0  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
00a2f0  7f7f 7f7f 7f7f 7f7f 7f8a 6969 b014 14ba  !!!!!!!!!!ii!!!!
00a300  0000 ba00 00ba 0000 ba00 00ba 0000 ba00  !!!!!!!!!!!!!!!!
00a310  00ba 0000 ba00 00b9 0505 b60d 0daf 1d1d  !!!!!!!!!!!!!!!!
00a320  a62f 2f9d 4141 915b 5b88 6d6d 8279 7980  !//!AA![[!mm!yy!
00a330  7e7e 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  ~~!!!!!!!!!!!!!!
00a340  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
```

b) several lines in a dump of PPM P6 red-cube image file

Figure 2.3: Example PPM P6 Data

3. PPM stores pixel primaries in R, G, B order, and there is no provision for an $\alpha$ value.

### 2.3.2 Logical shifts, and, or

Reading a PPM file and displaying it, requires you to pack the red, green and blue information for a single pixel into one 32 bit unsigned integer. When you have to write the file back out, it is necessary to unpack the red, green and blue components. I recommend that you do this using the following approach

```
int red, green, blue;
unsigned int pixel;

/* packing */
pixel = red << 24 | green << 16 | blue << 8;

/* unpacking */
red = (pixel >> 24);
green = (pixel >> 16) & 0xff;
blue = (pixel >> 8) & 0xff;
```

The operator `<<`, when used in an arithmetic expression, causes the bits in the memory cell specified to the left of the `<<` symbol to be shifted to the left by the number of positions specified to the right of the `<<` symbol. 0's are shifted into the rightmost end of the cell to fill vacated positions. Likewise, when used in an arithmetic expression, `>>` causes a shift to the right by the specified number of bits, with 0's shifted into the leftmost end of the cell. The operator `|` specifies a *logical or* operation between the value specified to its left and the value specified to its right. Similarly, the operator `&` specifies a *logical and* operation between its left and right operands. Both the logical *or* and *and* operations operate bit by bit between corresponding bit positions in their two operands. A logical *or* of two bits yields a 1 if either of the operand bits is a 1, otherwise it yields a 0. In other words, if either one or the other or both operands are 1 the result is 1. A logical *and* of two bits yields a 0 if either of the operand bits is a 0, otherwise it returns a 1. In other words, the result is 1 only if the left operand and the right operand are 1. These operations are diagrammed in Figure 2.4.

There are many practical uses of these logical operations, but for our purposes, the most important is that they allow us to do selective operations on groups of bits (*fields*) within a byte or word. Logical *or* allows

```
        OR               AND

0 | 0 = 0         0 & 0 = 0
0 | 1 = 1         0 & 1 = 0
1 | 0 = 1         1 & 0 = 0
1 | 1 = 1         1 & 1 = 1

  01100000          01101010
| 00001010        & 00001111
  --------          --------
  01101010          00001010
```

Figure 2.4: Logical **or** and **and** Operations

superposition of fields within a word, and logical *and* allows *masking off* of fields, leaving only the values in the bit positions that have been logically *anded* with 1 bits. Figure 2.5 gives a few examples to show how these logical operations work. The examples use only 8 bits for simplicity, but the same principles hold for a 32 bit quantity.

```
unsigned char a = 6;      a:     00000110
unsigned char b = 10;     b:     00001010
unsigned char c = 106;    c:     01101010
unsigned char byte;       byte:  xxxxxxxx
```

a) starting values in variables `a`, `b`, `c`, and `byte`

```
byte = a << 4;              byte:  01100000
byte = (a << 4) | b;        byte:  01101010
byte = c & 0x0f;            byte:  00001010
byte = (c >> 4) & 0x0f;     byte:  00000110
```

b) examples of *shifts*, *or*, *and*

Figure 2.5: Logical *Shift*, *Or* and *And* Operations in C

### 2.3.3 Dynamic memory allocation

You will not know how big to make the pixmap storage array until you have read the PPM Header information to get the image width and height. Once you have that, in `C++` simply do

```
unsigned long *Pixmap;
Pixmap = new unsigned long[width * height];
```

or in traditional C simply do

```
unsigned long *Pixmap;
Pixmap = (unsigned long *)malloc(width * height * sizeof(long));
```

to allocate exactly enough space to store the image when it is read in.

Note that Pixmap will just be a big 1D array, not a 2D array nicely arranged by scanline. Thus, your program will have to figure out where each scanline starts and ends using image width.

### 2.3.4 Reading from the image file

Unless you are quite experienced with C++ I/O facilities for handling files, I recommend that you use the file input/output routines from standard C, rather than using the stream I/O facilities of C++. To make use of them you will have to

```
#include <stdio.h>
```

Once you have opened a file for reading via

```
FILE * infile;
infile = fopen(infilename, "r");
```

you can read individual bytes from the file via

```
int ch;
ch = fgetc(infile);
```

Once you have opened a file for writing via

```
FILE *outfile;
outfile = fopen(outfilename, "w");
```

you can write individual bytes to the file via:

```
int ch;
fputc(ch, outfile);
```

For reading and writing ASCII data from the file, `fscanf()` and `fprintf()` are very handy. Note: `fscanf()` ignores white-space.

### 2.3.5 C command-line interface

The C command-line interface allows you to determine what was typed on the command line to run your program. It works as follows – declare your `main()` procedure like this:

```
int main (int argc, char *argv[]){
   .
   .
   .
}
```

Unix parses the command line and places strings from it into the array `argv`. It also sets `argc` to be the number of strings parsed. If the command line were:

```
ppmview in.ppm out.ppm
```

then the `argv` and `argc` data structures would be built as shown in Figure 2.6. Thus, the file name of the input file would be given by `argv[1]`, and the output file by `argv[2]`. `argv[0]` contains the name of the program itself.

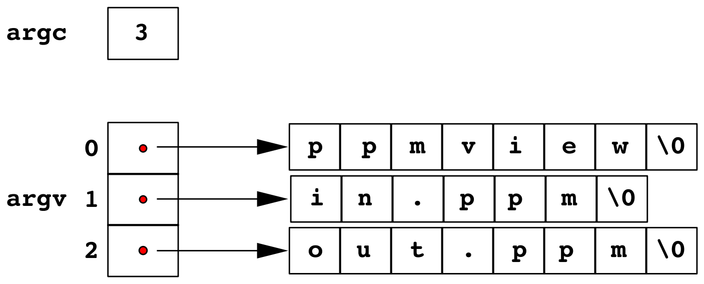


Figure 2.6: argc and argv data structures in C

# Chapter 3

# Display Devices

## 3.1 CRT's - Cathode Ray Tubes

Commonly called a *picture tube*, from the days when most people only encountered them in their television sets, the *cathode ray tube* or *CRT* is the primary device used to display digital images. Before the invention of transistors, chips and integrated circuits, electronic devices depended upon vacuum tubes as “electronic valves” and “switches”. Since CRT's are simply highly specialized vacuum tubes, it will be useful to understand the basic concepts employed in vacuum tubes before studying studying CRT's.

### 3.1.1 Vacuum tube basics

A simple vacuum tube has four active elements enclosed in an evacuated tube, as shown in the schematic diagram of Figure 3.1. Wires extending to the base of the tube allow a voltage to be applied to the heater coil, and a second voltage across the cathode and plate. A control voltage can be applied to the grid. The device can be thought of as a “valve” or a “switch”, since small changes in control voltage can produce a large change in current through the cathode/plate circuit. The tiny control voltage regulates like a valve, or can turn the current on or off like a switch. Here is how it works:

1. The cathode is negatively charged, giving it an excess of electrons

2. The heater heats the cathode, imparting energy that causes the release of some of the excess electrons

3. The free electrons are attracted to the positively charged plate, resulting in a current through the cathode/plate circuit

4. Since the electrical charge on the grid is nearer to the cathode than the plate is, it can prevent electrons from passing on to the plate or "encourage" them. A more positive grid voltage attracts electrons, increasing their acceleration towards the plate. A more negative grid voltage inhibits electrons.

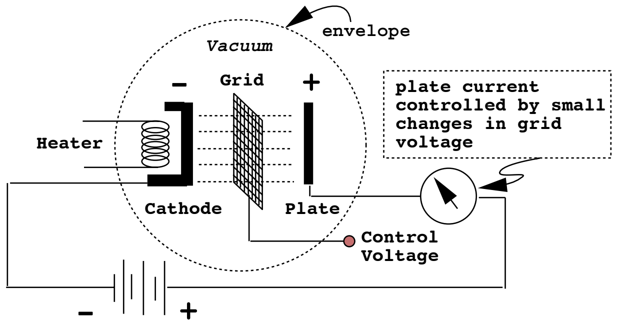


Figure 3.1: Schematic Diagram of Vacuum Tube

### 3.1.2 CRT's

A *CRT* or *Cathode Ray Tube* works on exactly the same principle as a simple vacuum tube but the internal organization is somewhat different. A schematic diagram showing the organization of a simple CRT is shown in Figure 3.2. As the electrons travel from cathode to plate they are focused into a beam and directed onto precise positions on the plate. In a CRT, the plate is a glass screen, coated with phosphor. The phosphor on the screen glows more or less brightly depending on the intensity of the beam impinging on it.

The flow of electrons from cathode to plate works like in a regular vacuum tube. However, focusing coils align the electrons into a beam, like a lens focuses light into a beam. Steering coils push the beam left/right and up/down so that it is directed to a particular spot on the screen. The grid control voltage adjusts the intensity of the beam, and thus the brightness of the glowing phosphor dot where the beam hits the screen.

A CRT can be used to display a picture in two different ways. The electron beam can be directed to "draw" a line-drawing on the screen – much like a high-speed electronic "Etch-a-Sketch". The picture is drawn over and over on the screen at very high speed, giving the illusion of a permanent image. This type of device is known as a *vector display*, and was

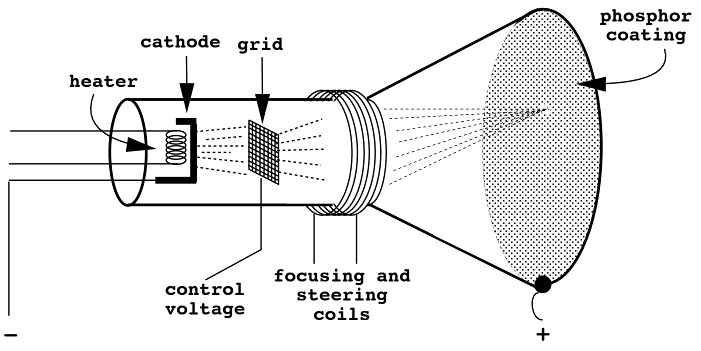


Figure 3.2: Schematic Diagram of CRT

quite popular for use in Computer Graphics and Computer Aided Design up until the early 1980's. By far the most popular type of CRT-based display device today is the *raster display.* They work by scanning the electron beam across the screen in a regular pattern of *scanlines* to "paint" out a picture, as shown in Figure 3.3. As a scanline is traced across the screen by the beam, the beam is modulated proportional to the intended brightness of the corresponding point on the picture. After a scanline is drawn, the beam is turned off, and brought back to the starting point of the next scanline.

The resulting pattern of scanlines is known as a *raster.* The NTSC broadcast TV standard that is used throughout most of America uses 585 scanlines with 486 of these in the visible raster. The extra scanlines are used to transmit additional information, like control signals and closed caption titling, along with the picture. The NTSC standard specifies a *framerate* of 30 frames per second, with each *frame* (single image) broadcast as two *interlaced fields.* The first of each pair of fields contains every even numbered scanline, and the second every odd numbered scanline. In this way the screen is *refreshed* 60 times every second, giving the illusion of a solid flicker-free image. In actuality, most of the screen is blank (or dark) most of the time!

Please see Foley, vanDam, Feiner & Hughes for many more details on CRT's.

A color CRT works like a monochrome CRT, but the tube has three separately controllable electron beams - we say it has three electron *guns.* The screen has dots of red, green and blue colored phosphors, and each of the three beams is calibrated to illuminate only one of the phosphor colors. Thus, even though beams of electrons have no color, we can think of the CRT as having red, green and blue electron guns. Colors are made using

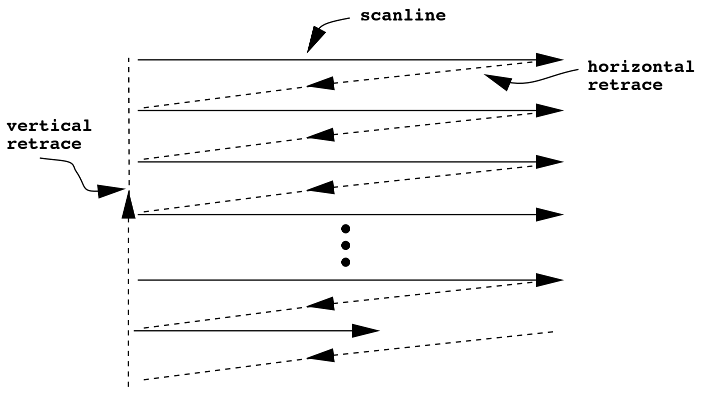


Figure 3.3: Raster Scan Pattern

the RGB system, as optical mixing of the colors of the adjacent tiny dots takes place in the eye.

Figure 3.4 shows the most typical triangular pattern or *triad* arrangement of phosphors on the back of the glass screen of color CRT screens. On the inside of the screen, an opaque *shadow mask* is placed between the three electron guns and the phosphors to assure that each gun excites only the phosphors of its appropriate color. High precision color CRT's, that require the ability to draw a fine horizontal scanline, use an *inline* rather than a triad phosphor arrangement, to keep the scanline confined to a more finely focused vertical area.

## 3.2 Framebuffers

A *framebuffer* is simply an array of computer memory, large enough to hold the color information for one frame (i.e., one screenful), and display hardware to convert the frame into control signals to drive a CRT. The simple framebuffer schematized in Figure 3.5 holds a monochrome (black & white) image in a bitmap. The circuitry that controls the electron gun on the CRT loops through each row of the image array, fetching each pixel value (1 or 0) in turn and using it to either turn on the electron gun for white or turn it off for black. Of course the timing has to be such that the memory fetches and conversion to grid voltages is synchronized exactly with the trace of the beam across the corresponding screen scanline.

Figure 3.6 shows a greyscale framebuffer with three bits per pixel, that uses a DAC (digital to analog converter) to convert numeric grey level to

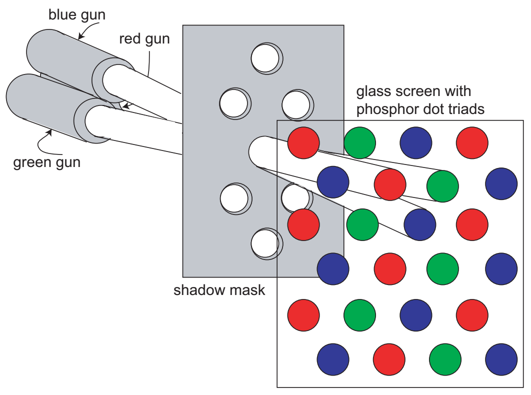


Figure 3.4: Phosphor Triad Arrangement in Color CRT's redrawn from Meko website, *The European Source for Display Data and Market Research*, http://www.meko.co.uk.

one of $2^3 = 8$ different voltages.

Figure 3.7 introduces the notion of a look-up table. Each pixel value from the framebuffer is used as an index into a table of $2^n$ entries (8 in the $n = 3$ example). Each table entry has a stored value whose precision is usually greater than the framebuffer resolution. This gives a palette of only $2^n$ colors, but the palette can be drawn from $2^m$ greylevels, where $m$ is the number of bits per entry in the lookup table. One way this could be used in a greyscale framebuffer would be to correct for non-linearities in the display and in human perception so that each step in grey level would result in a uniform perceptual step in luminance level.

Figure 3.8 shows how a framebuffer can be arranged to drive a color display via three lookup tables. Virtually all 8-bit per pixel color displays – like the ones in older Macintoshes, PC's or workstations – utilize a 24 bit/pixel lookup table with 256 entries ($2^8$). This gives a palette of 256 colors per frame, but the colors are drawn from a selection of nearly 17 million ($2^{24} = 16,777,216$).

A full color resolution framebuffer, called a *truecolor* framebuffer, is shown in Figure 3.9. This type of device will have at least 24 bits per pixel (8 bits per color primary), either driving 3 color guns directly or (as shown in the figure) through a separate lookup table per color primary that can be used to correct nonlinearities or to obtain certain effects (like overlay planes). Very high end graphic displays may have more than 24 bits allocated per pixel, to handle such tasks as color compositing, depth-buffering for hidden surface resolution, double buffering for real-time animated dis-

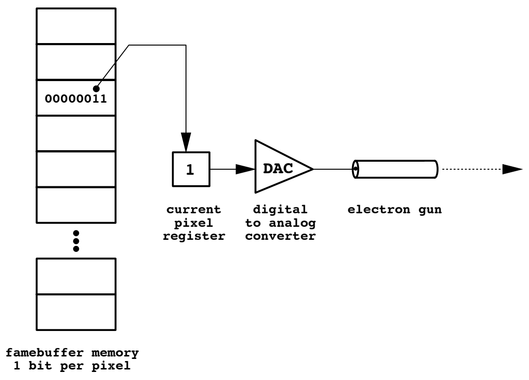


Figure 3.5: Monochrome Framebuffer

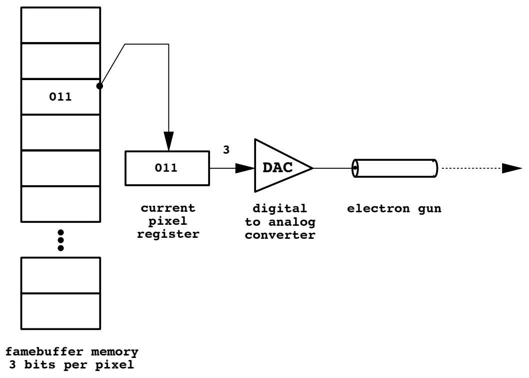


Figure 3.6: 3-Bit Per Pixel Greyscale Framebuffer

play, and overlays.

## 3.3 Gamma Correction

So far we have treated the transfer of information from the framebuffer to the CRT as if it were linear. For example, in an eight-bit grey-scale framebuffer without a lookup table, pixel values can range from 0, representing black, to 255, representing white. Thus, we would expect that a pixel value of 127 would represent a middle grey, and in that even increments of pixel values would represent even increments of grey. In fact, this is not the case.

There are two reasons why even increments of pixel values do not re-

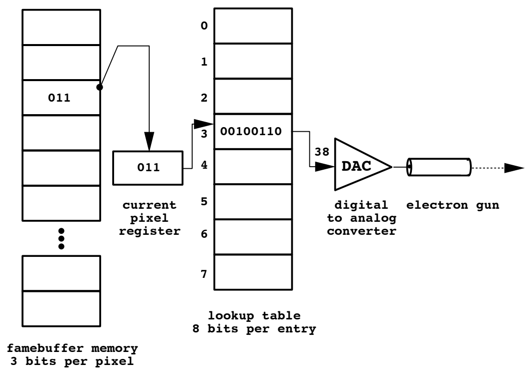

011
011
current pixel register
0
1
2
3
4
5
6
7
00100110
38
DAC
digital to analog converter
electron gun
lookup table 8 bits per entry
famebuffer memory 3 bits per pixel

Figure 3.7: Greyscale Framebuffer with Lookup Table

sult in even increments of perceived grey. The first is that the human eye registers equal increments of color intensity, not as a function of the difference between intensities (luminance), but as a function of the ratio between intensities. In other words, if $I_1 < I_2 < I_3$ are three intensities, the step between $I_2$ and $I_1$ would look the same as the step between $I_3$ and $I_2$ if

$$I_2/I_1 = I_3/I_2,$$

but the steps would look unequal if

$$I_2 - I_1 = I_3 - I_2.$$

Because of the ratio law, in fact, the step between $I_2$ and $I_1$ would appear to be greater than the step between $I_3$ and $I_2$. This relationship is shown in Figure 3.10, which plots perceived intensity $I_p$ in dimensionless units versus actual intensity $I_a$ on a unit scale. The solid line indicates how perceived intensity will vary with actual intensity, and the dashed line shows how perceived intensity would vary if the relationship were linear.

In practice, the story is further complicated by the fact that the relationship between CRT grid voltage and phosphor luminance is also non-linear. The actual phosphor luminance $I_a$ due to grid voltage $I_v$ is given by a curve of the form

$$I_a = I_v^\gamma,$$

where $\gamma$ is a positive constant greater than 1. This relationship is shown in Figure 3.11 for a CRT with $\gamma = 2$. For most CRTs in common usage, values of $\gamma$ range between 1.6 and 2.4.

If we take both the nonlinearity of the CRT and the nonlinear perceptual response into account, the two effects interact to make a more complex relationship. This effect is shown in Figure 3.12.

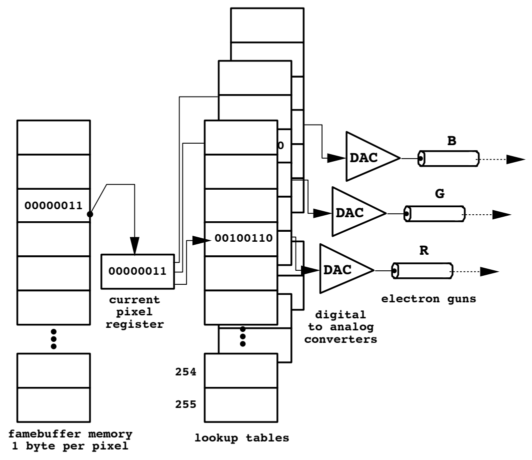


Figure 3.8: 8-Bit Color Framebuffer with 3 Lookup Tables

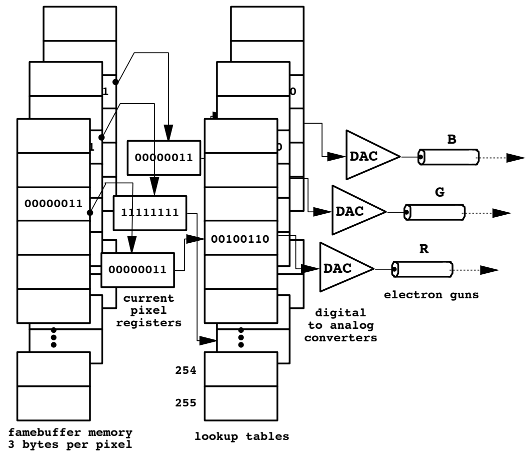


Figure 3.9: Truecolor Framebuffer with Three Lookup Tables

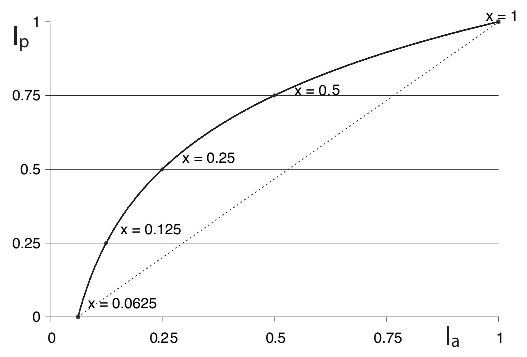


Figure 3.10: Perceived Intensity vs. Actual Intensity

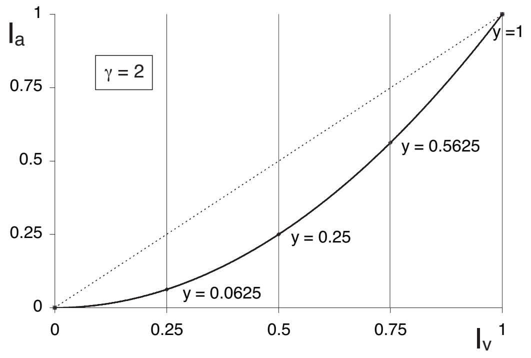


Figure 3.11: Actual Intensity vs. Stored Intensity (voltage), gamma = 2

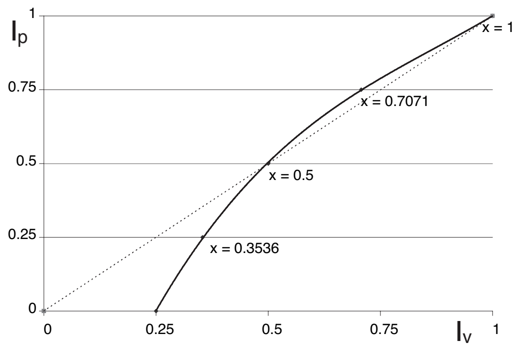


Figure 3.12: Perceived Intensity vs. Stored Intensity (voltage)

The process of correcting for the nonlinearities introduced by the nature of human perception and the physics of CRTs is called *gamma correction*. If we did not have to take the perceptual issue into account, gamma correction would simply consist of scaling each pixel value to the scale 0...1, raising it to the power $1/\gamma$, and then restoring it to its original scale before using it to determine grid voltage. Actually, the problem is complicated by the fact that perceived intensity varies as the ratio of actual intensity. In most cases a suitable value for $\gamma$ can be found experimentally, that will give an even perceived gradation across the full range of greys. However, the $\gamma$ value used for gamma correction and the $\gamma$ of the CRT will differ somewhat.

Gamma correction can be done on a per-pixel basis at the time of display, replacing a stored image with a gamma corrected image. On a framebuffer with a lookup table, however, the lookup table can be loaded with gamma corrected intensity values, that are simply indexed by the colors in the image. This approach has the virtues that it need only be done once, even if we are displaying a sequence of images, that it requires much less computation even for a single image, and that it does not require modifying the image itself. When we study compositing, we will see that pre-gamma correcting images is a very bad idea if we plan on doing any work on the image that requires adding or averaging pixels. We will see that this includes compositing, spray painting, filtering and many other common operations.

# Chapter 4

# Color

## 4.1 What is Color?

There is probably nothing more important to the study of images than the notion of color, what it means exactly, how it can be encoded, and how it can be manipulated. In our attempt to answer the question "what is color?", we will examine several ways of looking at the phenomenon. We will first take the point of view of the physicist, later the physiologist, and finally the artist. Each point of view has its own validity within its own context, and knowledge of each will help us to develop a more complete understanding of this higly complex question.

### 4.1.1 Physical vs. physiological models

The physical model of color is directly related to the physical phenomenon of light, and is actually a way of describing distributions of light energy. Light is energy in the form of electromagnetic waves, with wavelengths in the visible range from 400 to 700 nanometers. The energy of an electromagnetic wave is generally not concentrated at a single wavelength but is distributed across a range of wavelengths. This distribution is known as the *spectrum* of the wave, and it is this spectrum that is interpreted by our eyes as the color of the light. Knowing only this distribution, we can make reasonable predictions about perception of color.

Graphs of two light energy spectra are shown in Figure 4.1. The area under each curve gives the total energy per unit illuminated surface area. This total energy relates directly to the luminance or percieved brightness of the color. Both spectra shown in Figure 4.1 have about the same luminance. The "dominant" wavelength of the curve determines the hue or color name of the color. Both of the spectra in Figure 4.1 have a maximum concentration of energy around 560 nm, and energy at this wavelength is perceived by humans to have a yellow hue. The peakedness of the spectrum, i.e. the percentage of the total energy concentrated in a narrow band around the dominant wavelength, determines the saturation or purity of the color. The spectrum in Figure 4.1 that is labeled "yellow" will look like a fairly pure yellow or orange, whereas the spectrum labeled "brown" will look muddy, like a yellow ochre.

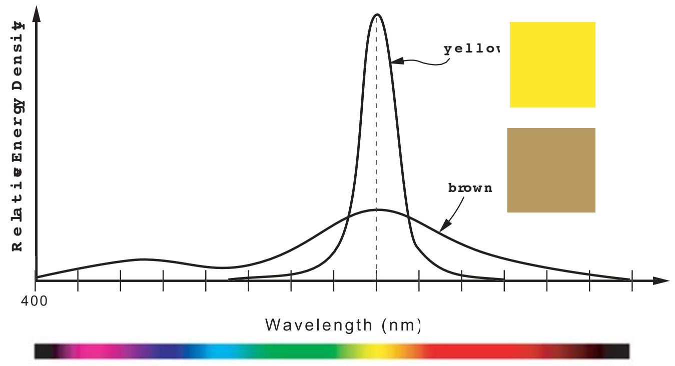
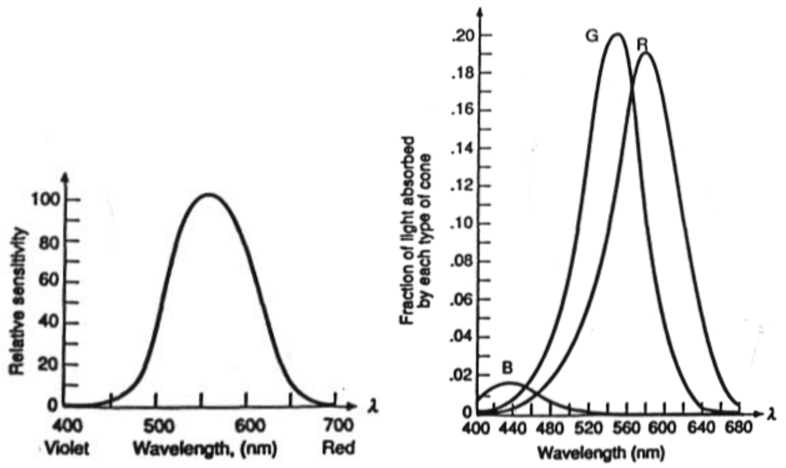


Figure 4.1: Physics of Color

However, from the point of view of the physiology of human vision, we can find no "machinery" in the eye that does anything like measure the shape of light spectra. Instead, perception of color is the result of various mental processes that receive their initial input from broadly-tuned photo sensors in the retina. These sensors are of two main types.

Retinal *rod* cells measure illumination level (grey level) and have a spectral sensitivity or *efficiency function* of the shape shown in Figure 4.1.1a. Even though the rod cells convey no color information, they are maximally sensitive to the greens and yellows. This means that under very low illumination, when there is no color perception, we still see yellow and green objects as being brighter than red or blue objects.

Retinal *cone* cells provide differential measurements of illumination, with spectral tuning that is much finer than that exhibited by rod cells. The cone cells give us the ability to discriminate hue. There are three distinct cone types, that can be thought of as corresponding very roughly to the color primaries red, green, and blue. Figure 4.1.1b shows the relative sensitivities of the three cone types to light across the visible spectrum of wavelengths.



a) rods b) cones

Figure 4.2: Spectral Sensitivity of Retinal Rod and Cone Cells copied from Foley, van Dam, Feiner and Hughes, *Computer Graphics Principles and Practice*, Addison-Wesley, 1990, pg. 577.

However, curves of rod and cone sensitivity are only measures at the eye. They say a lot about input to brain processes, but very little about the use to which the brain puts these inputs. Mental processing is complex and very ill understood. Judging from the amount of brain area devoted to it, visual processing is one of the most complex tasks undertaken by the brain. Thus, we would be making a gross oversimplification of the problem if we tried to relate eye physiology directly to color perception.

### 4.1.2 Color as a contextual phenomenon

The artist, Josef Albers, elevated the art of perceptual color manipulation to a very high level in his development of *color-field* painting. His working thesis was that color is a relative perceptual phenomenon, that transcends attempts to exactly quantify or measure individual colors. In other words, the color we will perceive cannot be predicted outside of the context within which the color will be presented. A memorable painting by Albers is simply a violet square immersed in a field of yellow. On close inspection, however, the violet square is seen to be slate grey. The presentation of the grey square in the context of a brilliant yellow surround, dramatically alters our perception. I will show two simple studies in class, that demonstrate some of the ideas that Albers worked with in his painting. For more on the contextual nature of color, see Joseph Albers, *Interaction of Color*, Yale Univ. Press, 1963.

The point is, that in the end, color perception transcends the physical and the physiological, involving the integration of context into the "reading" of a presentation.

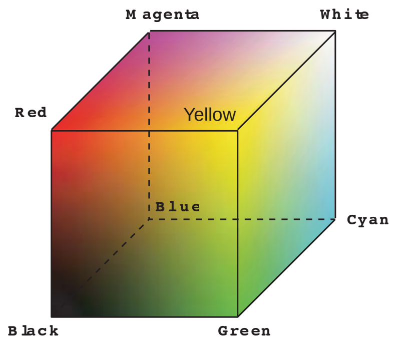
### 4.1.3 Tri-stimulus theory

Nevertheless, there is much evidence that a tri-stimulus theory of color (like the RGB system) provides a very useful color model, and if we ignore contextual issues, allows us to represent specific colors in a very compact form. This form is highly suitable for manipulation in a computer. In general, tri-stimulus theory says that color can be quantified by a 3-parameter system. There is much experimental and experiential evidence to back this theory, and it underlies most of current color technology in the print, broadcast, and film industries. The foundation principle of tri-stimulus theory is that most perceptual colors can be produced by presenting a mixture of three primary colors. A direct and important consequence of this theory is that many different light energy distributions will be read as the same perceptual color.

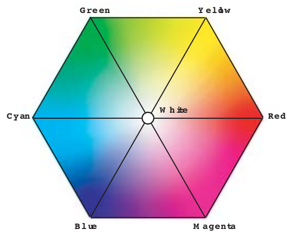
Some color systems extend the three color primary idea, and use a tri-stimulus system, where the three parameters are more abstract color measures.

## 4.2 HSV Color Space

One of the most widely used examples of an abstract tri-stimulus system is the HSV color space. The HSV system attempts to represent all perceptual colors using three measures that relate directly to how artists often think about color. It provides separate measures of *hue* (corresponding to dominant color name), *saturation* (purity of color), and *value* (brightness on grey scale). Its structure is derived directly from the RGB system, and in fact there is a simple translation from RGB to HSV and back. Figure 4.2 diagrams the relationship between the two systems. If the RGB color cube of Figure 4.2a is viewed along its white-black diagonal, it presents the hexagonal silhouette shown in Figure 4.2b. The complete HSV system is a cone-shaped space derived from this projection and shown in Figure 4.4.



a) RGB



b) HSV

Figure 4.3: Relationship Between RGB and HSV Color Systems

The HSV color space is parameterized by color coordinates $(h, s, v)$, standing for hue, saturation and value. This parameterization is shown in the three diagrams of Figure 4.5. Hue $h$ is measured by angular position around the face of the cone. As shown in Figure 4.5a, it goes from 0 to 360 degrees, starting with red at 0°, and proceeding counterclockwise around the color-wheel through yellow at 60°, green at 120°, cyan at 180°, blue at 240° and magenta at 300°. Saturation $s$ is measured by distance from the central axis of the cone. As shown in Figure 4.5b, it is a fraction that goes from 0 for grey at the center to 1 for fully saturated at the boundary of the cone. Value is measured by $v$. As shown in Figure 4.5c, it is a fraction that goes from 0 for black at the apex or point of the cone to 1 for full intensity at the base of the cone, measured along the axis of cone. For example, a full intensity, fully saturated green would be $(120, 1, 1)$. A very dark but fully saturated green would be $(120, 1, 0.3)$, a pastel green would be $(120, 0.3, 1)$, and a neutral green brown would be $(120, 0.3, 0.3)$.

## 4.3 CIE Color Space

The final space we will look at is the CIE system. This system was developed in the 1930's to place the determination of color and illumination

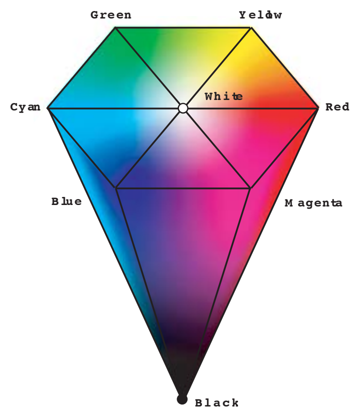


Figure 4.4: HSV Color Cone

on a more scientific basis with respect to human perception. One problem with both the RGB and HSV systems, is that they do not allow for precise color specification because they have no set basis for the colors that can be formed. In other words, the pure red color on one display or printer may look quite different from pure red on another device, although they both have the same RGB specification $(1, 0, 0)$ or HSV specification $(0, 1, 1)$. The CIE system, on the other hand, makes it possible to specify exactly reproducible colors. Thus, CIE colors can be catalogued and then displayed on CRT's, printed in ink, mixed as paint - always giving the same result.

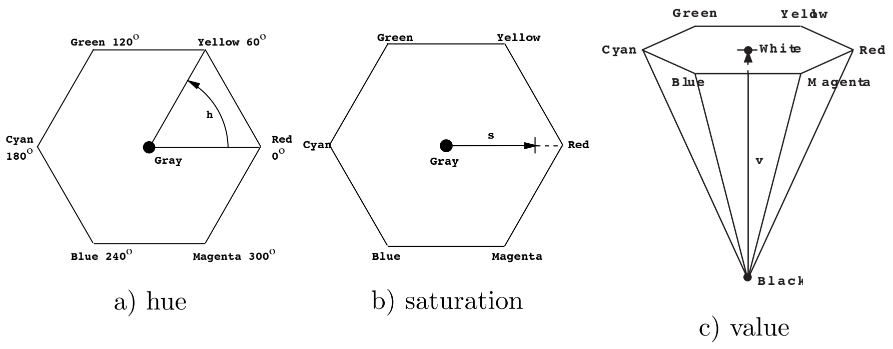


Figure 4.5: Parameterization of HSV Color Space

The CIE system was developed using the experimental configuration shown in Figure 4.6. Observers were presented with a split screen. A colored test light illuminated one side of the screen, and a set of three pure primary lights illuminated the other side. The observer could adjust the intensity of each primary with a dial. The task was to adjust the screen color produced by the three primaries, to match the color projected by the test light.

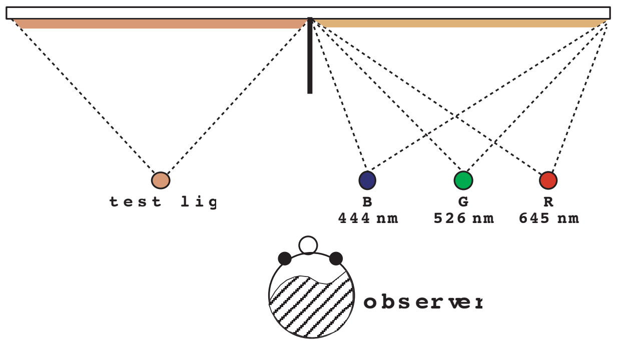


Figure 4.6: CIE Test Apparatus

Using a range of pure single-wavelength test lights, the curves of primary intensities shown in Figure 4.3 were obtained. For each wavelength of test light, the exact setting of each primary necessary to reproduce the perceived color of that test light was determined. A complication was that for some colors the primary settings needed to be negative – in other words it was not possible to reproduce all test colors with the three additive primaries. To deal with this complication, the nonreproducable colors were matched by adding primary colors to the test light. This was considered to be equivalent to subtraction of the corresponding primary from the additive mixture and shows up as negative values on the curves.

From the test data, three functions of wavelength, known as the CIE **x**, **y** and **z** color matching functions were developed. These functions are graphed in Figure 4.3. They do not correspond to realizable light sources, but can be thought of mathematically as if each were a primary light source. The corresponding three light levels of the matching functions necessary to match a given color are called the color's CIE $XYZ$ coordinates. Thus, any color $C$ can be represented by a weighted sum of these 3 primaries

$$C = X\mathbf{x} + Y\mathbf{y} + Z\mathbf{z}. \tag{4.1}$$

Usually, however, the CIE coordinate system is given in modified form,

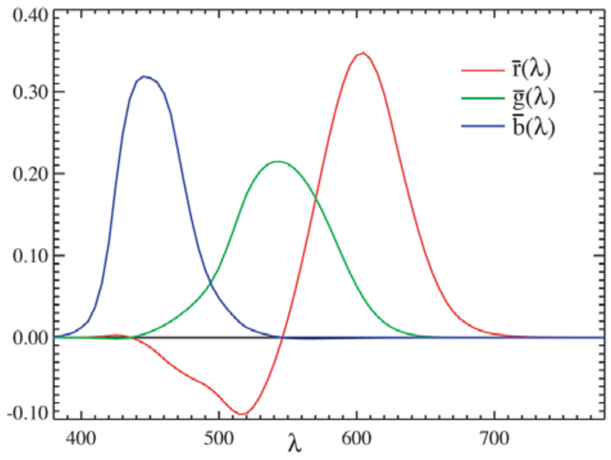


Figure 4.7: CIE Color Matching Experiment Data copied from Wikpedia, *CIE 1931 Color Space*, http://en.wikipedia.org/wiki/CIE_1931_color_space.

by defining the color's chromaticity coordinates $(x, y, z)$. These coordinates measure the color's chromatic content, and are given by

$$x = X/(X + Y + Z), \tag{4.2}$$

$$y = Y/(X + Y + Z), \tag{4.3}$$

$$z = Z/(X + Y + Z) = 1 - (x + y). \tag{4.4}$$

The CIE matching function **y** was chosen to be identical to the human response to luminance (compare Figures 4.1.1a and 4.3). Thus, the CIE Y coordinate does not contribute to the chromatic content of a color and can be thought of as the luminance or brightness of the color. Since the $z$ component of chromaticity can be calculated directly from the $x$ and $y$ components, a unique color specification is given by $(x, y, Y)$. This is known as the CIE xyY color specification. It has the advantage that all of the chromatic information is contained in the coordinate pair (x,y) and all of the luminance information is given by one coordinate Y. A cross section through the CIE xyY space for a fixed luminance (i.e. a fixed value of Y) looks like that shown in Figure 4.3.

There are methods of going from a catalogued CIE color to an RGB triple, that require knowing some characteristics of the display device (e.g. the CIE coordinates of the phosphors of a CRT). The science of using CIE

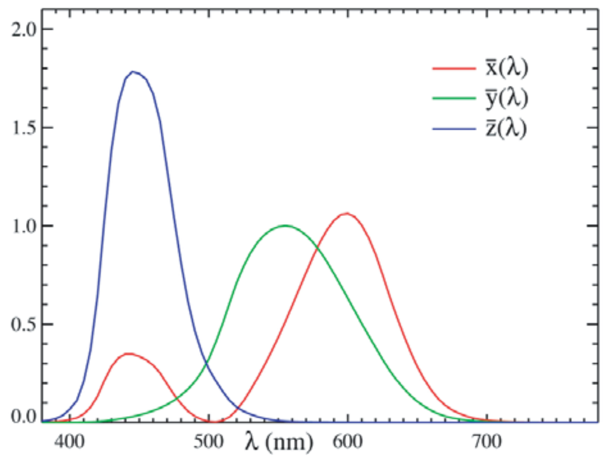


Figure 4.8: The CIE XYZ Color Matching Functions
copied from Wikpedia, *CIE 1931 Color Space*,
http://en.wikipedia.org/wiki/CIE_1931_color_space.

information fits into the broad area of colorimetry, and is something that you may want to study further if you have a deep interest in color.

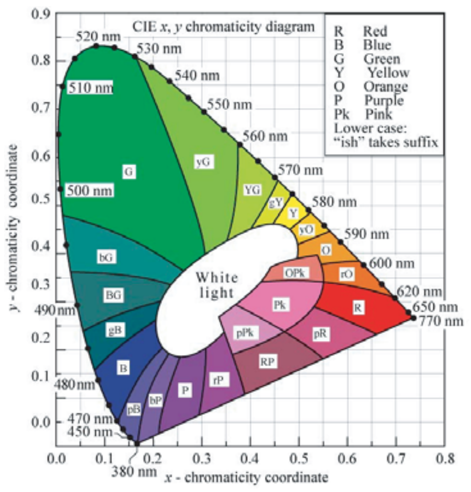


Figure 4.9: Cross Section of the CIE Space for Fixed Luminance Y from Schubert, E.F., *Light Emitting Diodes*, Colorimetry, 2003. ¡http://www.ecse.rpi.edu/ schubert/Light-Emitting-Diodes-dot-org/¿

# Chapter 5

# Simple Image File Compression Schemes

We have noted earlier that one of the big issues that we will need to deal with in treating the subject of image files is file size. What ways are available to store accurate image information in a file, while minimizing the storage used for the file and at the same time keeping file access time to a minimum? This issue is clearly of importance in the animation and film industries, where it is often necessary to store many thousands of images to fully describe a scene. It is also of crucial importance in multimedia, where it is often necessary to squeeze the maximum out of the limited storage capacity of a CD-ROM or other personal computer storage medium. And, not of least significance, image storage size has crucial implications for the speed of transmittal of the image between computers, especially within the context of visually-oriented computer networking such as is done over the Internet using the World Wide Web. The formula for transmission time is quite simple – it is exactly proportional to the size of the file as it is encoded for transmission.

All compression schemes exploit the fact that most images contain a good deal of redundant information, and that this redundancy manifests itself in various kinds of *coherency* in the image. We will define coherency as the tendency for one portion of an image to be similar to other portions. Often coherency is most apparent in regions of the image that are physically near each other. For example, a digital image containing a pale yellow wall in the background will have broad expanses of nearly identical colors, and in fact it may have many thousands of pixels with exactly the same color value. This is the simplest kind of coherency – many copies of the same color. Coherency, however, can be a more sophisticated concept. For example, we could have the same wall, but now with a light gradation over its surface, so that we have a smooth and predictable variation in light intensity (value) across the wall. Another more sophisticated example could be a picture containing a wall in the background that is covered with a patterned wallpaper, so that we have large regions that contain the same repeating texture. There might be schemes for exploiting this coherency, for big space savings.

In this chapter, we will look at several of the simpler file compression schemes, namely run-length encoding, the use of color tables, and a more sophisticated table-based encoding scheme. In a later chapter, we will look at two much more powerful but also much more complex schemes – the JPEG protocol and Wavelet compression.

## 5.1 Run-Length Encoding

Run-length encoding, in its simplest form, takes advantage of color coherency along a scanline. It is a readily observable fact that in certain images there are many regions along a scanline where adjacent color values are repeated across several pixels. This group of identical pixels, taken together, is called a *run*. Runs appear frequently in synthesized computer graphic images, where it is not unusual for large areas to tend to have the same color. Runs appear less frequently in photographic images, especially when they are of natural scenes.

In the simplest run-length encoding scheme, instead of storing each pixel value along a scanline, a pair of numbers is stored for each run along the scanline. The first number in the pair is a repeat count, and the second is the pixel value. For example, suppose we had the scanline

2 2 2 2 2 3 4 1 1 1

which is ten pixels long. The run-length encoding of this scanline could be written

(5 2) (1 3) (1 4) (3 1)

which contains one pair for each of the four runs in the scanline, requiring only eight distinct values, and saving two units of storage. (Note that in the above example, the parentheses are used only for clarity, and would not actually need to be be stored.) The savings of 20% in total space is about what can be expected for a typical computer graphic image. However, there are no guarantees. In the worst case, where no adjacent pixels have the same color (i.e. all runs are of length one) this scheme actually doubles the space used, since a pair of numbers would be required to represent each pixel!

Fortunately, there is an easy improvement that can be made on the simple run-length encoding scheme, that solves this problem. The trick is to somehow "flag" the repeat count, such that the repeat count without the flag is just the same as in the simple scheme, but the count with the flag is used to prefix a string of pixels that have different values. Then to encode the scanline, each run of length two or more would be output using standard run-length encoding, but all consecutive runs of length one would be grouped together and output as a group prefixed by a flagged repeat count. Using this scheme, the scanline from the example above would be encoded

$$(5\ 2)\ (\overline{2}\ 3\ 4)\ (3\ 1)$$

which requires only seven values to encode the original ten pixels. This line would be decoded as follows:

$(5\ 2) \Rightarrow 2\ 2\ 2\ 2\ 2$

$(\overline{2}\ 3\ 4) \Rightarrow 3\ 4$ *the flag indicates that 2 explicitly given values follow*

$(3\ 1) \Rightarrow 1\ 1\ 1$

With this modified runlength encoding scheme, we add only one extra value per scanline, even in the worst case when every pixel has a different value. In actual implementation, worst-case performance might be a bit worse than this, since we will use a limited storage space (typically one byte) for the runlength count, so that the length of a run or an unencoded group of pixels will typically be limited to less than the length of a scanline. Many popular computer graphics image file formats use this type of run-length encoding scheme. Important examples are wavefront RLA (`.rla`) files, Silicon Graphics Iris RGB (`.rgb` or `.sgi`), and Softimage Picture files (`.pic`).

### 5.1.1 Silicon Graphics Iris RGB File Format

Silicon Graphics developed its own image file format, known as the Iris RGB file format, and supported by the SGI GL 3D interactive graphics standard. It is not the standard in OpenGL (which supports no particular file format), but nevertheless, the Iris image file format remains in wide use. Iris RGB files typically carry the file name suffix `.rgb`, although the suffix `.sgi` is also common. An Iris RGB file can be written in either *verbatim* or run-length encoded modes. Overall file layout is shown in Figure 5.1, and depends upon which of these two modes is chosen. The verbatim file format of Figure 5.1a contains only the header and the uncompressed image data.

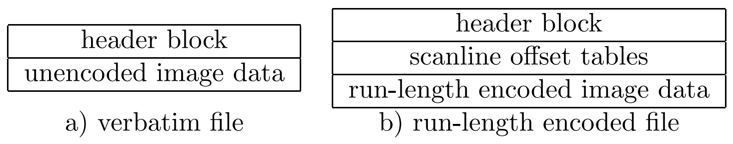


Figure 5.1: SGI Iris RGB File Layout

A run-length encoded file of Figure 5.1b contains tables giving scanline offsets, in addition to the header and the run-length encoded image data.

The header is laid out according to the `C++` data-structure shown in Figure 5.2. The magic number of an Iris RGB file is $01DA_{16}$, and is stored in the first two bytes of the file. The `storage` byte indicates whether the file is stored verbatim or run-length encoded. The image is organized into channels, with one channel per color primary and additional channels for any auxilliary values, such as an alpha value. The `bpc` byte indicates whether one or two bytes per pixel are used for channel data. The `dimensions` byte indicates how many scanlines and channels are stored in the file, and will normally be set to 3, indicating that the `ysize` and `zsize` entries below give the number of scanlines and channels. If `dimensions` is set to 1, the image file consists of just one scanline and one channel, so `ysize` and `zsize` are ignored. If `dimensions` is set to 2 there is only one channel, so `zsize` is ignored. The `xsize`, `ysize` and `zsize` entries are each two bytes long, and give the scanline length and image height in pixels, and the number of channels. The `pixmin` and `pixmax` entries are each four bytes long, and are used to hold the minimum and the maximum channel values in the image. The `imagename` entry is an 80 byte area that is used to store any text description of the image that is desired. Typically this will store the filename of the image. The `colormap` entry is normally set to 0. Non-zero entries were used in the past to give indexed colormap information for files with pixels stored as indices into a color table.

## Verbatim format

Image data in verbatim Iris RGB files is stored starting with the bottom image scanline, working to the top of the image. Thus, the first pixel stored is in the bottom lefthand corner and the last stored is in the upper righthand corner. All of the entries for channel 1 are stored, then all of the entries for channel 2, etc. For a four channel RGB$\alpha$ image, channel 1 is used for red, 2 for green, 3 for blue and 4 for $\alpha$. Channel data will be stored using either one or two bytes per pixel, depending on the value of

```
struct Iris_Header{
  short magic;          // 01DA(hex) Iris RGB magic number
  char storage;         // 0 = verbatim, 1 = run length encoded
  char bpc;             // 1/2 bytes per pixel per channel
  short dimensions;     // 1 = 1 scanline, one channel
                        // 2 = ysize scanlines, one channel
                        // 3 = ysize scanlines, zsize channels
  unsigned short xsize; // image (scanline) width in pixels
  unsigned short ysize; // image height (number of scanlines)
  unsigned short zsize; // number of channels
  long pixmin;          // minimum channel value in image
  long pixmax;          // maximum channel value in image
  long unused;          // unused spacer
  char imagename[80];   // ASCII text describing image
  long colormap;        // 0 = normal, others values obsolete
  char spacer[404];     // unused spacer
};
```

Figure 5.2: SGI Iris RGB File Header Format

the `bpc` entry in the file header.

### Run-length encoded format

Scanlines of image data in run-length encoded Iris RGB files can be stored in any order. However, access to the scanline data is done through the *scanline offset tables*. There are two such tables, immediately following the file header. The first is the *scanline start table*, which gives the location in the file (i.e. the starting byte number from the start of the file) for each channel of each scanline. The second table is the *scanline length table*, which give the *encoded* length of each channel of each scanline. There are four bytes per table entry, i.e. each entry is an `unsigned long`. Both tables are ordered using the same ordering scheme as for verbatim image data. Starting from the bottom of the image, there will be one entry per scanline for channel 1, then channel 2, etc.

Channel data is run-length encoded on a scanline by scanline basis. Encoding is done using the scheme shown in Figure 5.3. Each channel is run-length encoded using an unsigned 8-bit[1] (one byte) value for the repeat

[1]Recall that in 8 bits there is the possibility to store $2^8 = 256$ different values. For

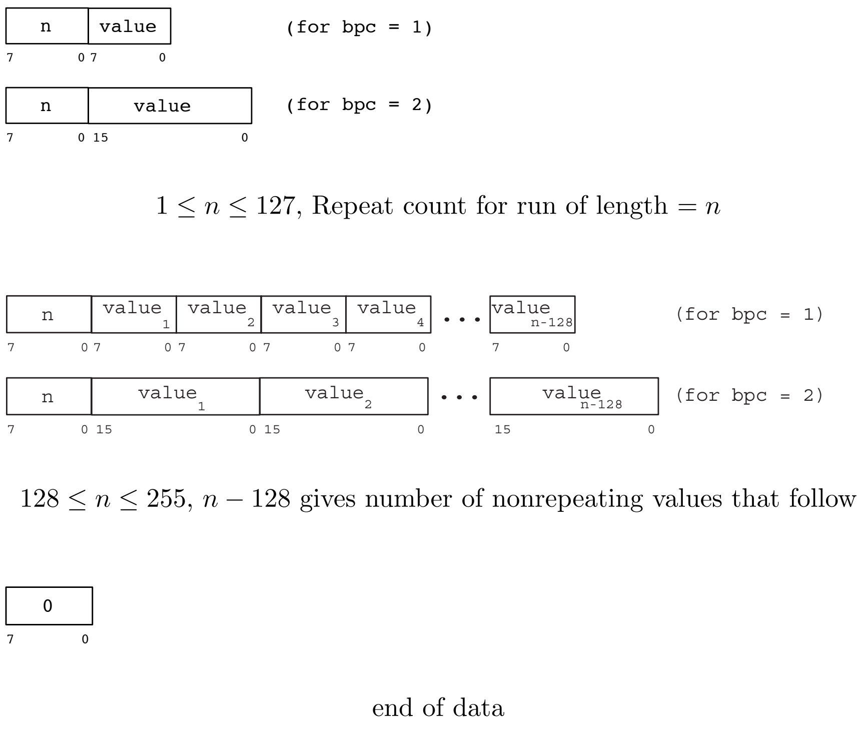

n
value
(for bpc = 1)
(for bpc = 2)
$1 \leq n \leq 127$, Repeat count for run of length $= n$
value 1, value 2, value 3, value 4 ... value n-128
$128 \leq n \leq 255$, $n - 128$ gives number of nonrepeating values that follow
0
end of data

Figure 5.3: SGI Iris RGB Run-length Encoding Scheme

count. If this value is between 1 and 127 it stands for the length of a run, and is followed by a one or two byte channel value that is to be repeated to form the run. Thus, if this repeat count is 10, it means that the channel value following the count should be repeated 10 times to form a run of 10 identical values. However, if the repeat count is between 128 and 255 the count is followed by a sequence of nonrepeating channel values. The number of nonrepeating values will be the count minus 128. A repeat count of 0, indicating end of data, is used to mark the end of each scanline.

### 5.1.2 SoftImage Picture Files

The SoftImage animation system also uses its own image file format, known as the SoftImage Picture format. Files in this format usually carry the filename suffix `.pic`. In Picture files, image compression is done using the

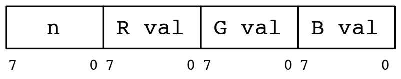
unsigned 8 bit number's, this available space is used to encode values from 0 to 255.

run-length encoding scheme shown in Figure 5.4. Unlike the Wavefront RLA scheme, the Picture file scheme treats all count bytes as unsigned numbers. In this scheme, the run-length count is stored in either an eight or a 24 bit field. A non-repeating string of explicitly stored values is preceded by a single byte count that must be a number between one and 127, that indicates the length of the nonrepeating sequence minus one. A short run, runs of between two and 128 pixels, are encoded by preceding the repeated value by a single byte-count between 129 and 255, which is interpreted as the length of the run plus 127. Longer runs of from 128 to 65,535 pixels are encoded by a single byte containing the value 128, followed by a two byte field giving the explicit repeat count. The scheme is made a bit more complicated by the fact that encoded values may be either 24 bit RGB triples or single byte values. Generally, in a full RGB image with an alpha channel, the RGB values will be encoded as triples, and the alpha values will be single bytes encoded as a separate channel from the color information. A channel information block in the image file tells the decoding program how to interpret the channels of image data.

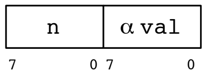
SoftImage Picture files are organized into a header and an image data section. The header has two parts, the picture information block and the channel information block, as shown in Figure 5.5.

`C++` code detailing the picture information block is shown in Figure 5.6, and is reasonably self explanatory. The magic number for a Picture file is $5380F634_{16}$ and is stored in the first four bytes of the file. Floats are stored as four-byte SGI format floating point numbers. The `aspect` field is typically 1.0. The `fields_code` field will be 0 if there is no image data, 1 if only the odd numbered scanlines are stored for the image, 2 if only the even numbered scanlines are stored, and 3 if all scanlines for the image are stored. The channel information section specifies what channels are present in the image data, and how they are organized. Usually, the R, G and B channels are grouped together into 24 bit values per pixel and the alpha values (if present) occupy a second group.

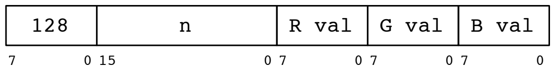
The channel information block immediately follows the picture information block in the file, and has one entry per channel of image data. The format of an entry in the channel information block is is four bytes long, with the format shown in Figure 5.7. The `chained` byte is set to 1 except for the last entry. So if there are two image channels, the first entry would contain a 1 in the first byte, and the second entry would contain a 0. The `size` byte will typically be set to 8, indicating the number of bits allocated to each value in a channel. The `type` field indicates whether or not the channel is run-length encoded. The four upper bits of the `primaries` byte are used to indicate which of the three color primaries or alpha are encoded in the channel. Typically, this byte will be set to `0xe0` to indicate that the



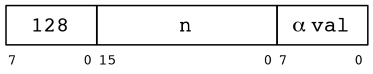
or



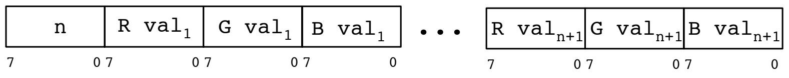
$129 \leq n \leq 255$, Repeat count for short run of length $= n - 127$ (i.e. $2 \cdots 128$)



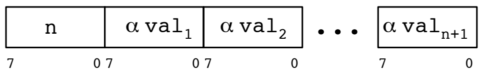
or



$129 \leq n \leq 65,535$, Repeat count for long run of length $= n$



or



$0 \leq n \leq 127$, $n + 1$ gives number of nonrepeating values that follow

Figure 5.4: Softimage Picture File Runlength Encoding Scheme

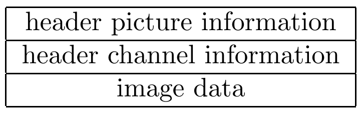


Figure 5.5: SoftImage Picture File Format

```
struct PIC_Header{
  long magic;              // 0x5380f634 -- PIC magic number
  float version;           // PIC file format version number
  char comment[80];        // any text describing the image
  char id[4];              // "PICT"
  short width;             // image width in pixels
  short height;            // image height in scanlines
  float aspect;            // pixel aspect ratio (width/height)
  short fields_code;       // 3 for full frame image
  short unused;            // should be 0 (but might not be!)
};
```

Figure 5.6: SoftImage Picture File Picture Information Block

```
typedef struct{
  unsigned char chained;   // 0 = last, 1 = more channels follow
  unsigned char size;      // 8 -- number of bits/channel value
  unsigned char type;      // 2 = run-length encoded, 0 unencoded
  unsigned char primaries; // bits indicate R, G, B, alpha
                           // R 0x80, G 0x40, B 0x20, alpha 0x10
}PIC_CHANNEL_INFO;
```

Figure 5.7: SoftImage Picture File Channel Information Block

channel contains R, G and B data, or to `0x10` to indicate that it contains alpha data only.

The image data immediately follows the channel information block, and is organized by scanline, with the top scanline of the image appearing first in the file. Each scanline is encoded in channels as indicated by the channel information block, typically as an RGB channel followed by an alpha channel, as shown in Figure 5.8.

## 5.2 Color Tables

Another compression scheme that can be quite effective, if a picture is known to have a limited color palette, is to store a table of the RGB colors in the image file, and then for each pixel store the index to the table entry that holds the RGB color for that pixel. For example, if an image has 256

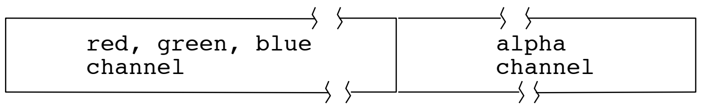


Figure 5.8: SoftImage Picture File Scanline Organization

or less colors, a one byte (eight bit) entry for each pixel will suffice to index into a table containing full 24 bit RGB information. This reduces the image file size by 2/3, less the space required to store the 256 table entries.

This scheme can even work for images with large numbers of colors, if one is willing to take some degradation of image quality. There are various ways to *quantize* an image to reduce the number of distinct colors so that the color table method is effective. A simple way of reducing the number of bits per pixel is by discarding low order bits for each color primary. This essentially "throws away" the least significant part of the primary color value, leaving a picture with a reduced color palette. For example, discarding the bottom two bits for each primary will reduce the potential total number of different colors by a factor of $2^2 = 4$. For natural scenes, with high image complexity, this often is hardly noticable. However, for computer graphic images, especially ones with smoothly varying color gradations, color quantization leads to very noticable *banding* of colors. To avoid artifacts such as banding, more sophisticated techniques for reducing the number of colors in an image can be used. These schemes attempt to choose the colors that will be preserved in an optimal way. One example is a scheme for choosing those elements of the RGB color cube which approximate the colors in the original image with the least *variance* between the original and quantized images.

## 5.3 Lempel–Ziv–Welch Encoding

Another very popular compression scheme is also based on a table, but it involves 1) no loss of data, and 2) no need to store the table! It is known as the Lempel-Ziv-Welch (LZW) algorithm and forms the basis of the well-known Unix `compress` utility. In a nutshell, this technique works by "discovering" and remembering patterns of colors (or any other data you may want to compress), and storing these patterns in a table – then only table indices are stored in the file. Unlike the color table technique, table entries can grow arbitrarily long, so that one table index can stand for a long string of data in the file. And, by a clever construction, it turns out that the table itself never needs to be stored in the file.

### 5.3.1 The LZW encoding algorithm

To see how the LZW encoding algorithm works, we will follow a simple example. To begin with, let us assume that there is a fixed number of color values (like 256 for 8-bit pixels). For our example, let us represent a space of only 4 possible colors – A, B, C, or D. Allocate a table called `T`, a current-input pixel register called `in`, a current table index register called `index`, and a string variable called `prefix`, and consider the example input scanline

| | | | | | | |
|---|---|---|---|---|---|---|
| A | B | A | C | A | B | A |

Then the algorithm works like this

1. Initialize the table `T` with the possible 1-pixel color values, and the `prefix` variable with the empty string.

| T | | | | |
|---|---|---|---|---|
| value | A | B | C | D |
| index | 0 | 1 | 2 | 3 |

| | |
|---|---|
| `in` | |
| `index` | |
| `prefix` | `""` |

```
2. while (input not exhausted)
     a load input register in with next input pixel value
     b append in to string in prefix
     c if (prefix not in table T)
         i output index
         ii insert prefix in table T
         iii set prefix = in
     d set index = position of prefix in table
3. output index
```

Figure 5.9 shows how this algorithm would progress when run on the sample input. In this example, it compresses the original seven input pixels into only 6 output values.

### 5.3.2 LZW Decoding Algorithm

The genius of Lempel-Ziv is in the decoding algorithm. This converts the encoded file back into the original sequence, without having the table that was built upon encoding. In fact, the decoding algorithm recreates this table as it runs.

The algorithm utilizes the following logic:

**Sequence of Prefixes and Outputs**

| `prefix` | `in` | output |
|---|---|---|
| `""` | A | - |
| A | B | 0 |
| B | A | 1 |
| A | C | 0 |
| C | A | 2 |
| A | B | - |
| AB | A | 4 |
| A | - | 0 |

**Resulting Table `T`**

| index | value |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | AB |
| 5 | BA |
| 6 | AC |
| 7 | CA |
| 8 | ABA |

Figure 5.9: LZW Encoding of the Input String: A B A C A B A

1. We know the first, 1-pixel, table entries. In our example, these were (0, A) (1, B) (2, C) (3, D). Thus, if we read any input from the encoded file that has a value in this range (e.g. 0 through 3), we can directly translate it.
2. We keep track of the previous matched table entry, and when we input the next code from the file, note whether it is either in the table already or it is not.
3. If the new code is in the table, then we know what it stands for, and we also know that the previously translated code, plus the first character of this code must be the next entry to be added to the table if it is not already there.
4. If the new code is not in the table, then the previously translated code, with its own first character appended to the end, is the translation for the input, and must also be the next entry to be added to the table.

In pseudocode form, the LZW decoding algorithm is given by:

```
1. Initialize the table T, e.g. T = [(0, A) (1, B) (2, C) (3, D)]
2. Read 1st value from the file, code = getcode()
3. Output(T[code])
4. oldcode = code
5. for(code = getcode(); code ≠ End-of-File; code = getcode())
   a if(code < tablelength) /* i.e.  code already in T */
       i output(T[code]);
       ii Insert T[oldcode] with T[code][0] appended, into table T
   b else /* code not in table yet */
       i translation = T[oldcode] with T[oldcode][0] appended
       ii output(translation)
       iii insert translation into table T
   c oldcode = code
```

Figure 5.10 shows how the example encoded file from Figure 5.9 would be decoded. Recall that the encoded file contains the sequence (0 1 0 2 4 0), which is the encoding of the original scanline (A B A C A B A).

| **Sequence of Inputs, Outputs, Actions** | | | |
|---|---|---|---|
| `oldcode` | `code` | translation | action |
| - | 0 | A | `T` = [(0,A) (1, B) (2, C) (3, D)] |
| 0 | 1 | B | add AB to `T` |
| 1 | 0 | A | add BA to `T` |
| 0 | 2 | C | add AC to `T` |
| 2 | 4 | AB | add CA to `T` |
| 4 | 0 | A | add ABA to `T` |
| 0 | end of file | — | — |

Figure 5.10: LZW Decoding of the Encoded String: 0 1 0 2 4 0

The way to understand why this algorithm works correctly, is to think about how the original encoding algorithm builds its table. Imagine the decoding algorithm as using exactly the same table building logic. However, instead of reading pixels directly from the input file, it gets its input pixels from reading codes and decoding them via the partially built table. As long as a code that is read is in the table already, simply take its translation pixel by pixel and feed it to the encoding algorithm's table building logic. The only time this has a problem is when a code is read that is not in the table. In this case, we know that the missing code must be the next one to be added to the table, and even better, we even have the missing entry partially built. In the encoding algorithm, the partially built table entry is the "prefix" string. Equivalently, in the decoding algorithm, it is the table entry for `oldcode`. So, instead of being stopped by the missing table entry, we can proceed on using the characters in the partially built new table entry. If we do this, we see that the first character in the partially built entry (i.e. the first character of the translation for `oldcode`) becomes the last character of the new entry that gets added to the table. Working through the following simple example will demonstrate this to you.

Try the following – both encode and decode the sequence (A B A B A B A)[1]. You should get the encoding (0 1 4 6), reducing the original seven input values to four output values. When you try decoding this sequence, you will see that the code 6 is read before the corresponding table entry is built. Note that results obtained with the LZW algorithm are more impressive when there are numerous repeated patterns in the input.

[1]also try to pronounce it :)

### 5.3.3 GIF - Graphics Interchange File Format

The GIF format was developed by CompuServe, Inc. to meet the demand for a highly compressed image file format for storage in remote archives and downloading over networks. The format assumes that display devices are at most 8-bit/pixel resolution, and employ a color-lookup-table. Accordingly, GIF stores a color table along with an image, and the image itself is stored as a sequence of indices into the color table. GIF gets further compression by using LZW encoding to compress the image data.

A GIF file for a single RGB image is organized as follows:

1. Three bytes containing the magic number "GIF"

2. Three bytes containing the GIF version number. This defines the GIF standard employed by the file, it typically is "87a".

3. Seven bytes containing the *logical screen descriptor* for the file. The first four bytes contain the window width and height, two bytes provide information on color table format, and the last byte is the color table index for the background color. Note, the window width and height are stored in *byte reversed* form, least significant byte followed by most significant byte.

4. Global color table. The color table is stored as a consecutive list of RGB values. There is no accommodation for an alpha channel. Although it can be varied, typically the table has 256 entries, each containing a three bytes, one for each of the Red, Green, and Blue components of the color.

5. Image descriptor. This begins with a byte containing the identifying code 0x2C. This is followed by eight bytes giving coordinates of upper left hand and lower right hand corners of image in the display window (in *byte reversed* order), and typically a single byte containing 0 (although there may be much more graphical control information).

6. Image data, in LZW encoded form. When decoded, this gives the color table index for each pixel in the image. Indices are stored in scanline order starting at the upper lefthand corner of the image.

7. Trailer byte. This single byte contains the code 0x3b, and indicates the end of the GIF file.

As an example of the GIF format, Figure 5.11 contains portions of an annotated hexadecimal dump of a GIF file encoding of the red cube image,

```
Byte      Values                                     Decoding

          Header
000000    4749 4638 3761                             GIF87a

          Logical Screen Descriptor
000006    2c01 c800                                  byte reversed w x h
00000a    f700                                       CMap: 8-bits x 256
00000c    00                                         background index

          Global Color Table
00000d      7a 0000
000010    7a01 017b 0000 7b02 027b 0303 7b04 047b    reds and red-greys
000020    0606 7b09 097b 0d0d 7b11 117c 0000 7c01
000030    017c 0606 7c09 097c 0d0d 7c15 157c 1c1c
000040    7c1d 1d7d 0000 7d01 017d 0202 7d04 047d
          ... etc. etc. ...
000100    7f7a 7a7f 7b7b 7f7c 7c7f 7d7d 7f7e 7e

00010f      7f 7f7f                                  middle grey

000112    8000 0080 0202 8004 0480 0909 8022         reds and red-greys
000120    2280 5050 8051 5180 5b5b 8062 6280 6f6f
          ... etc. etc. ...
000300    0000 0000 0000 0000 0000 0000 00

          Image Descriptor
00030d      2c
00030e    0000 0000 2c01 c800 00                     (0, 0), (300, 200), 0

          Table Based Image Data
000317      08                                       LZW min code size = 8
000318    fe                                         254
000319      00 ad08 1c48 b0a0                        packed data
000320    c183 0813 2a5c c8b0 a1c3 8710 234a 9c48
000330    b1a2 c58b 1833 6adc c8b1 a3c7 8f20 438a
000340    1c49 b2a4 c993 2853 aa5c c9b2 a5cb 9730
000350    63ca 9c49 b3a6 cd9b 3873 eadc c9b3 a7cf
          ... etc. etc. ...
001060    bcda abbe faab c01a acc2 6aa6 0101 00

          Trailer
00106f      3b
```

Figure 5.11: Hexadecimal Dump of cube.gif, with Explanatory Comments

`cube.gif` of Figure 2.3. This is the same image that we saw in the PPM assignment.

There are a lot of other details and features supported by GIF, that we will leave for the interested student to study on his own. For that purpose, you can pick up the file `gif89a.doc` in
`/usr/local/misc/courses/viza654/documentation/`
if you want more information on GIF. In the same directory you will find `lzw.and.gif.doc`, by Steve Blakestock, which gives more detail on LZW encoding and the GIF implementation of it.

# Chapter 6

# The PostScript Page Description Language

PostScript[1] is the most widely known and used alternative to the raster-oriented pixmap-style image description technique. Although PostScript can support images stored as bitmaps or pixmaps, its main use is as a language for describing how a *page* should be drawn. In other words, PostScript "thinks" of an image as a sequence of steps – an *algorithm* – for drawing that image from a small set of drawing primitives.

PostScript became widely known and used when several laser-printer manufacturers agreed to build printers that could "understand" the PostScript language. Such printers look at a PostScript file to be printed as a program. The printer loads the program, and executes it – thus drawing out the image. Such printers are really small computers, with several megabytes of memory, that are attached to an electrostatic copy machine, a laser scanner, and an input port through which they can *download* PostScript files from the computer that is requesting the printing.

Our object in this course will be to achieve a basic level of understanding of the structure of PostScript. We will learn just enough so that we have a feel for this very important file format, and can look at a PostScript file and understand basically what it is all about. We will not attempt to become expert PostScript programmers, but will develop a little of our own PostScript code for a simple problem. Students should not look to these brief notes as anything but an outline of the highpoints of PostScript.

For a complete description of PostScript please see the pair of PostScript reference books in the laboratory. The *PostScript Language Reference Man-*

[1]PostScript is a trademark of Adobe Systems Incorporated.

*ual* is a red covered book, and gives a complete description of the details of PostScript. The blue covered *PostScript Language Tutorial and Cookbook* is a "how to" guide, that has lots of fancy examples of things that you can do with PostScript. Both books are by Adobe Systems Incorporated, and are published by Addison-Wesley, copyright 1985, 1986. In addition, a small group of example PostScript programs can be found in
`/usr/local/misc/courses/viza654/handouts/postscript/`.

Figure 6.1a contains a listing – in plain text form – of the PostScript file (i.e. the PostScript program) that will produce the *Hello World* target picture shown in Figure 6.1a. The file `target.ps` can be found in the handouts directory listed above. If you have the file in your local directory, then the Unix command

```
lpr target.ps
```

will cause the target picture to be sent to the laser printer and printed. The first line of the file (actually a PostScript comment) tells the printer that this is a PostScript file, and not simply a plain text file, causing the printer to read and interpret the file as a PostScript program, rather than just printing it as pages of text.

There are two handy tools on our system for examining PostScript files, to see what they will draw without having to go to the printer. These are the programs `xpsview` and `ghostview`. Simply type

`xpsview target.ps` or `ghostview target.ps`

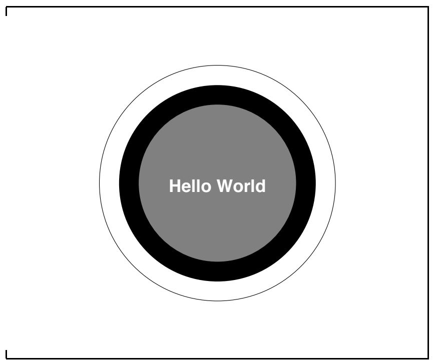
to display the image produced by the target program on the screen. Both programs come with a simple interactive interface that should be reasonably self explanatory. My personal preference is for `xpsview`, as I find that it is faster and gives a truer representation of what finally ends up on the printer. However, `xpsview` is not available on all machines, so you should know about both programs.

Sometimes, for debugging purposes, you may want to obtain a printout of the text of a PostScript file. You cannot do this by simply sending the file to the printer, as the printer will interpret the file as PostScript. To print the text of a PostScript file, use the program `psf`, which formats text files for printing, to change the ascii text of the file into a PostScript program that will "draw" the contents of your file. The output of `psf` can then be sent to the printer via `lpr`.

PostScript is not a particularly compact representation, so `.ps` files tend to be large. Fortunately, they are all text, so programs like the Unix `compress` utility are quite successful in reducing the amount of space needed to store them.

```
%!PS-Adobe-

/circle {newpath      % procedure to draw a circle
  0 360 arc
  closepath
} def

/inch {72 mul} def    % procedure to convert pts. to inches

/pg save def          % current state saved in pg

90 rotate             % page rotated clockwise to landscape mode
0 -8.5 inch translate % restore y-coordinate to bottom edge

% draw the target
5.5 inch 4.25 inch 3 inch circle stroke
5.5 inch 4.25 inch 2.5 inch circle fill
5.5 inch 4.25 inch 2.0 inch circle 0.5 setgray fill

% draw the text into the target
/Helvetica-Bold findfont 32 scalefont setfont
(Hello World) dup stringwidth pop
11 inch exch sub 2 div 8.5 inch 32 sub 2 div moveto
1 setgray show

showpage pg restore   % restore state after page drawn
```

a) Program



b) Resulting Page at 1/5 Scale

Figure 6.1: PostScript Program to Make *Hello World* Target Picture

## 6.1 Basic Notions of the PostScript Language

The basic underlying notion in PostScript is that of the page. A PostScript program works a page at a time, creating a blank *virtual* page, drawing into the page to produce an image, and finally outputting the page to an output device. Measurements within a page are done in *points* a measurement used by typographers – 1 pt.= 1/72 inch. By default, the origin of a page is its lower lefthand corner, as shown in Figure 6.2. The horizontal direction from this corner is the x coordinate, and the vertical direction is the y coordinate.

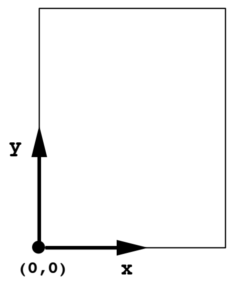


Figure 6.2: Origin of a Page

All drawing commands are done in a continuous coordinate system – not a discrete system like in a pixmap (i.e. there are no pixels). Because of this, PostScript images are continuously scalable to an arbitrary size without loss of detail.

PostScript maintains a current path – i.e. a sequence of points, curves, and lines – that can be added to by drawing commands. When a path is completed, it can be painted into the current page be either *stroking* the path (drawing its outline) or *filling* the inside of the path with a color.

PostScript also maintains a current transformation matrix (CTM), that acts to transform the original coordinate system of the page into a system more convenient for drawing a portion of the current path. Translate, rotate and scale operators allow one to manipulate the CTM.

## 6.2 Structure of PostScript Language

The basic data structure of PostScript is the stack. This is a first-in/last-out structure that works just like a stack of cards or books. New things are added to the top. Removals are also from the top. These basic operations on a stack are called *push* and *pop*, and are illustrated in Figure 6.3.

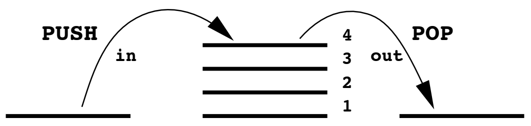


Figure 6.3: Operations on a Stack

PostScript maintains the following stacks:

- **operand stack** results of executing program objects,
- **systemdict** system dictionary stack of predefined PostScript operators,
- **userdict** user dictionary stack of user defined operators,
- **execution stack** call stack for procedure calls,
- **graphics state stack** top item is current context for drawing.

## 6.3 PostScript Language Syntax and Semantics

The following brief outline of PostScript Language syntax and semantics lists the valid objects in the language, and the action taken by the PostScript interpreter upon encountering the object.

- Numbers: examples of standard notations are `123, -123, -0.002, 1e14`. Radix notation follows the base 16 example `16#FF73`. **action** push numeric value onto stack.
- Strings: delimited by parentheses, as in the example `(this is a string)`. Parentheses can be inserted in a string if they are balanced like `(more (stuff))`, or can be *escaped* by preceding them by a \ character – i.e. `\(` and `\)`. Standard C special characters like `\n` can also be used. **action** push entire string onto stack as a single unit.
- Names: a sequence of characters containing no whitespace and without any of the syntax characters like `(`, `)`, `[`, `]`, \, /, for example `abc @nyc 2for1`. **action** the name should be bound to either a procedure or a value. The associated procedure is executed or the associated value is returned.

- Literal Names: any name immediately preceded by a `/` character, like `/abc`. **action** evaluates to the name itself, i.e. the name `abc` in the above example.
- Arrays: a sequence of objects delimited by `[` and `]`, like `[123 abc (xyz)]`. **action** The `[` symbol marks the start of an array, and the `]` symbol marks its end. The objects between `[` and `]` are evaluated one by one, then the entire result is pushed onto the stack as a single unit.
- Procedures: a sequence of objects delimited by { and }, such as `{5 10 moveto}`. **action** acts just like an array, only the objects between the delimiters { and } are left unevaluated.
- Comments: from `%` symbol to the end of the current line, example `% - this is a comment`. **action** comments are not evaluated and have no effect.

## 6.4 Execution of PostScript Programs

Execution of a PostScript program is stack oriented. As each expression is evaluated, its operands are popped from the operand stack, and its result is pushed back onto the operand stack. Figure 6.4 shows example execution of program text that adds the numbers 5 and 10 together. Following the example, we see that program execution is from left to right through the program text. The execution of a number from the program text places the number's numeric value on the operand stack. The execution of a procedure, such as `add` pops the procedure's operands from the top of the stack, performs the designated operation, and pushes the result back onto the stack. We see that `add` retrieves the top 2 values from the stack, adds them, and pushes their sum back onto the stack.

Because of the stack orientation of PostScript, arithmetic is naturally done in *postfix* form. What this means is that operators follow their operands, rather than being between their operands as in normal arithmetic (e.g. `5 10 add` rather than $5 + 10$). This takes a bit of getting used to, but is actually just as convenient as the usual *infix* scheme. One consequence of the scheme is that order of evaluation is strictly dependent upon position within a sequence, and thus correct evaluation of expressions does not require the use of parentheses.

The arithmetic operators that are built into PostScript are: `add sub mul div idiv mod`. They work as shown in Figure 6.5. As an example, the standard infix expression

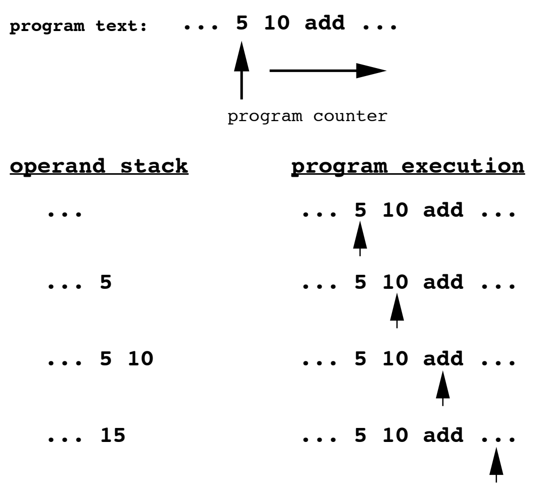


Figure 6.4: Example PostScript Program Execution Sequence

$$(3x + 2y)/4,$$

is equivalent to the PostScript postfix notation

`3 x mul 2 y mul add 4 div`.

| | | | |
|---|---|---|---|
| `a b add` | $\longrightarrow$ | $a + b$ | |
| `a b sub` | $\longrightarrow$ | $a - b$ | |
| `a b mul` | $\longrightarrow$ | $ab$ | |
| `a b div` | $\longrightarrow$ | $a/b$ | real result |
| `a b idiv` | $\longrightarrow$ | $a/b$ | integer result |
| `a b mod` | $\longrightarrow$ | $a$ mod $b$ | |

Figure 6.5: PostScript Arithmetic Operators

### 6.4.1 Procedure definition and execution

The operator `def` turns a name into a variable, by *binding* the name to either a value or a procedure. It takes the top 2 stack elements – the top element being the value to be bound or assigned to the literal name which is the second element on the stack. To define a procedure, the top element must be a procedure { ... }. `def` binds the name to the procedure, so that when the name is executed in a program, the procedure bond to it is evaluated. The following example defines a procedure with the name `sqr`, that computes the square of the current top stack element.

```
/sqr {dup mul} def       % squares the top stack element
```

An example use of the `sqr` procedure would be

```
4 sqr
```

which would leave the value 16 (4 squared) on the top of the stack after execution.

So, a procedure can be looked at as requiring a certain stack configuration when it is called, and that leaves the stack in a new configuration after its execution. Typically, a procedure will remove its arguments from the stack (*pop* them from the stack), compute one or more results and place the result value(s) on the stack (*push* them onto the stack). Thus, to fully document what a procedure does, one must specify 1) what operands it expects and in what order they must appear on the stack, 2) what values are returned to the stack by the procedure, and 3) what *side effects* (if any) the procedure has. Side effects are effects of running the procedure, other than changes to the operand stack. For example, some procedures affect other stacks, like the graphics stack, and others cause graphics to be drawn.

### 6.4.2 Control flow

Normally, PostScript programs are executed in sequence. Flow of control operators break the normal sequence of execution, and allow selective execution and looping. The most important control flow operators supported by PostScript are `if`, `ifelse`, and `for`.

Before describing these operators, it is necessary to show how logical expressions are built up. As in any programming language, there are operators to test relationships. These relational operators are shown in Figure 6.6.

The control flow operators are summarized in Figure 6.7. The `if` operator in PostScript pops a boolean expression and a procedure from the stack, and executes the procedure only if the boolean expression is true. Similarly, the `ifelse` operator pops a boolean expression and two procedures from the stack. It executes the top procedure if the boolean expression is false, and the second procedure if the boolean expression is true. The `for` operator provides a looping mechanism. It pops an initial counter value, a counter increment value, a counter limit, and a procedure from the stack. The procedure is executed once for each counter value, from the initial value up to the limit, with the counter incremented by the increment value after

| operators | | | | | | |
|---|---|---|---|---|---|---|
| operator | `gt` | `ge` | `lt` | `le` | `eq` | `ne` |
| relationship | $>$ | $\geq$ | $<$ | $\leq$ | $=$ | $\neq$ |

| relational operator examples | | |
|---|---|---|
| prior stack | operator | ending stack |
| `6 2` | `gt` | `true` |
| `6 2` | `lt` | `false` |

Note that `true` & `false` are defined symbols in PostScript

Figure 6.6: Relational Operators in PostScript

each execution. The current counter value is pushed onto the stack just before the procedure is executed. The procedure is responsible for removing this counter value from the stack. If the increment value is negative, the counter is decremented by this value on each iteration, and termination is when the counter is less than the limit. If the initial value exceeds the limit the procedure is never called.

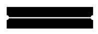
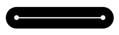
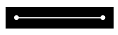
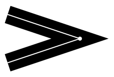
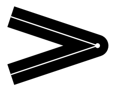
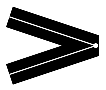
| control flow operator examples | | |
|---|---|---|
| prior stack | operator | ending stack |
| boolean-expr proc-true | `if` | results of proc |
| `6 2 gt {(big)}` | `if` | `(big)` |
| `6 2 lt {(small)}` | `if` | — |
| boolean-expr proc-true proc-false | `ifelse` | results of proc |
| `6 2 gt {(big)} {(small)}` | `ifelse` | `(big)` |
| `6 2 lt {(big)} {(small)}` | `ifelse` | `(small)` |
| initial incr limit procedure | `for` | results of proc |
| `0 0 2 10 add` | `for` | 30 |

Figure 6.7: Control Flow Operators in PostScript

## 6.5 Graphics in PostScript

### 6.5.1 Graphics State

The *graphics state* in PostScript is a description or list of parameters that control how drawing commands will be interpreted. This includes:

- CTM – current transformation matrix, determines the drawing coordinate system.
- point – where the virtual *pen* is that is "tracing out" the drawing
- current path – accumulation of lines and curves defining a boundary that will be drawn in by a `stroke` or `fill` command.
- clip path – defines a boundary, outside of which `stroke` and `fill` commands will have no effect.
- grey level (or color for color PostScript)
- line width – width in points of a stroke
- cap – 0  , 1  , 2 
- join – 0  , 1  , 2 
- miter limit – limit on length of sharp miters (when join = 0)
- dash – dash pattern for dashed lines
- flatness – measure of how accurately curved lines are rasterized when filled or stroked.

The graphics state can be saved, in its entirety, on the graphics state stack via the operator

`gsave`.

It can be restored from this stack via

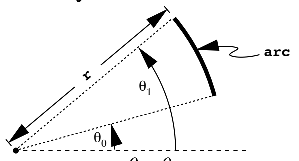
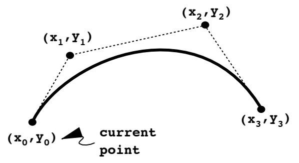
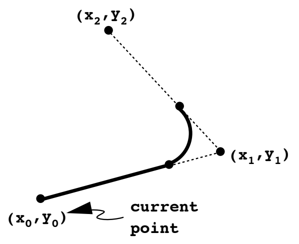
`grestore`.

`gsave` and `grestore` give one the ability to, for example, scale an individual object while it is being drawn without affecting the rest of the page. Simply save the graphics state on the stack, change the scale and draw the object, and then restore the graphics state from the stack before continuing.

Commands that affect the parameters of the current graphics state are

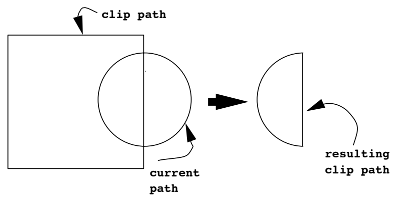
| | | |
|---|---|---|
| `n` | `setgray` | black $0 \leq$ `n` $\leq 1$ white |
| `n` | `setlinewidth` | `n` is line width in points |
| `n` | `setlinecap` | `n` $= 0, 1, 2$ |
| `n` | `setlinejoin` | `n` $= 0, 1, 2$ |
| `n` | `setmiterlimit` | |
| `pattern offset` | `setdash` | `pattern` = array of dashes and gaps, `offset` = offset from start of line |
| `n` | `setflat` | `n` $< 1$: very fine, `n` larger: cruder |

Drawing operators, are those that have side effects that alter the current path. These are

| | | |
|---|---|---|
| `x y` | `moveto` | move current drawing point to (x, y) |
| `dx dy` | `rmoveto` | move current drawing point by increment (dx, dy) |
| `x y` | `lineto` | draw line from current point to (x, y) |
| `dx dy` | `rlineto` | same as lineto, but incremental movement by (dx, dy) |
| `x y r` $\theta_0$ $\theta_1$ | `arc` | counter clockwise arc |
|  | | Circular arc measured from the x axis, with center (x, y) and radius r |
| `x y r` $\theta_0$ $\theta_1$ | `narc` | clockwise arc |
| $x_1$ $y_1$ $x_2$ $y_2$ $x_3$ $y_3$ | `curveto` | Bezier curve |
|  (x$_2$,y$_2$)<br>(x$_1$,y$_1$)<br>(x$_3$,y$_3$)<br>(x$_0$,y$_0$)<br>current point | | Bezier control points are the current point and the 3 (x, y) argument pairs |
| $x_1$ $y_1$ $x_2$ $y_2$ `r` | `arcto` | curve tangent to 2 lines |
|  (x$_2$,y$_2$)<br>(x$_1$,y$_1$)<br>(x$_0$,y$_0$)<br>current point | | Determines the curve of radius r that is exactly tangent to the lines defined by the current point and ($x_1$, $y_1$), and by ($x_1$, $y_1$) and ($x_2$, $y_2$). A straight line is drawn from the current point to the first tangent point, and an arc is drawn between the two tangent points |
| | `newpath` | initialize the path to be empty |
| | `closepath` | connect start and end of current path |

There are three operators that affect or make use of the clip path. These are

| | |
|---|---|
| `initclip` | set clip path to device outer boundary |
| `clip` | clip path = intersection of clip and current paths |



The clip path is replaced by the intersection of the clip path and the current path. This means that all of the interior of the current drawing path that lies within the clip path becomes the new clip path.

| | |
|---|---|
| `clippath` | set current path to clip path |

Operators that cause the current path to be drawn into device's raster, or otherwise affect the output page are

| | |
|---|---|
| `erasepage` | that's what is does! |
| `fill` | fill inside current path with gray (or color) |
| `stroke` | outline current path using line style parameters |

The command that causes the current page to be sent to the output device (i.e. printed on the page for a laser printer) is

`showpage`

Commands that affect the current transformation matrix are

| | | |
|---|---|---|
| `dx dy` | `translate` | |
| `sx sy` | `scale` | |
| `angle` | `rotate` | angle degrees counterclockwise |

The way to think about these commands is that they operate on the coordinate axis, as shown in Figure 6.8. The effect of these operations on the image is to cause all commands that affect the current path to be executed in the transformed coordinate frame.

### 6.5.2 Working with Text

Characters in PostScript are actually drawn using the drawing commands described above. However, it is convenient to have a special set of commands to handle text characters. Characters are stored in fonts. You can think of a font as a dictionary that associates each drawable character (like the letter 'a') with a procedure for drawing that character. The names for the fonts that are typically found in all PostScript implementations are: `Times-Roman`, `Helvetica`, `Courier`, and `Symbol`. There

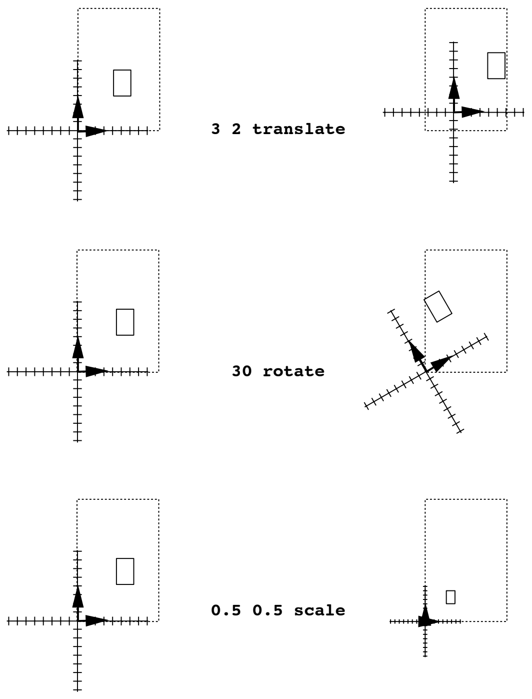


Figure 6.8: Translate, Rotate and Scale Transform the Coordinate Axes

are also bold, italic, and bold-italic versions: `Times-Bold`, `Times-Italic`, `Times-BoldItalic`, `Helvetica-Bold`, `Helvetica-Oblique`, `Courier-Bold`, `Courier-Oblique`. Of course, most implementations have many more fonts available.

The following operators operate on fonts and display text

| | | |
|---|---|---|
| `key` | `findfont` | looks up the font with the name `key` and pushes it onto stack. e.g. `/Times-Roman findfont` |
| `font scale` | `scalefont` | applies scale to font, returning the scaled font to the stack |
| `font` | `setfont` | makes font the current font, e.g. `/Courier findfont 14 scalefont setfont` makes Courier 14 the current font |

Once a font is set, the command

```
string show
```

will draw the string into the page starting at the current point. If character outlines are needed, then

```
string boolean charpath
```

will append the outlines of the characters in `string` to the current path. If the `boolean` operand is `false`, the outline will be suitable for stroking. If `true`, the outline will be suitable for filling or as a clip boundary.

# Chapter 7

# Compositing

Image compositing has been used for many years in traditional film, graphics and animation to achieve a variety of effects. Compositing can be as simple an operation as to lay down a "masked out" set of pixels from one image onto a background image. This is similar to the process shown in Figure 7.1 that is used in classical animation. Foreground characters are painted with opaque ink onto transparent *cels*, laid down over a background painting, and photographed to make a composite. This allows the animators to use a single background image for an entire animated scene. For most animation work, this idea is pushed quite far, with a typical frame being the composite of multiple cel layers. Other techniques, such as *blue screening* use the same kind of idea to composite film or video imagery together – again superimposing the image of an opaque foreground character over a background. Modifications of this idea are to make the foreground characters semi-transparent, blending foreground characters with background images, to create a variety of effects.

Compositing is particularly useful in computer graphics, and its ease of use in this medium is largely responsible for the explosion of interest in digital techniques in film special effects production. The technique of morphing is usually implemented by combining image warping with compositing to blend one image into another in such a way that it looks like the first image is "turning into" the second image. Other uses of compositing are to integrate animated imagery with recorded live action imagery, to produce effects that would be impossible to stage, such as characters with missing limbs, action in fantastic settings, and to integrate new footage with stock footage.

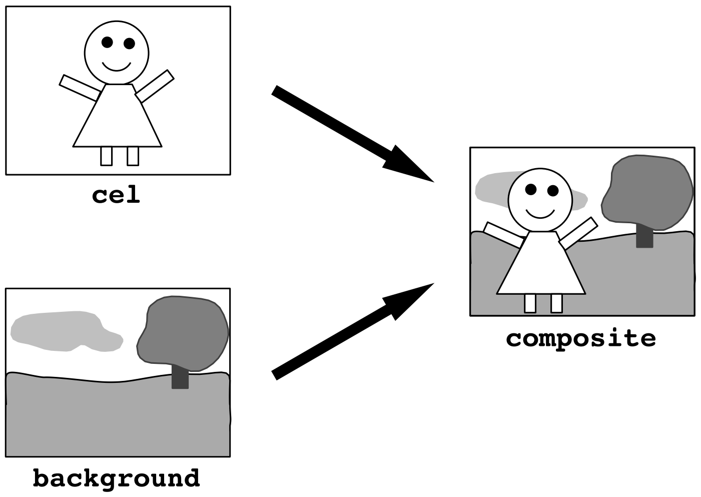


Figure 7.1: Compositing Foreground *Cel* with Background Image

## 7.1 Alpha

Recall that in our early discussion of framebuffers and image storage, we noted that besides storing red, green and blue primary values for each pixel, a full-color framebuffer, a pixel of which is diagrammed in Figure 7.2, will often also have eight bits per pixel available to store an *alpha* value. So far we have not dealt with this quantity other than to mention its presence. In principle, this alpha value could be used for any convenient purpose in an image manipulation algorithm, but its usual use is for compositing.

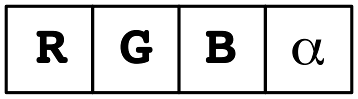


Figure 7.2: Four Component Pixel Value

When used for compositing, the *alpha* value of a pixel is interpreted as the pixel's *opacity*. On a scale from 0 to 1, an alpha value of 0 indicates that the pixel is fully transparent and a value of 1 indicates that it is fully opaque.

## 7.2 The Over Operator

The **over** operator, operating between two images, is the basic method for compositing a foreground image $F$ over a background image $G$ to produce a composite image $P$. The entire process would be written symbolically as

$$P = F\,\mathbf{over}\,G.$$

To define the **over** operator, let us let $C$ represent a color channel value (i.e. a red, green or blue value) for a specific single pixel in one of the images, and let a subscript on $C$, such as $C_F$, indicate which image the color channel value is from. Similarly, let $\alpha$ represent the alpha value for the same pixel, again with a subscript to indicate the image. Then, the per pixel operation implied by the **over** operator is given by

$$C_P = \alpha_F C_F + (1 - \alpha_F) C_G. \tag{7.1}$$

This is often rewritten for implementation as

$$C_P = \alpha_F (C_F - C_G) + C_G.$$

to save a multiply. Note, that all we are saying by this formulation is that the output color is computed by a linear interpolation between foreground and background pixel color channel values. The computation for each of the red, green and blue color channels is identical. If it is desired to composite multiple images over the same background image, this process can be repeated, compositing each foreground image into the background image, one at a time.

However, it is often desirable to composite multiple foreground images together and then composite the result with a background. In other words we would like the **over** operator to be associative, so that we can group compositing operations in any way and still get the same end result. More precisely, if we let $A$ and $B$ represent two different foreground images, and $G$ the background image, we would like the **over** operator to be defined such that it obeys the associative law

$$A\,\mathbf{over}\,(B\,\mathbf{over}\,G) = (A\,\mathbf{over}\,B)\,\mathbf{over}\,G.$$

The questions to be dealt with are 1) how do we reformulate the **over** operator to use the two alpha values in the foreground images to composite the color-primary values into the intermediate composited foreground image, and 2) how do we extend the **over** operator to combine the alpha values from the two foreground images to provide an alpha value for the intermediate image?

The problem is simplified somewhat by the fact that we know what the final result of the multiple compositing should be for the color-primary values. This is simply the result obtained by compositing each of the foreground images into the background image one at a time, using the process given by Equation 7.1. If we first composite image $B$ with the background, and then composite image $A$ with the result, by Equation 7.1 we would have

$$C_P = \alpha_A C_A + (1 - \alpha_A)[\alpha_B C_B + (1 - \alpha_B) C_G]. \tag{7.2}$$

Also, it is clear that the final composited image would have an alpha value of 1 for all pixels, since the compositing is being done into the fully opaque background image $G$. Thus, we have

$$\alpha_P = 1. \tag{7.3}$$

Now, let

$$H = A\,\mathbf{over}\,B. \tag{7.4}$$

stand for the the unknown intermediate foreground image that would be formed if we first composited the two original foreground images to form an intermediate image $H$. Then the final operation to composite the intermediate image $H$ with the background image $G$ is described by

$$C_P = \alpha_H C_H + (1 - \alpha_H) C_G, \tag{7.5}$$

Equation 7.2 can be rearranged to yield

$$C_P = \alpha_A C_A + (1 - \alpha_A)\alpha_B C_B + (1 - \alpha_A)(1 - \alpha_B) C_G. \tag{7.6}$$

Comparing Equations 7.5 and 7.6, we see that for associativity to hold we must have

$$\alpha_H C_H = \alpha_A C_A + (1 - \alpha_A)\alpha_B C_B, \tag{7.7}$$

and

$$(1 - \alpha_H) = (1 - \alpha_A)(1 - \alpha_B). \tag{7.8}$$

Equation 7.7 can be solved for $C_H$ to yield

$$C_H = \alpha_A/\alpha_H C_A + (1 - \alpha_A)\alpha_B/\alpha_H C_B. \tag{7.9}$$

and Equation 7.8 can be solved for $\alpha_H$ to yield

$$\alpha_H = \alpha_A + (1 - \alpha_A)\alpha_B. \tag{7.10}$$

## 7.3 Associated Colors

Now, Equations 7.9 and 7.10 give us methods for computing both the color-primary and alpha values for the intermediate image $H$. The problem with this formulation is that the two equations give us different calculations for opacity (alpha) and color value. It would be handy to have a unified scheme, so that color-primary and alpha values could all be calculated by the same method, meaning that we would not have to differentiate between the two kinds of quantities used in representing a pixel. The idea that will provide the desired unification is to use Equation 7.7, where each pixel color-primary value appears premultiplied by its associated alpha value.

The association of each color-primary with its alpha value can be done initially in the framebuffer by a simple multiplication. When an image is stored in this way we say it is an *associated color* image. Let $\mathcal{A}$ and $\mathcal{B}$ be the two associated-color foreground images corresponding to images $A$ and $B$, and let $\mathcal{H}$ be the intermediate associated-color image obtained by compositing $\mathcal{A}$ **over** $\mathcal{B}$. Then by Equation 7.7 our formulation for color primary value reduces to

$$C_{\mathcal{H}} = C_{\mathcal{A}} + (1 - \alpha_A) C_{\mathcal{B}}. \tag{7.11}$$

which would be computed in the same way as we computed alpha in Equation 7.10.

Equation 7.11 then is the equation governing our unified **over** operator and fully describes the computation needed to composite both color-primary and alpha values between pixels of associated-color images to produce an associated-color output image. One nice result of this formulation is that it is no longer necessary to differentiate between a foreground and a background image, except to note that a background image would be an image all of whose pixels have alpha values of one.

Two final things about associated-color images are reasonably obvious but worthy of note. The first is that when all alpha values are 1, the original image and its corresponding associated-color image are identical. The second is that before displaying an associated-color image, it is important to remember to divide each stored associated-color color-primary value by its associated alpha value to recover the original color. Otherwise, images will appear washed out.

## 7.4 Operations on Associated-Color Images

The desire to unify the color-primary and alpha value calculations was motivated by more than aesthetics. It turns out that the associated color image representation together with the **over** operator of Equation 7.11, gives us exactly the formulation that is needed to do other operations on images whose pixels have opacity information stored with them. For example, if we wish to blur a foreground image before compositing it with another image, we would want the blurred image to maintain a reasonable set of alpha values, i.e. we would want the alpha channel to be in some sense "blurred" along with the image color channels. If we did not do this, our resulting image would have clean hard edges where the original alpha mask was, although the rest of the image was blurred. It turns out that if we store our images using the associated-color representation, then operations like blurring, and other image manipulations like size and shape changes will work correctly if we apply the same operations to both the color channels and the alpha channel.

To verify this, let us look at a combined process of shrinking (minifying) two images and compositing. If we take two images and composite them first, and then follow this composition by the minification, it would be desirable that the result would be the same as if we first minified each image separately and then composited them. Let us say that our minification is simply a scale by one-half in the horizontal direction, so that the resulting image is one-half of the original width, but the height remains the same. To understand what color computations happen when we combine shrinking with compositing, we need only look at two adjacent pixels on a scanline in one image and the corresponding two adjacent pixels in the other image. The shrinking operation will be to replace each adjacent pair of pixels on a scanline in the original image with a single pixel whose color value is the average of the two original pixels. Call the two pixel associated-color color-primary values in the first image $\mathcal{P}$ and $\mathcal{Q}$, both with alpha value $\alpha_1$, and let the primaries in the second image be $\mathcal{R}$ and $\mathcal{S}$, both with alpha values $\alpha_2$. Now if we were to minify both pictures first, averaging the adjacent pixels into a single output pixel as shown in Figure 7.3a, then the resulting composite would have the value

$$(1/2\mathcal{P} + 1/2\mathcal{Q})\,\textbf{over}\,(1/2\mathcal{R} + 1/2\mathcal{S}),$$

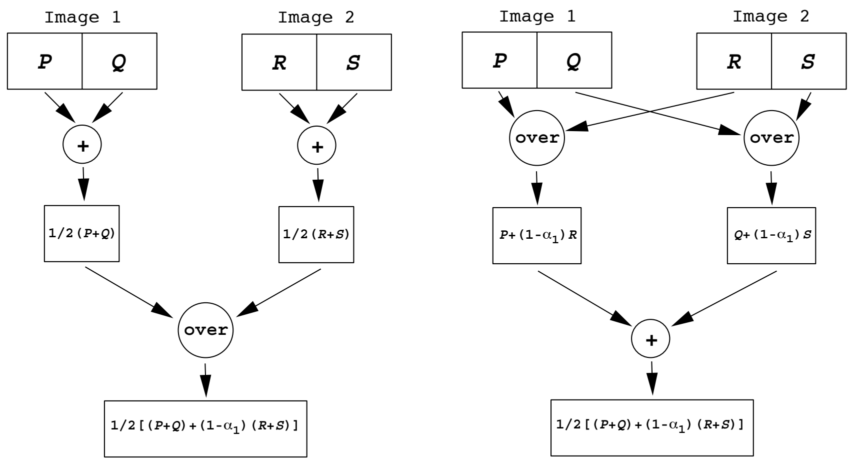
which yields

$$1/2[(\mathcal{P} + \mathcal{Q}) + (1 - \alpha_1)(\mathcal{R} + \mathcal{S})].$$

If we were to composite first and then minify, as shown in Figure 7.3b, we would have

$$1/2(\mathcal{P}\,\textbf{over}\,\mathcal{R}) + 1/2(\mathcal{Q}\,\textbf{over}\,\mathcal{S})$$

which reduces to the same form. A careful check will show that this same property holds over image magnification. In fact, a whole variety of image warping and filtering operations can be performed within a consistent context, given that the images are stored as associated color images. It is also easy to show that this is not true with normal color images. If the same process is attempted, changing the order of operations will result in different colors in the final image.



a) minification first b) compositing first

Figure 7.3: Combining Minification and Compositing

## 7.5 Other Compositing Operations

Once the basic notion of image storage in associated color form has been established, it becomes easy to define additional image combination operators. These are shown in Table 7.1, and are the standard image combination operators.

Table 7.1: Associated Color Image Combination Operations

| op | | | per pixel operation |
|---|---|---|---|
| $\mathcal{A}$ | **over** | $\mathcal{B}$ | $C_\mathcal{A} + (1 - \alpha_\mathcal{A})C_\mathcal{B}$ |
| $\mathcal{A}$ | **in** | $\mathcal{B}$ | $\alpha_\mathcal{B} C_\mathcal{A}$ |
| $\mathcal{A}$ | **out** | $\mathcal{B}$ | $(1 - \alpha_\mathcal{B})C_\mathcal{A}$ |
| $\mathcal{A}$ | **atop** | $\mathcal{B}$ | $\alpha_\mathcal{B} C_\mathcal{A} + (1 - \alpha_\mathcal{A})C_\mathcal{B}$ |
| $\mathcal{A}$ | **xor** | $\mathcal{B}$ | $(1 - \alpha_\mathcal{B})C_\mathcal{A} + (1 - \alpha_\mathcal{A})C_\mathcal{B}$ |

# Chapter 8

# Filtering

There is a large body of literature on digital images that has come out of the engineering and scientific communities. This is largely motivated by the desire to more effectively transmit, store, enhance, and reconstruct visual information in a variety of applications. Most notable in this area is work done to enhance and analyze images from microscopy, astronomy, satellites and space probes. NASA missions during the 1970's depended heavily on the ability of digital image processing techniques to extract clean images from data transmitted over many millions of miles of space from very weak space vehicle transmitters. Much of what is now common practice in working with images came out of this work. In this course we will not attempt to exhaustively cover this broad field, but will focus on several well understood procedures that have especially widespread use in the manipulation of images for graphics, special effects, digital painting and photographic enhancement.

Digital image processing algorithms tend to fall into two broad categories, those of filtering and warping. Warping is simply any change to an image that operates on the image's geometric structure. Algorithms for warping will be covered in a later chapter. Here, we will focus on image filtering.

We define a filtering operation to be one in which we modify an image based on image color content, without any overt change in the underlying geometry of the image. The resulting image will have essentially the same size and shape as the original. Let $P_i$ be the single input pixel with index $i$, whose color is $C_i$, and let $\hat{P}_i$ be the corresponding output pixel, whose color is $\hat{C}_i$. We can think of an image filtering operation as one that associates with each pixel $i$ a neighborhood or set of pixels $N_i$ and determines a filtered output pixel color via a filter function $f$ such that

$$\hat{C}_i = f(N_i).$$

This process is diagrammed in Figure 8.1.

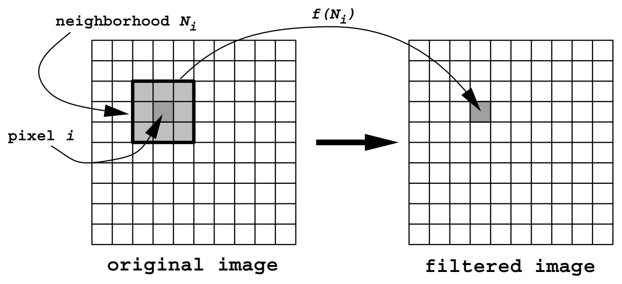


Figure 8.1: Diagram of the Filtering Process

## 8.1 Global Filters

A special class of filtering operations modifies an image's color values by taking the entire image as the neighborhood about each pixel. These can be thought of as *global filters* or simply recoloring algorithms. Of these techniques, the two most commonly encountered are *normalization* and *histogram equalization*.

### 8.1.1 Normalization

The goal of image normalization is to adjust the range of image colors to make sure that all colors fall within the range of colors allowed by the frame-buffer or file format being used, and at the same time assure that the full range of color intensities or values is used. A typical and simple normalization process would be to examine an image to determine the minimum and maximum values in the image, and then to rescale all of the colors using a linear map which places the minimum value at 0 and the maximum value at 1 (or 0 and 255 on an 8-bit integer scale).

We first find $C_{\text{min}}$ and $C_{\text{max}}$, the minimum and maximum image channel values, by scanning all of the pixel channel values in the entire image. Then for each channel $j$ of each pixel $i$ in the image we compute a normalized value

$$C_{ij} = \frac{C_{ij} - C_{\text{min}}}{C_{\text{max}} - C_{\text{min}}},$$

This process guarantees that the resulting image uses the full range of values available, and can be very helpful in the process of repairing a badly underexposed or overexposed image. Later we will see that many image processing algorithms do not guarantee that the resulting pixel colors will lie within the allowable range. Normalization will prove to be a very handy technique for readjusting the colors.

Figure 8.2 shows the effect of the normalization process. Figure 8.2a is a photograph of a natural scene that is exposed normally. Figure 8.2b shows the same scene but with the color values in the image artifically shifted toward the darks so that the maximum color value is only 1/2 of the full possible range. Finally, Figure 8.2c shows what this image would look like after the normalization process. Some artifacts are introduced due to the loss of color accuracy when the dark image was made, but otherwise the resulting image recovers most of the value information in the original image.

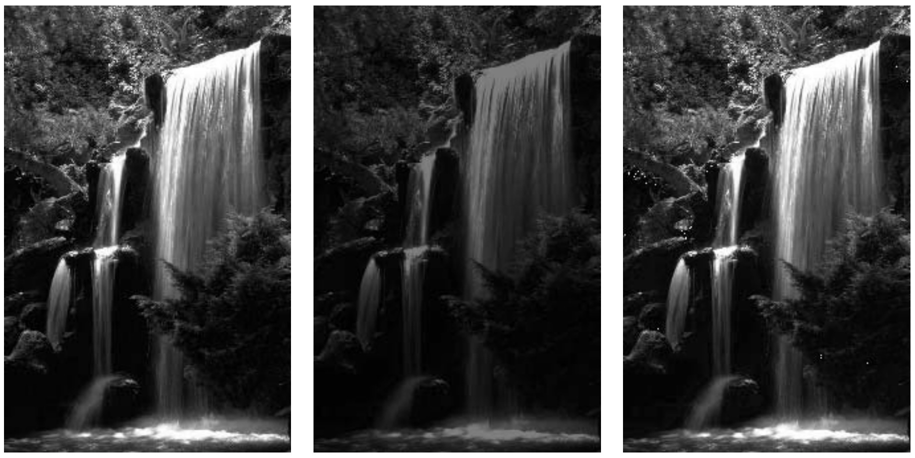
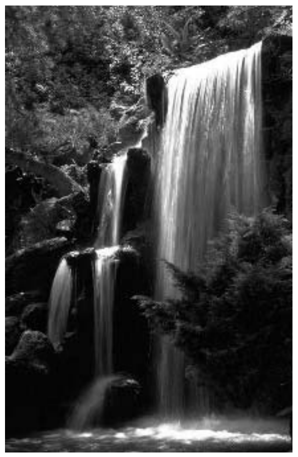

a) original image b) image at 1/2 brightness c) image renormalized

Figure 8.2: Image Normalization

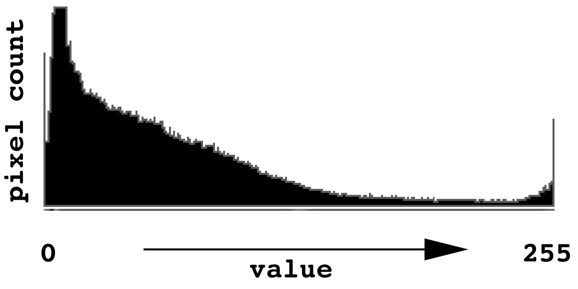
As an aside, normalization, as well as all of the image manipulation procedures discussed in these notes, should be done before any attempt at gamma correction of the image for the display. If gamma correction is done before normalization, for example, the linear normalization operation becomes highly nonlinear, failing to preserve either the correct relationships among colors in the image or the correct gamma correction.

### 8.1.2 Histogram equalization

An image histogram is simply a set of tabulations recording how many pixels in the image have particular attributes. Most commonly, such a histogram records pixel count for a set of discretized brightnesses or values. For example, examine the original natural scene image in Figure 8.3a, and its histogram in Figure 8.3b. In this case, the histogram has 256 *bins*, each corresponding with one of the integer values from 0 to 255. The height of the vertical line at each of these values indicates how many pixels in the image have a value that, when placed on a scale from 0 to 255 and rounded to the nearest integer, equals the value on the horizontal axis. Although the image of Figure 8.3a is normally exposed, the histogram shows that it is strongly dominated by darks, while still having a good number of pixels that are very bright. In fact, we see that all 256 possible values that can appear in the image are being used. A look at the image itself will verify the information shown in the histogram.



a) original image



b) histogram

Figure 8.3: Normally Exposed Image and its Histogram

The construction of the image histogram shown in Figure 8.3b is based on brightness values. We begin by initializing the 256 entries in integer array $H$ to 0. This array will be used to collect value counts for the histogram. Recall that given the red $R_i$, green $G_i$, and blue $B_i$ components of pixel $i$, its perceived value is given by

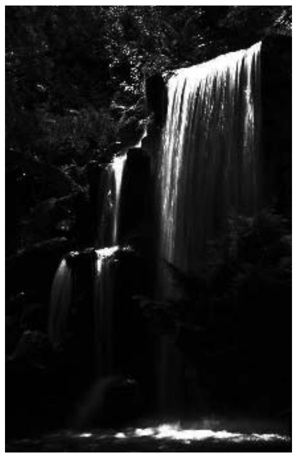
$$Y_i = 0.30R_i + 0.59G_i + 0.11B_i \tag{8.1}$$

Then for each pixel $i$ in the image, we compute its value $Y_i$ using Equation 8.1, scale to the range 0 to 255 and round to produce the discrete pixel value

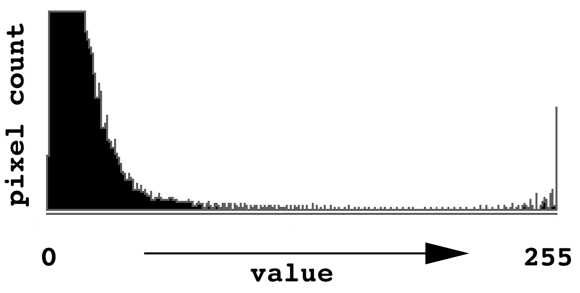
$$\bar{Y}_i = \text{round}(255 Y_i).$$

This is done across the image, keeping a count of the number of pixels having each of the 256 possible discrete values, by incrementing $H[\bar{Y}_i]$ by 1 for each discrete pixel value computed.

The goal of histogram equalization is to not only assure that the full range of values is used, but that the distribution of values in the image is as uniform as possible across the range of allowable values. To take a more extreme case, consider the image of Figure 8.4a. It was made from the image of Figure 8.3a by artificially enhancing the darks and suppressing the lights, leading to the histogram shown in Figure 8.4b. Clearly, this process has resulted in an image in which it is difficult to see detail, since so many of the values are compressed into a small range. However, since the image still uses the full range of values, a simple normalization would not improve the image. Histogram equalization will act to more uniformly spread the values in the image across the full range, brightening the lighter darks, while leaving bright pixels relatively unchanged.



a) darkened image



b) histogram

Figure 8.4: Darkened Image and its Histogram

Given an image and its histogram $H$, it is a simple matter to adjust the image colors so that they make a much more uniform use of the available pixel colors. Simply compute a new value $\hat{Y}_i$ for each pixel $i$ in the image using its discretized value $\bar{Y}_i$ and the histogram by the formula

$$\hat{Y}_i = 1/T \sum_{j=0}^{\bar{Y}_i} H[j], \tag{8.2}$$

where $T$ is the total number of pixels in the image. The original pixel red, green and blue color channel values would then be rescaled by the scale factor

$$S_i = \hat{Y}_i / Y_i. \tag{8.3}$$

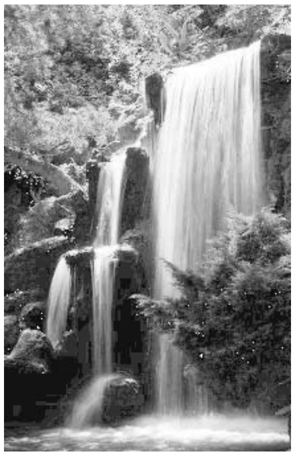
Equation 8.2 simply guarantees that each remapped pixel color will be set such that its value is proportional to the count of all pixels dimmer than or as bright as the pixel being remapped. In this way, pixel colors maintain the same relative brightness ordering with respect to each other, but will be shifted up or down the value axis to adjust the histogram. After remapping, it may be the case that by increasing a pixel's value, one or more of its channel values may exceed the maximim of 255. To correct this, the histogram equalization can be followed by normalization.

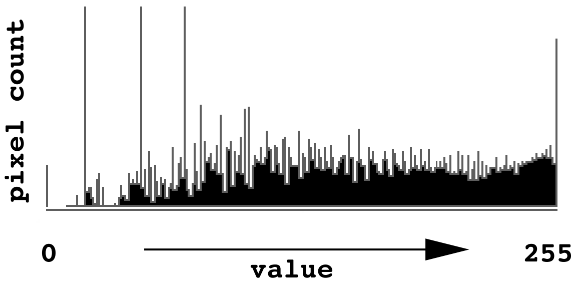
Figure 8.5a shows the dark image of Figure 8.4a after histogram equalization is performed. Comparing the histograms of Figure 8.5b and Figure 8.4b, it is clear how effective the algorithm is in its attempt to more evenly utilize the available color values. The histogram equalized image of Figure 8.5a is clearly not successful as an image from an artistic point of view, but it is very successful compared with Figure 8.4a in terms of our ability to see and correctly interpret the details in the image.

One final note on histogram equalization applies if you are using a framebuffer with a color lookup table. A histogram equalized image can be produced for display much more quickly simply by using Equations 8.2 and 8.3 to modify color lookup table entries rather than the image pixels themselves.

## 8.2 Local Filters

A local image filter, as opposed to a global filter, is simply any operation that modifies an image's color content by replacing each pixel's value with a new value obtained by examining only pixels in a local neighborhood about the pixel. Image content outside of the local neighborhood has no effect on the pixel's new value.


a) equalized image



b) histogram

Figure 8.5: Histogram Equalized Image and its Histogram

### 8.2.1 Construction of local neighborhoods

For all of the filters being discussed here, the neighborhood about a pixel will be taken to be a rectangular region centered on the pixel. The size of the rectangular neighborhood will be determined by the requirements of the filter function, but will be subject to the constraint that its width $w$ and height $h$ must be odd so that the neighborhood's center can lie on a pixel. Thus, in general

$$w = 2n + 1,$$

and

$$h = 2m + 1,$$

where $n$ and $m$ are positive integers. Figure 8.6 diagrams the relationships between $w$ and $n$, and $h$ and $m$.

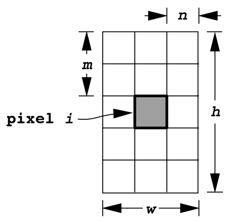


Figure 8.6: Rectangular Local Neighborhood About Pixel $i$

When using a rectangular neighborhood around a pixel for purposes of filtering, the question of what to do around the edges of the image invariably arises. For instance when the pixel under consideration is less than $n$ pixels from the left edge of the image, the neighborhood extends beyond the image's boundary, and it becomes unclear what pixel colors should be included. There are two common ways of addressing this dilema, by either shrinking the output image or expanding the input image.

Shrinking the output image is the simplest approach. Here we simply ignore the $n$ pixel columns on the left and right of the image and the $m$ scanlines on the top and bottom of the image. If we do this, we will compute no values for the corresponding pixels in the output image, producing an output image that is smaller than the input image by a few scanlines and columns.

If it is important that the output image be the same size as the input image expanding the input image is the best method. The idea is to think of the image as lying on a 2D plane of infinite extent, and conceptually tile all of this plane with pixels. We then need to consider what color values the pixels external to the image should have. The three most commonly used choices are 1) to give all of the external pixels a fixed value (typically black), 2) to tile the space external to the image with copies of the image, and 3) to tile the external space with alternating mirrored copies of the image. Setting the pixels outside of the image's boundary to a fixed value is clearly the easiest, but will give poor results for many filters or images, since there may be no reasonable correspondence between this fixed color and the image colors. The option of tiling space with the image can produce good results where the image is periodic in nature, with its right edge matching its left, and its top edge matching its bottom. For example, the tiling shown in Figure 8.7 appears to be reasonable in the horizontal direction, due to the repetitive nature of the image. However the tiling in the vertical direction clearly leads to discontinuities that may produce bad results with many filters. The option of tiling with mirrored copies of the image is shown in Figure 8.8. This procedure nearly always gives good results, as it tends to result in very clean seams between the real image and its replications. As a rule, this method is recommended.

## 8.2.2 Median filter

The *median filter* is a local filter that is typically used as a way to improve an image that suffers from *shot noise*. Shot noise is random dark or light spots, usually a single pixel in size, caused by dirt, electronic noise or image transmission errors.

The median filter works as follows. For each pixel in the image, choose

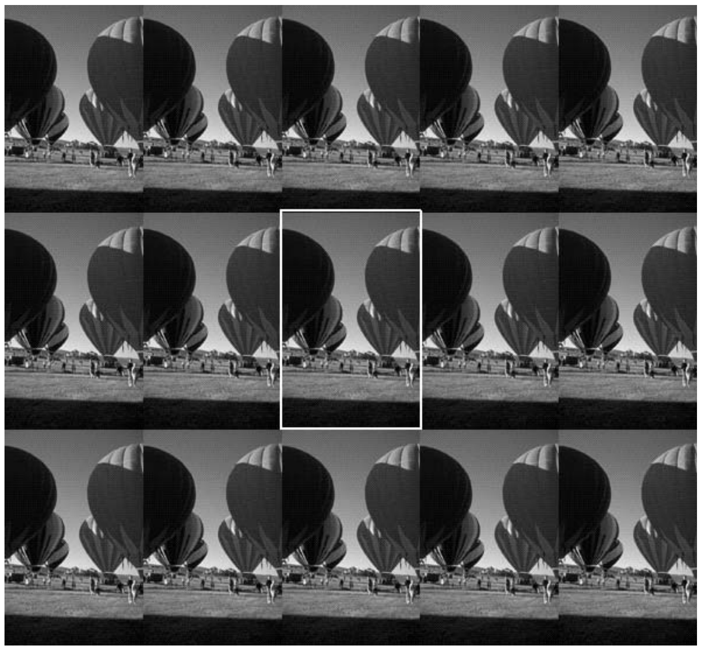


Figure 8.7: Image Tiling Space

a square neighborhood with the pixel being considered at its center. The output pixel value should be the median of the values in the neighborhood. The median is determined by taking each color value in the neighborhood, sorting them into an ordered list, and choosing the value that sorts to the middle position in the list.

The logic of the median filter stems from the observation that a small region of an image will usually contain pixel values that are closely related to each other. Further, one would expect that the pixel in the center of the region would contain a color value very close to the average of the colors surrounding it. Now, if we know that an image contains shot noise, and if the pixel at the center of the region is significantly different from the average, we can consider its value to be “suspect”. In this case, the best we can do is to replace its value by the average of the “non-suspect” pixels in the region. The median filter accomplishes this by sorting pixels with very high or low values to the two extreme ends of the sorted list of colors. However, the color in the center of the list, i.e. the median color, will be near to the mean of the uncorrupted pixels in the region.

Figure 8.9 shows the kind of results that can be obtained with a median

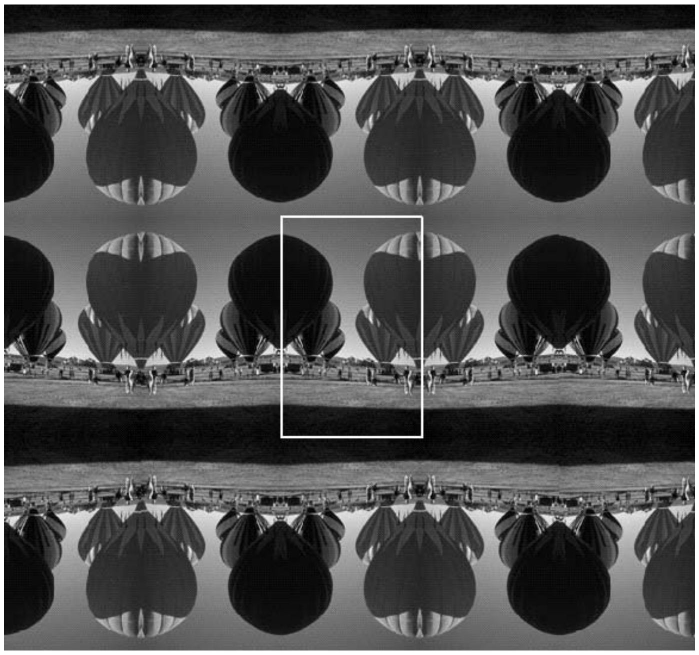
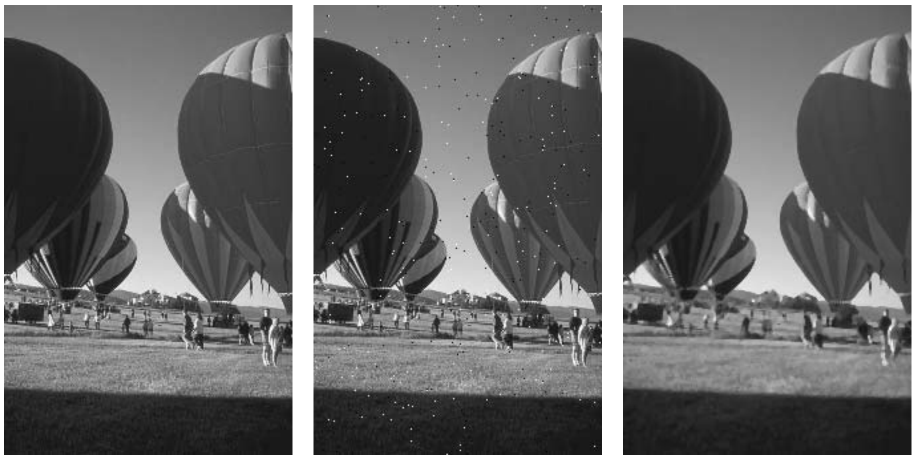


Figure 8.8: Mirrored Image Tiling Scheme

filter. Figure 8.9a is a clean original image, and Figure 8.9b is a version of this image degraded with shot noise. Finally, Figure 8.9c shows that image after applying a median filter employing a $3 \times 3$ square neighborhood. Note that virtually all of the noise is entirely removed, and the image suffers only minimal degradation.

### 8.2.3 Convolution

Probably the most important set of filtering operations is implemented by taking a weighted sum or difference of pixels within a neighborhood to produce each output pixel. This approach implements the mathematical concept of the *convolution* of two functions. The idea is easy to follow, as it can be thought of as sliding a rectangular array of weights over an image, replacing the pixel positioned at the center of the array by the appropriately weighted sum of the pixels under the array. The array of weights is commonly called the convolution *kernel*. Figure 8.10 shows a snapshot in time of the convolution process, with a kernel of weights centered over the pixel in the fourth scanline from the top and the fourth column from the



a) original image b) with shot noise c) after filtering

Figure 8.9: Median Filtering

left of the image. Pixel values are multiplied by kernel values, and the sum replaces the pixel in the fourth scanline and fourth column in the output image.

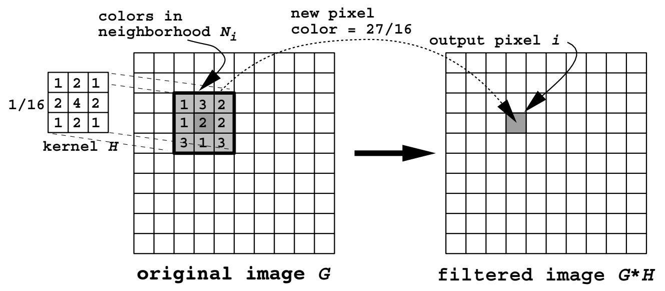


Figure 8.10: Convolution Filtering

Since the convolution process is a fixed process, a convolution filter is identified by its particular kernel. The simplest of the convolution filters is a uniform filter, or box filter. It is simply one in which all of the weights are the same, and that sum to 1. For each pixel $P_i$, it produces an output that is the numerical average of the pixels in its neighborhood.

Before proceeding, we should make our terminology and mathematical description more precise. In mathematical notation, it is usual to represent the filter kernel by the letter $H$. We will call our input image $G$ and the output image $\hat{G}$. For the time being, let us examine the weighting process in one dimension (i.e. along a scanline) since it will be easier to draw and describe. Later, the notation can be easily extended to two dimensions.

The discrete convolution process, involving "sliding" $H$ over $G$ to produce a weighted output, is indicated by the mathematical symbol $*$. Assume that both $G$ and $H$ are represented as arrays with indices beginning at 0 as in the `C` Programming Language. Then their discrete convolution would be written mathematically as

$$\hat{G}[i] = (G * H)[i] = \sum_{j=-\infty}^{\infty} G[j]H[i-j]. \tag{8.4}$$

There are several things to note about Equation 8.4. First, the summation is infinite. However, we may assume that the kernel width is the small odd integer $w = 2n + 1$, and we can assume that all kernel elements outside of the array bounds are 0. To simplify the notation it will be convenient to reindex the kernel $H$ from $-n$ to $n$ instead of from 0 to $w - 1$. The second is that the subscript on $H$ decrements from $\infty$ to $-\infty$ as the index of summation $j$ increases. One way to view this is that according to the precise mathematical definition of convolution, we mirror $H$ before multiplying in the summation. This is so that if we convolve a single point, the convolution will reproduce the point-spread function represented by the convolution kernel. Finally, we note that Equation 8.4 requires the extension of $G$ by exactly $n$ elements on either end to produce valid outputs for each of the $N$ elements in the original input $G$. Thus, if the input $G$ has width $N$, it must be expanded, for purposes of the convolution operation, to have $N + 2n$ elements, and indices from $-n$ to $N + n - 1$. Since we are not interested in the output of the convolution operation beyond the original extent of input $G$, after these modifications, Equation 8.4 can be rewritten

$$\hat{G}[i] = (G * H)[i] = \sum_{j=i-n}^{i+n} G[j]H[i-j],\ i \in [0, N-1]. \tag{8.5}$$

Figure 8.11 shows the operation implied by Equation 8.5 for a kernel of width $w = 3$ and an image scanline of width $N = 7$. The Figure clearly shows the weighting process, and emphasizes that since the array indexing for $H$ is reversed from the index of the summation, we have to think of the kernel as being reflected about its center before being superimposed over the image.

All of this can be extended to two dimensions in a straightforward way. Assume that the kernel $H$ is a rectangular array of width $w = 2n + 1$ and

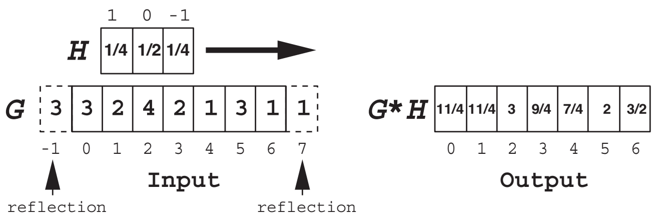


Figure 8.11: Convolution $G * H$

height $h = 2m + 1$, and that the image $G$ is of width $M$ and height $N$. Then we may write

$$\hat{G}[i,j] = (G{*}H)[i,j] = \sum_{l=i-m}^{i+m} \sum_{k=j-n}^{j+n} G[l,k]H[i{-}l, j{-}k],\ i \in [0, M{-}1],\ j \in [0, N{-}1]. \tag{8.6}$$

Equation 8.6 assumes that two-dimensional array indexing is *row major*, where the first index is the row (scanline) index and the second is the column index.

If, for some reason, it is desired to use a non-rectangular kernel, this can be modeled by making the kernel $H$ a rectangle, but filling its entries with 0's outside of the boundary of the desired kernel shape.

A final note on convolution kernels has to do with kernel construction and efficiency of processing. Many of the important convolution kernels have the separability property that they can be represented as the *outer product* of two one dimensional vectors. If we take the 1-dimensional convolution kernel $H_1$ and express it as the column vector $\mathbf{H}_1$, then the matrix

$$\mathbf{H}_2 = \mathbf{H}_1\mathbf{H}_1^t,$$

can be used as the 2-dimensional convolution kernel $H_2$. A example would be the 1-dimensional tent filter kernel

|  |  |  |
|---|---|---|
| 1 | 2 | 1 |

which can be used to construct the 2-dimensional tent filter kernel

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}.$$

Thus, if we can imagine a 1-dimensional filter and its properties, we can often easily create a 2-dimensional filter that will give similar results.

Along with ease of construction, the seperability property leads to an important efficiency consideration. This is that we can get the same result by filtering twice using the 1-dimensional filter as we would get by applying the 2-dimensional filter once. If the filter kernel is small, this has no obvious advantages but can have a real effiency payoff for large kernels. Consider a square filter kernel of width $w$. To filter an image using this kernel requires $w^2$ operations per pixel. If the kernel were seperable, then an alternative would be to process the scanlines of the image using the 1-dimensional factor of the kernel to produce an intermediate image, and then process the columns of the intermediate image using the same 1-dimensional kernel. Now, the 1-dimensional convolution process takes only $w$ operations per pixel and it is done twice, requiring a total of $2w$ operations. Using the seperability property results in a filtering algorithm whose time grows only linerally with the size of the kernel, whereas the direct 2-dimensional convolution takes time that grows with the square of $w$. There is clearly a big speed advantage as $w$ becomes large.

### 8.2.4 Important convolution filters

There are two main categories of convolution filter. Those that tend to smooth or blur the image are known as *low-pass* filters and those that tend to enhance edges, noise and detail are known as *high-pass* filters. In general, a low-pass filter is typified by a kernel dominated by positive weights, whereas a high-pass filter will have more of a balance between positive and negative weights. To gain an intuitive feeling for this terminology, think of an image of the surface of a pond covered with ripples of varying size. There are large smooth ripples covering large areas, and smaller ripples riding on top of the bigger ones. The large ripples provide the general contour to the water's surface and the little ones provide the detail to the surface. What a low-pass filter does is that it tends to remove detail by averaging over a local area, analogous to letting only the large smooth ripple through. What a high-pass filter does is that it tends to flatten out large ripples while enhancing the the small ones. They do this by taking differences over a local area, so that the output in a region of nearly identical values goes to zero, while the output in a region with rapidly varying detail will be enhanced.

An important class or family of low-pass or smoothing filters can be obtained by starting with a 1-dimensional *box* or *pulse* filter kernel of width three,

| | | |
|---|---|---|
| 1 | 1 | 1 |

Convolving with this box filter yields the simple average of the three pixels in a scanline neighborhood. Figure 8.12a shows this filter kernel graphed.

This can be extended to the 2-dimensional filter kernel

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

by the principle of seperability. It is easy to verify for yourself that if a scanline were convolved with this kernel, and then the resulting scanline were convolved with this kernel again, that the result would be the same as if the original scanline had been convolved with the kernel

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 1 |

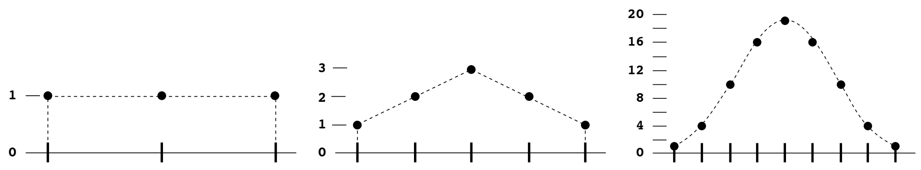
This is simply the convolution of the original kernel with itself. This kernel has the shape shown in Figure 8.12b, and is known as the *tent* filter. What it does is to apply linearly decreasing weights to pixels with distance from the center. Again, by the separability principle it can be used to form the 2-dimensional filter

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 1 |
| 2 | 4 | 6 | 4 | 2 |
| 3 | 6 | 9 | 6 | 3 |
| 2 | 4 | 6 | 4 | 2 |
| 1 | 2 | 3 | 2 | 1 |

The final filter kernel shown in Figure 8.12c is the result of convolving the box kernel with itself two more times. The kernel is bell shaped, and is given by

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 10 | 16 | 19 | 16 | 10 | 4 | 1 |

which can again be extended to 2-dimensions by separability. Further repetitions of convolution by the box kernel will tend more and more towards a *gaussian* curve.



a) box filter b) tent filter c) bell filter

Figure 8.12: Filters Formed From Successive Convolution of the Box Filter

The filters resulting from successive convolutions of the box filter tend to give better image smoothing results than a simple box filter of the same width. Figure 8.13 shows how these filters compare on an image containing a white square on a black background. Figure 8.14 makes the same comparisons on an image consisting of a series of vertical lines, and Figure 8.15 makes comparisons on a ripple image. In each of these figures image a) is the original image and image b), c) and d) are versions of image a) filtered by box filters of kernel width 3, 5 and 9 respectively. Image e) is filtered with a tent filter of width 5 and should be compared with image c). Image f) is filtered with a bell-shaped filter of width 9 and should be compared with image d).

There is a noticable improvement in image quality in the bottom row, with edges smoothly and cleanly blurred rather than blended together with the muddy appearance visible in the second row. This is especially apparent when comparing images d) and f) in Figures 8.14 and 8.15. Another thing to note is that a simple box filter has a strong directional bias in the horizontal and vertical directions that is largely corrected in the higher order filters. This is most obvious when comparing images d) and f) in Figure 8.15.

Figure 8.15 is an especially useful image in demonstrating why this family of filters are called low-pass filters. In the center, where the ripple size is large, there is very little difference between images a) and f). However, in the corners, where the ripples are small and closely spaced, there is a marked blur and decrease in intensity. The low-pass filter passes the low-frequency (high-wavelength) ripples relatively unchanged, but nearly eliminates the high-frequency (low-wavelength) ripples.

The final filter to be considered here is the high-pass filter with the kernel

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | -8 | 1 |
| 1 | 1 | 1 |

This filter acts to suppress large areas of uniform or smoothly varying color, while emphasizing edges, fine texture, and noise. Because the sum of the kernel weights is 0, an area of constant color will result in a filter output of 0. However, along edges or wherever the image has high detail the filter output will be a large positive or negative value. Figure 8.16 was made using this high-pass filter.

The images a), c) and e) are original unfiltered images and images b), d) and f) are the corresponding high-pass filtered images. Due to the high-pass filtering, only the edges in image a) pass through to image b). Image d) replaces each black/white or white/black boundary in image c) with a thin line, and replaces each black or white interior with black. Image f) is perhaps the most interesting, especially compared with the low-pass filtered image f) in Figure 8.15. Here the smooth low-frequency ripples in

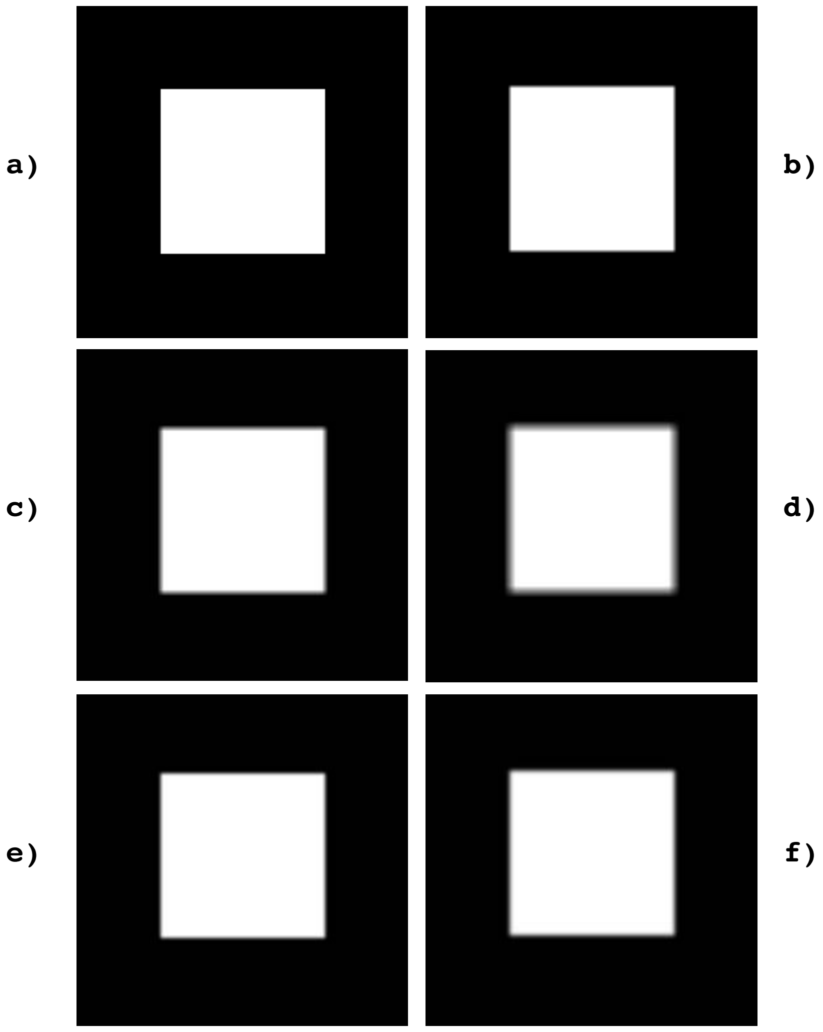


Figure 8.13: Square Image Low-Pass Filtered Versions

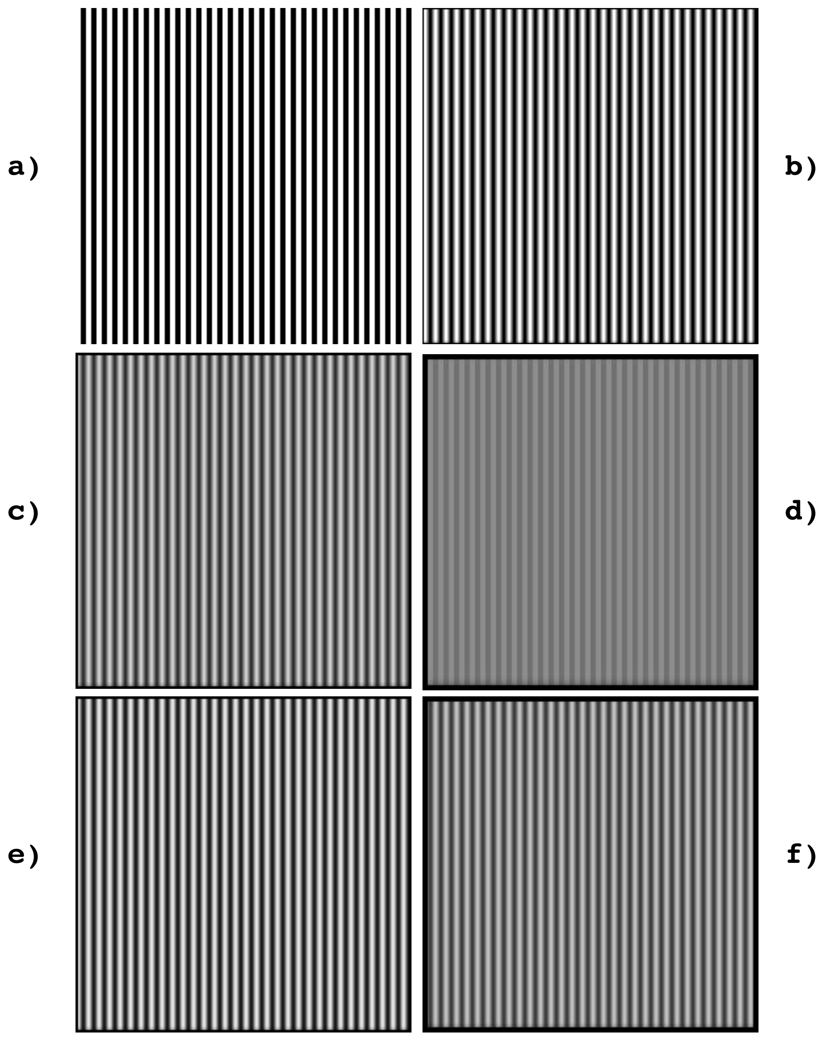


Figure 8.14: Vertical Line Image and Low-Pass Filtered Versions

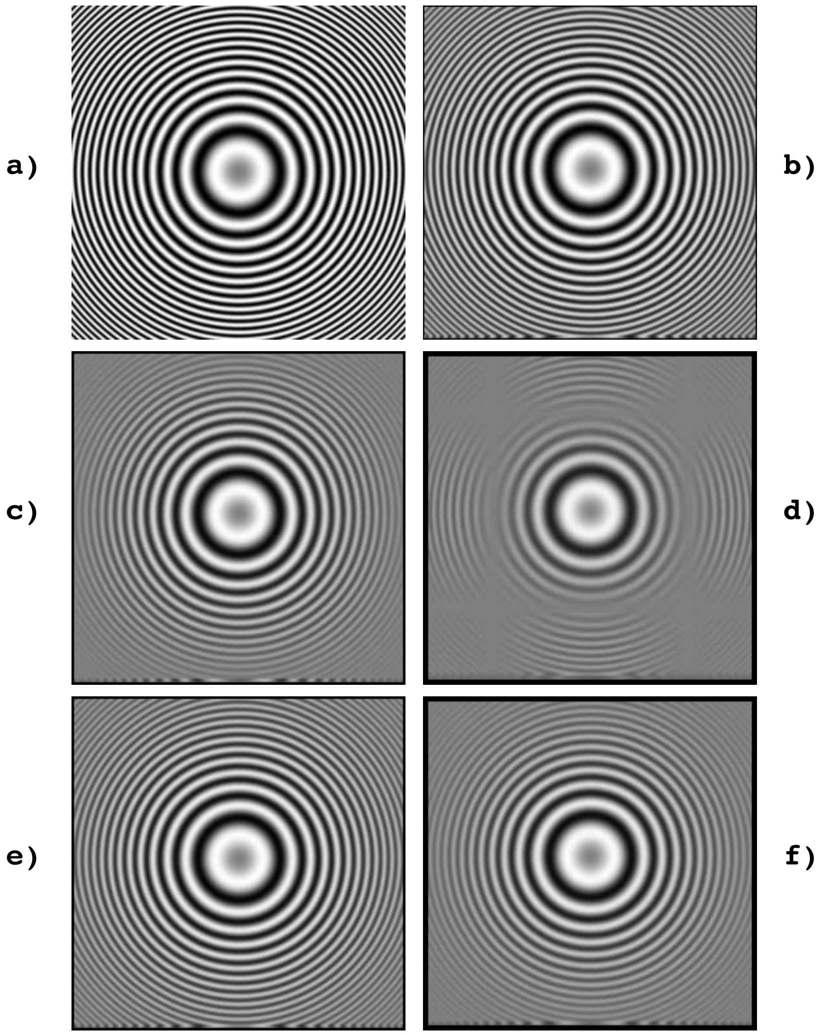


Figure 8.15: Ripple Image and Low-Pass Filtered Versions

the center of the image are suppressed, leaving only the effects of the fine high-frequency ripples in the corners.

Note that since the filter can produce negative as well as positive values, filter outputs must be adjusted to the range of the framebuffer. The filtered images in Figure 8.16 were made by scaling the absolute value of the filter output to fit the range of allowable values. Another approach would have been to compute the output image and then normalize the result.

There are many other possibilities for convolution filters. These include filters that do a combination of low and high-pass filtering, thus smoothing the image while emphasizing edges, filters that are directionally sensitive, emphasizing edges oriented in a particular direction, and filters that detect and emphasize particular patterns in the image. The convolution filtering homework assignment will give the student an opportunity to experiment with some of these filters on their own.

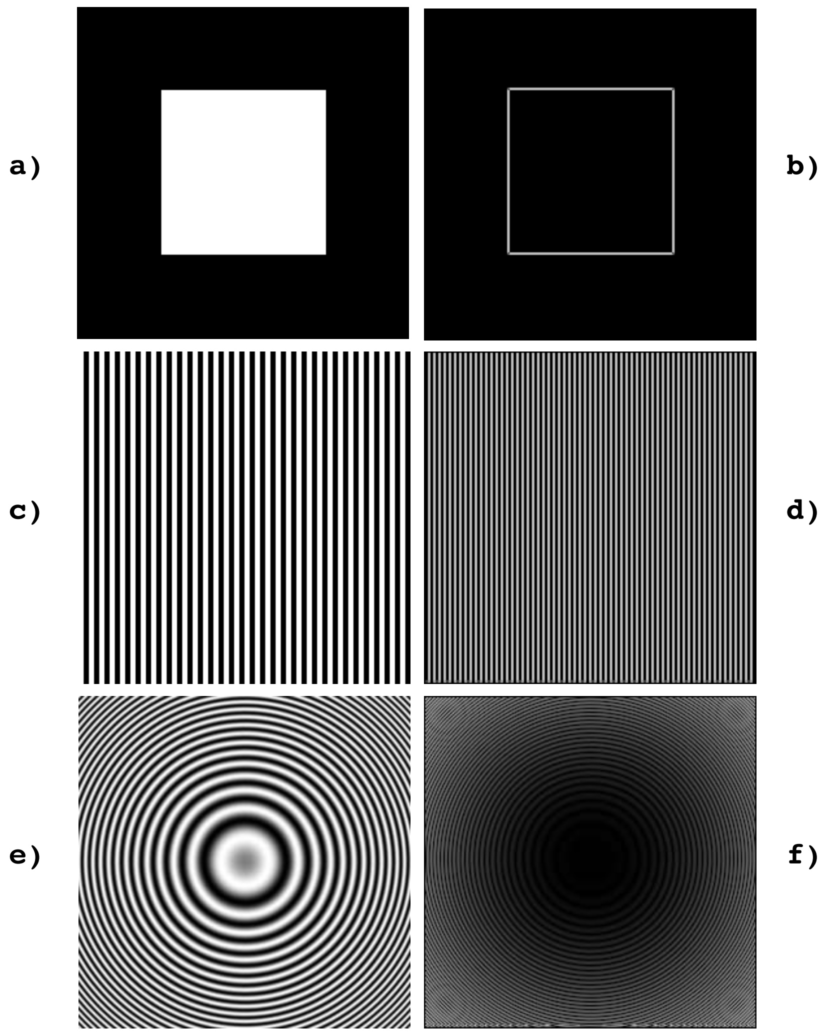


Figure 8.16: Original Images and High-Pass Filtered Images

# Chapter 9

# Image Warps

The word warp brings mental images of large shape distortions, but the technical term *image warp* is used to refer to any transformation on an image that changes any of its geometric properties. Thus a transformation that simply changes the size of an image and a transformation that makes complex twists and bends to an image are both referred to as warps. Other image operations, even ones that make drastic changes, are not referred to as warps as long as they only change non-geometric attributes such as color, texture, graininess, etc.

We often refer to the operation that does the warp as a map or function that sends an input image into a warped output image. To nail this idea down more precisely, let us start by measuring the input image in coordinates $(u, v)$, and our resulting warped image in coordinates $(x, y)$. Then any warp can be thought of as a mapping that sends each coordinate pair $(u, v)$ into a corresponding pair $(x, y)$. Any such map can be expressed as two functions, $X$ that determines the transformed $x$ coordinate from $(u, v)$, and $Y$ that determines the transformed $y$ coordinate from $(u, v)$. This concept is illustrated in Figure 9.1, and is written mathematically as

$$\begin{aligned} x &= X(u, v), \\ y &= Y(u, v), \end{aligned}$$

or in vector notation as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} X(u, v) \\ Y(u, v) \end{bmatrix}. \tag{9.1}$$

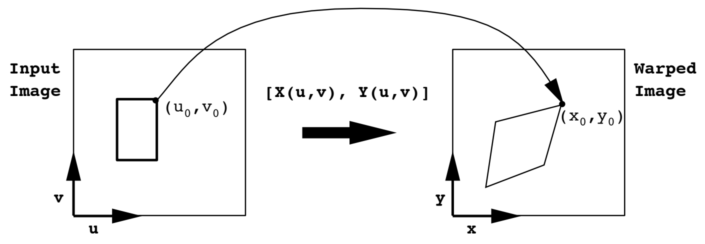


Figure 9.1: Image Map defined by functions $X(.,.)$ and $Y(.,.)$

## 9.1 Forward Map

The function pair $[X(), Y()]$ defines what is known as a *forward map* from the input image to output image. If the input image were continuous, and the forward map were reasonably smooth (i.e. it had no discontinuities), it would suffice to allow one to "paint in" the warped image from the input image. For each coordinate pair $(u, v)$ in the input image, we would simply color the point $[X(u, v), Y(u, v)]$ in the output image the same color as the point $(u, v)$ in the input image. However, if you reflect a bit you will see that this process is only of hypothetical interest, as it implies that we paint an infinitely finely represented image into another infinitely finely represented image using an arbitrarily large number of painting steps – *it works great but it takes forever!*

Realistically, digital images are not infinitely fine continuous representations but in fact consist of a finite number of discrete samples. We view an image by spreading the color of each sample over a small area – so that each sample corresponds with a pixel when we view the image. This seems to solve our problem, and allows us to write the simple algorithm

```
for(v = 0; v < in_height; v++)
  for(u = 0; u < in_width; u++)
    Out[round(X(u,v))][round(Y(u,v))] = In[u][v];
```

that will perform the forward map with only one operation per pixel in the input image. Note that the `round()` operations in the assignment statement are necessary to provide integer pixel coordinates. Unfortunately, the application of this simple forward map algorithm to a sampled image will result in the kinds of situations shown in Figure 9.2. The output image will be left with "holes" (i.e. pixels with unknown values) where the output is scaled up compared with the input, and multiple pixel overlaps where the output is scaled down with respect to the input.

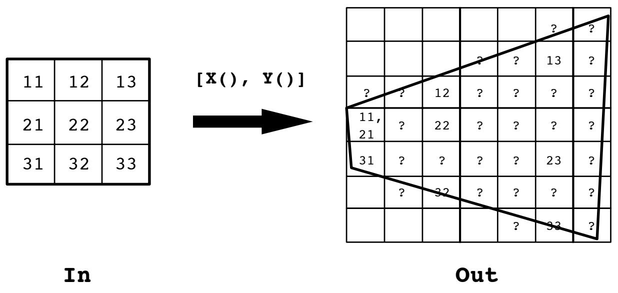


Figure 9.2: Forward Map Leaves Holes and Overlaps

The problem is that each pixel in the input image represents a finite (non-zero) area and actually projects an area of the input image onto the output, as shown in Figure 9.3. If our forward mapping algorithm carefully projected each pixel onto the output, we would see that each input pixel might fully cover some output pixels and partially cover several other output pixels, or it might only partially cover a single pixel. If we did this kind of projection, saving the percentage coverage of each output pixel by the input pixels, we could use these percentages to compute a weighted average color for each output pixel. This would give us an output image without holes or overlaps. But, the calculation of the correct percentages could get very complex, especially if the forward map is highly nonlinear (i.e. straight lines get mapped to curved lines). If the map is linear, a simple way to do the projection of a pixel is to project each of its four corners and then connect these projected corners by straight lines. However, if the mapping is non-linear, we would have to resort to other methods to get accurate results, such as mapping many sample points around the pixel contour. No matter what, this method is bound to be slow to compute and much harder to implement than our original simple pixel-to-pixel algorithm.

## 9.2 Inverse Map

The problems with the forward image mapping process can be solved by simply inverting the problem, turning it into an *inverse map*. Instead of sending each input pixel to an output pixel, we look at each output pixel and determine what input pixel(s) map to it, as shown in Figure 9.4. To do this we need to invert the mapping functions $X$ and $Y$. We will name these inverse mapping functions $U(x, y)$ and $V(x, y)$, and define them as

$$u = U(x, y),$$

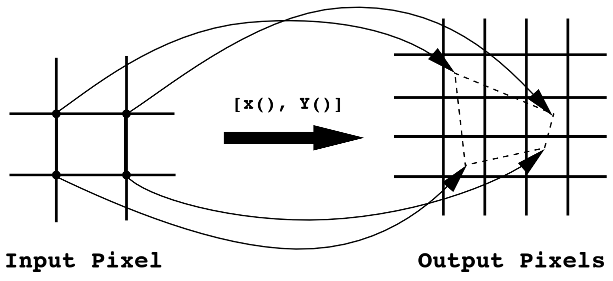


Figure 9.3: Projection of pixel area onto output raster

$$v = V(x, y),$$

or in vector notation as

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} U(x,y) \\ V(x,y) \end{bmatrix}, \tag{9.2}$$

with $U$ and $V$ chosen such that

$$\begin{aligned} u &= U[X(u,v), Y(u,v)], \\ v &= V[X(u,v), Y(u,v)]. \end{aligned} \tag{9.3}$$

In other words, the pair $[U(), V()]$ is chosen such that it exactly inverts or "undoes" the mapping due to the pair $[X(), Y()]$. Thus, if one samples the output image at point $(x, y)$, then $(u, v) = [U(x, y), V(x, y)]$ gives the coordinates $(u, v)$ in the input image that map to position $(x, y)$ in the output. As in the forward map, we have to round to give integer pixel coordinates, but here we round the coordinates into the input image, not the output image. Thus, even with a sampled digital image, we can compute our output image from the input image using the inverse map, without fear of holes or overlaps in the output image. The mapping algorithm is simply

```
for(y = 0; y < out_height; y++)
  for(x = 0; x < out_width; x++)
    Out[x][y] = In[round(U(x,y))][round(V(x,y))];
```

The inverse mapping method is simple and fast to compute, and thus is used extensively in most image warping tasks such as morphing and 3D texture mapping. It also has some nice features, such as the ability to sample through a mask to provide automatic clipping of the output image, as shown in Figure 9.5.

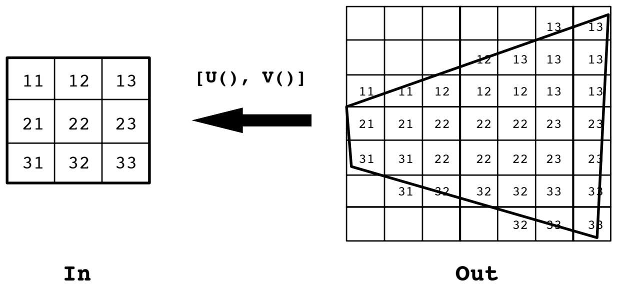


Figure 9.4: Inverse Map gives Complete Covering

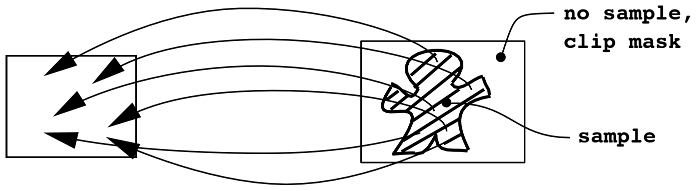


Figure 9.5: Inverse Mapping Through a Clip Mask

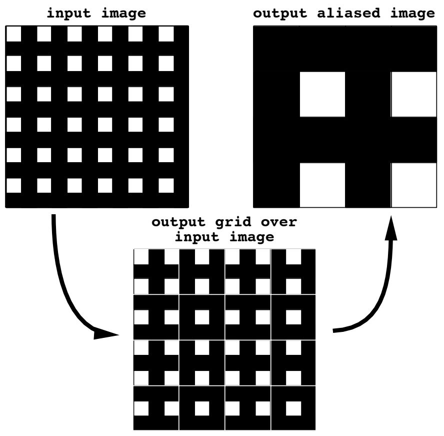
## 9.3 Warping Artifacts

However, nothing comes for free and in fact the inverse mapping process, while solving the serious problems of the forward map, still leaves us with artifacts that can degrade output image quality. These artifacts are due to the fact that in dealing with digital images we are dealing with sampled, not continuous, data. We will treat this issue in detail later, but for now it is enough to know that they fall into two important categories:

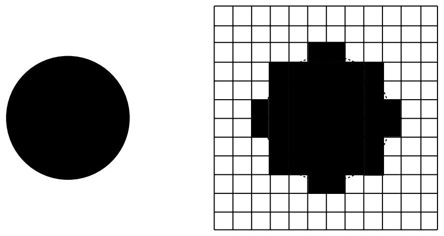
- **aliasing artifacts** caused by sampling the input too coarsely in some places, leading to missed features and unwanted patterning (Figure 9.6a) – aliasing will occur where a region of the output image is *minified* (scaled down) with respect to the corresponding region of the input image.

- **reconstruction artifacts** caused by using too crude a method (psf) to reconstruct the area around the input pixel, leading to staircasing or “jaggies” in the output (Figure 9.6b) – reconstruction artifacts will appear where a region of the output image is *magnified* (scaled up) with respect to the corresponding region of the input image.

We can minimize aliasing by doing a careful job of smoothing the input image before we sample it to produce the output image. We can minimize reconstruction artifacts by using a more sophisticated reconstruction scheme than the square flat areas that we get with pixels. However, these issues will require a careful treatment and we will not begin to build up the necessary tools until the next chapter.



a) aliasing artifacts



b) reconstruction artifacts

Figure 9.6: Aliasing and Reconstruction Artifacts

## 9.4 Affine Maps or Warps

An affine map on an image is of the general form

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}u + a_{12}v + a_{13} \\ a_{21}u + a_{22}v + a_{23} \end{bmatrix}.$$

In other words

$$X(u,v) = a_{11}u + a_{12}v + a_{13},$$
$$Y(u,v) = a_{21}u + a_{22}v + a_{23},$$

where the coefficients $a_{ij}$ are constants. Affine maps have several nice properties. They always have an inverse (except in some uninteresting degenerate cases), and they can be represented in matrix form. The map above can be written

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \tag{9.4}$$

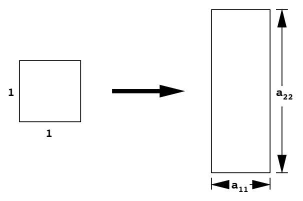
where a 1 is padded onto the end of each vector to allow for the translation constants $a_{13}$ and $a_{23}$. Multiplying the top row of the matrix by the vector, one can see that the effect of the constant $a_{13}$ is to translate the $x$ coordinate by an amount equal to the value of $a_{13}$, i.e.

$$x = a_{11}u + a_{12}v + a_{13} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}. \tag{9.5}$$

We have already seen some of the important affine transformations when we studied PostScript. These were *scale*, *translate* and *rotate*. An additional, highly useful, affine transformation is *shear*. In the matrix form of Equation 9.4 these transformations are:

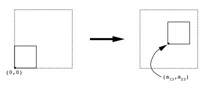
- **scale** $(x,y) = (a_{11}u, a_{22}v)$

$$\begin{bmatrix} a_{11}u \\ a_{22}v \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$



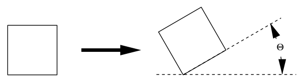
- **translation** $(x,y) = (u + a_{13}, v + a_{23})$

$$\begin{bmatrix} u + a_{13} \\ v + a_{23} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a_{13} \\ 0 & 1 & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$



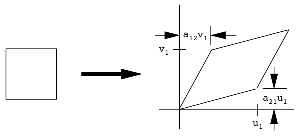
- **rotation** $(x, y) = (u \cos\theta - v \sin\theta, u \sin\theta + v \cos\theta)$

$$\begin{bmatrix} u\cos\theta - v\sin\theta \\ u\sin\theta + v\cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$



- **shear** $(x, y) = (u + a_{12}v, a_{21}u + v)$

$$\begin{bmatrix} u + a_{12}v \\ a_{21}u + v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & a_{12} & 0 \\ a_{21} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$



The structure of the scale, translation and shear transforms is reasonably obvious. The derivation of the rotation transform goes as follows. Refer to Figure 9.7 which shows the rotation of point $(x_0, y_0)$ through angle $\theta$ to make new point $(x_1, y_1)$. Since the rotation is about the origin, both points are at distance $r = \sqrt{x_0^2 + y_0^2} = \sqrt{x_1^2 + y_1^2}$ from the origin. The initial angle of rotation of point $(x_0, y_0)$ from the x axis is $\alpha$. Then from the definitions of the sine and cosine functions we have

$$x_0 = r\cos\alpha, \quad x_1 = r\cos(\alpha + \theta),$$
$$y_0 = r\sin\alpha, \quad y_1 = r\sin(\alpha + \theta).$$

The well known trigonometric identity

$$\cos(\alpha + \theta) = \cos\alpha\cos\theta - \sin\alpha\sin\theta,$$

may be multipled by $r$ to yield

$$r\cos(\alpha + \theta) = r\cos\alpha\cos\theta - r\sin\alpha\sin\theta,$$

so that by the definitions of $x_0$ and $x_1$ above we may obtain

$$x_1 = x_0\cos\theta - y_0\sin\theta.$$

Likewise, the identity

$$\sin(\alpha + \theta) = \sin\alpha\cos\theta + \cos\alpha\sin\theta.$$

may be multiplied by $r$ to yield

$$r\sin(\alpha + \theta) = r\sin\alpha\cos\theta + r\cos\alpha\sin\theta,$$

so that by the definitions of $y_0$ and $y_1$ above we may obtain

$$y_1 = y_0\cos\theta + x_0\sin\theta$$

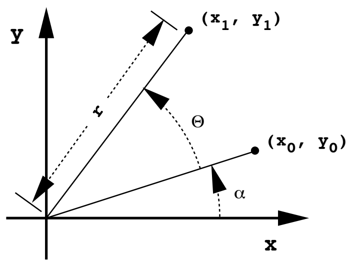


Figure 9.7: Rotation of a Point Around the Origin

## 9.5 Composing affine warps

In situations where one wants to do a series of affine transformations to one image, the matrix representation of the transform gives us an easy way to compose the transforms.

Let $S$ be a scale, $T$ a translation, $R$ a rotation, and $H$ a shear matrix. Let's do a rotation, followed by a scale, followed by a translation

1. $R\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix}$

2. $S\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \begin{bmatrix} u'' \\ v'' \\ 1 \end{bmatrix}$

3. $T\begin{bmatrix} u'' \\ v'' \\ 1 \end{bmatrix} = \begin{bmatrix} u''' \\ v''' \\ 1 \end{bmatrix}$.

This sequence of operations can be written in one equation as

$$T(S(R\begin{bmatrix} u \\ v \\ 1 \end{bmatrix})) = \begin{bmatrix} u''' \\ v''' \\ 1 \end{bmatrix}$$

but by the associative law, this gives the same result as

$$((TS)R)\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u''' \\ v''' \\ 1 \end{bmatrix}.$$

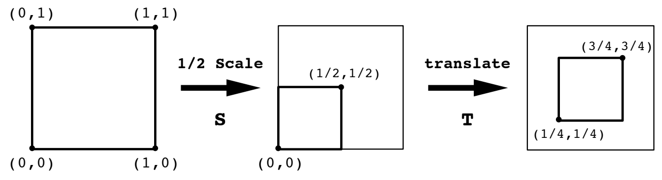
Thus, we see that we can premultiply the matrices to get a single composite transformation matrix

$$M = TSR,$$

that will do all 3 transformations (warps) in the specified order, in the single operation

$$M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u''' \\ v''' \\ 1 \end{bmatrix}.$$

This is a wonderfully compact and unified way of constructing a large variety of very useful warps in a very intuitive way – i.e. by simple composition of very easy to understand operations. Figure 9.8 shows an example of a composite operation that first scales an image by 1/2 and then translates it so that it is still centered in the original image rectangle.



$$S = \begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, T = \begin{bmatrix} 1 & 0 & 1/4 \\ 0 & 1 & 1/4 \\ 0 & 0 & 1 \end{bmatrix}$$

$$TS = \begin{bmatrix} 1 & 0 & 1/4 \\ 0 & 1 & 1/4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$M = TS = \begin{bmatrix} 1/2 & 0 & 1/4 \\ 0 & 1/2 & 1/4 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 9.8: Affine Composition – 1/2 Scale Followed by Centering Translation

## 9.6 Forward and Inverse Maps

A matrix $M$, which is a composite affine transformation built up as the product of a series of affine transformation matrices, is all we need to compute the forward map

$$M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Thus the forward map is implemented by simply multiplying the matrix $M$ by the pixel coordinates $(u, v)$ of each pixel in the input image to produce transformed pixel coordinates $(x, y)$ in the output image.

But, we already know that a forward map is not really what we want, as it will produce holes and overlaps in the output image. Fortunately, it is relatively easy to find the correct inverse map for any affine transformation expressed as a matrix $M$. All that we need to do is to find the matrix $M^{-1}$ with the property given by Equations 9.3 that

$$M^{-1}M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix},$$

i.e. that $M^{-1}$ exactly cancels the effect of $M$. Now when applied directly to output pixel coordinates $(x, y)$, $M^{-1}$ gives us the coordinates $(u, v)$ in the input image that map to $(x, y)$ under the forward map $M$,

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = M^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

If we can find such a matrix, the inverse map would be implemented by simply multiplying the matrix $M^{-1}$ by the pixel coordinates $(x, y)$ of each pixel in the output image to produce transformed pixel coordinates $(u, v)$ in the input image.

The matrix $M^{-1}$ is known (not surprisingly) as the inverse of the original matrix $M$. For a $3 \times 3$ matrix $M$, of the form

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix},$$

its inverse $M^{-1}$ is given by

$$M^{-1} = \frac{1}{a_{11}a_{22} - a_{12}a_{21}} \begin{bmatrix} a_{22} & -a_{12} & a_{12}a_{23} - a_{13}a_{22} \\ -a_{21} & a_{11} & -a_{11}a_{23} + a_{13}a_{21} \\ 0 & 0 & a_{11}a_{22} + a_{12}a_{21} \end{bmatrix}.$$

Thus, the terms of the inverse are easy to compute from the terms of the original affine transformation matrix, and it is a simple matter to write a `C` function that, given the affine transformation matrix $M$, returns its inverse $M^{-1}$.

In the next section we will begin to explore a class of nonaffine image warps that can nevertheless be expressed as matrices. We will again have the problem of determining the inverse matrix, but the compact formula above will not hold, since the bottom row of the forward matrix will not, in general, be simply $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$. In this case we will need to have a convenient algorithm for computing the inverse of such a matrix. The general formula for such an inverse is

$$M^{-1} = \frac{\mathcal{A}(M)}{|M|},$$

where $|M|$ is the *determinant* of the matrix $M$, and $\mathcal{A}(M)$ is the *adjoint* of $M$. Students with mathematical curiosity can check out a Linear Algebra book for details on how the adjoint and determinant are computed, or wait for the appendix to these notes which will be done in about a year!

## 9.7 Perspective Warps

Perspective warps are popular and useful warps that give the illusion that the image is receding off into space, as if it had been rotated out of the picture plane. Think of the image being painted onto a plane in three-dimensional space, and imagine a second plane, the plane of projection, which cuts through the image plane. A perspective warp can be thought of as a three-dimensional rotation of the image about the line of intersection between the two planes. It distorts the image's shape in such a way that groups of parallel lines that are also parallel to the line of intersection remain parallel to each other, but other groups of parallel lines recede to a common vanishing point. Figure 9.9 shows an example of a perspective warp in which it appears that the image has been rotated around its central horizontal axis. Note that all horizontal lines remain parallel to each other, but that the vertical lines (if extended upwards) would all intersect at a common point. Horizontal lines rotated to be "nearer" to the viewer appear longer than in the original image, and horizontal lines rotated to be "farther" from the viewer are shorter. This change in scale with perceived distance is known as foreshortening, and helps to create an illusion of depth in the perspective image.

Perspective warps extend the affine transforms by adding a non-linear scaling after the matrix multiplication, and make use of the two terms in the third row of the transform matrix that are 0 in the affine warps. They are obtained by setting the terms $a_{31}$ and/or $a_{32}$ to non-zero values. When

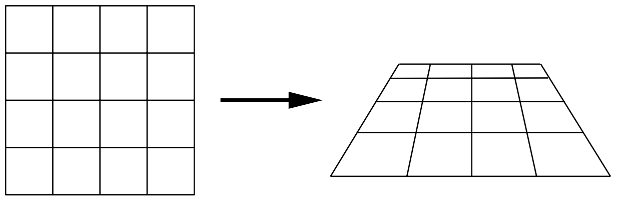


Figure 9.9: A Perspective Warp

this is done, the product of the matrix by pixel coordinates yields a result where the third coordinate of the resulting vector is not one. This can be seen in the example

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ a_{31}u + a_{32}v + 1 \end{bmatrix}.$$

It is common practice to call this third element the $w$-coordinate of the vector. For all of the affine warps, if the $w$-coordinate was one before the warp, it remained at one after the warp. For the perspective warps, $w$ is in general not one after the matrix multiplication is done. In fact, the $w$ coordinate is related to the distance of the point $(u, v)$ in the rotated picture plane from the original picture plane – i.e. it can be thought of as a non-linear measure of the distance of the pixel from the viewer.

Abstractly, when we originally formed the vector $\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$ from the two dimensional coordinate pair $(u, v)$, we were treating each two dimensional point as if it were a point in three-dimensional $(u, v, w)$ space, lying on the plane $w = 1$. This is shown pictorially in Figure 9.10, and is known as a homogeneous coordinate system (i.e. a system in which every point has an identical third coordinate).

Returning to the perspective transformation process, we noted that after the matrix multiplication, our homogeneous image coordinates no longer had a third coordinate of one. To complete a perspective warp, and restore our points to homogeneous coordinates with $w = 1$, we follow the matrix multiplication by a scaling operation. We divide each vector by its own $w$ coordinate, i.e.

$$\frac{1}{w}\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} u/w \\ v/w \\ 1 \end{bmatrix}.$$

This can be seen to be equivalent to connecting the three-dimensional point

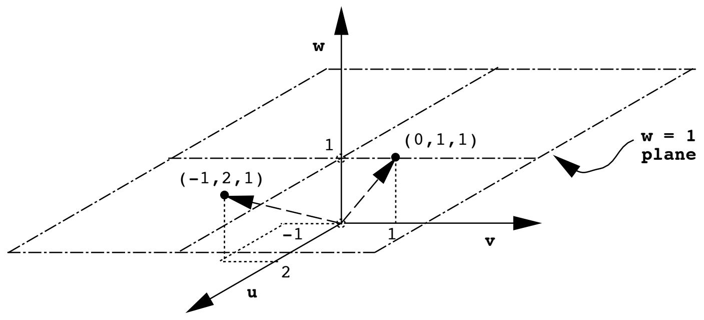


Figure 9.10: Homogeneous Coordinates Lie on the Plane $w = 1$

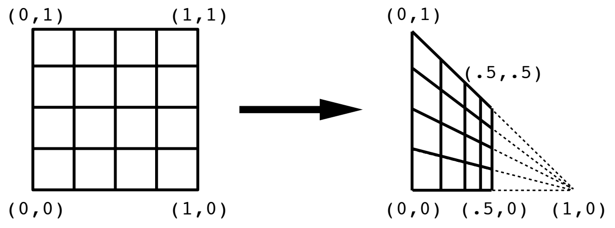
with the origin by a line, and determining the intersection of that line with the plane $w = 1$.

So, the process for computing the forward transformation for a perspective warp is to 1) multiply each pair of image coordinates by the perspective warp matrix, then 2) scale each coordinate triple by the reciprocal of its own $w$ coordinate. An example of this process is worked out in Figure 9.11, where the perspective warp matrix is

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

Note in this example, that no matter how large $u$ is, the projected $u$ coordinate cannot exceed 1. Similarly, for large $u$, the projected $v$ coordinate tends towards 0 – i.e. the vanishing point is at $(1, 0)$.

Unlike the affine warps, however, the perspective warp involves a nonlinear operation (dividing by $w$), and so it may be more difficult to find a suitable inverse function for the inverse mapping operation. Fortunately, it is possible to find the inverse to within a scale, and this will turn out to be all that we need. We shall see about this in the next Chapter.



$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$P \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \\ u+1 \end{bmatrix} \Longrightarrow \begin{bmatrix} \frac{u}{u+1} \\ \frac{v}{u+1} \\ 1 \end{bmatrix}$$

Figure 9.11: Example Perspective Warp

Chapter 10

# Inverse Projective Maps and Bilinear Maps

## 10.1 Projective Maps

A useful tool for doing warps would allow the user to change the shape of a rectangular image by moving corner points, as shown in Figure 10.1. This would allow us to make any arbitrary shape distortion that preserves linear edges. The problem is: what is the map from input to output?

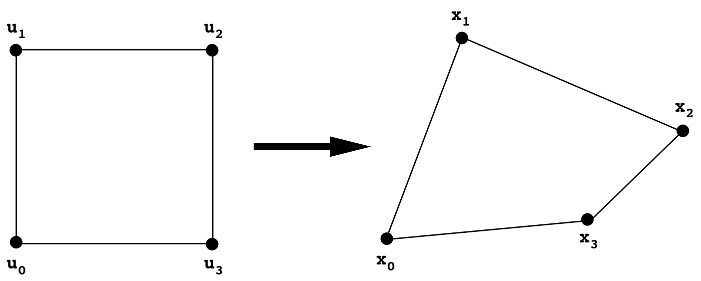


Figure 10.1: Interactive Image Warping by Moving Corners of Rectangle

The answer is that the map

$$M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} \Longrightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \tag{10.1}$$

with

$$x = u'/w', \;\; y = v'/w',$$

that we used to implement a perspective transform, will in fact warp a rectangle into an arbitrary convex quadrilateral. We call this entire class of operations the *projective transforms*.

The elements of the transformation matrix $M$ for a general quadrilateral warp are not hard to compute. It turns out that the lower righthand corner term $a_{33}$ of $M$ simply acts to provide a uniform scaling. Since this is redundant with the scaling effects of the other diagonal terms $a_{11}$ and $a_{22}$, we can arbitrarily set $a_{33}$ to one without any loss of generality. Having done this, each application of Equation 10.1 to a vertex $\overline{u_i}$ of the starting rectangle gives two equations

$$x_i = \frac{a_{11}u_i + a_{12}v_i + a_{13}}{a_{31}u_i + a_{32}v_i + 1},$$

$$y_i = \frac{a_{21}u_i + a_{22}v_i + a_{23}}{a_{31}u_i + a_{32}v_i + 1},$$

for vertex $\overline{x_i}$ of the corresponding vertex of the warped quadralateral. Since we have four corresponding coordinate pairs $(u_i, v_i)$, $(x_i, y_i)$ this gives us the eight necessary correspondences to solve for the eight unknown coefficients $a_{ij}$ of the transformation matrix $M$.

## 10.2 Inverse Projective Maps

For purposes of image warping we have defined a projective map to be any map that can be defined by a 3×3 homogeneous transformation matrix. The map itself consists of the following steps for transforming a two-dimensional coordinate pair $(u, v)$:

1. extend $(u, v)$ to a three-dimensional homogeneous vector

$$(u, v) \longrightarrow \begin{bmatrix} u \\ v \end{bmatrix} \Longrightarrow \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

2. multiply by the transformation matrix to get the transformed homogeneous vector

$$M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix}$$

3. scale the result by $1/w'$ (project onto the plane $w = 1$) and discard the $w$ coordinate, giving a transformed 2D coordinate pair $(x, y)$

$$1/w' \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} u'/w' \\ v'/w' \\ 1 \end{bmatrix} \Longrightarrow \begin{bmatrix} u'/w' \\ v'/w' \end{bmatrix} \longrightarrow (x, y).$$

Now, this whole process is invertible, to within a scale applied to the matrix $M$. If we compute $M^{-1}$ (or any scale of $M^{-1}$), then if we invert the process for the forward map, we get the inverse map:

1. extend $(x, y)$ to three-dimensional homogeneous form

$$(x, y) \longrightarrow \begin{bmatrix} x \\ y \end{bmatrix} \Longrightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2. multiply by $M^{-1}$

$$M^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix}$$

3. scale by $1/w''$ and discard the $w$ coordinate

$$1/w'' \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} x''/w'' \\ y''/w'' \\ 1 \end{bmatrix} \Longrightarrow \begin{bmatrix} x''/w'' \\ y''/w'' \end{bmatrix} \longrightarrow (u, v)$$

Note, in the above process, that any arbitrary scale factor is "washed out" when we scale by $1/w''$ in the last step. Thus, although the general projective mapping process contains a non-linear operation (divide by $w$), it is still possible to use a simple matrix inverse to compute the inverse map. This is of great importance to image warping algorithms, since it means that for any projective map we can always find an inverse map, making it a simple matter to use an inverse mapping process to compute the warped image.

Figure 10.2 gives the computation of the inverse of a general $3 \times 3$ projective transformation matrix. It is taken directly from the Wolberg text on pages 52-53. One important note is that since we just need $M^{-1}$ to within a scale, we can skip the divide by the determinant $|M|$, and simply use the adjoint $\mathcal{A}(M)$ in place of the inverse. This gives a nice time savings in the original calculation of the inverse.

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$|M| = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{32}a_{21} - a_{13}a_{22}a_{31} - a_{12}a_{21}a_{33} - a_{11}a_{32}a_{23}$$

$$\mathcal{A}(M) = \begin{bmatrix} a_{22}a_{33} - a_{23}a_{32} & a_{13}a_{32} - a_{12}a_{33} & a_{12}a_{23} - a_{13}a_{22} \\ a_{23}a_{31} - a_{21}a_{33} & a_{11}a_{33} - a_{13}a_{31} & a_{13}a_{21} - a_{11}a_{23} \\ a_{21}a_{32} - a_{22}a_{31} & a_{12}a_{31} - a_{11}a_{32} & a_{11}a_{22} - a_{12}a_{21} \end{bmatrix}$$

$$M^{-1} = \frac{\mathcal{A}(M)}{|M|}$$

Figure 10.2: $3 \times 3$ Matrix Inverse

## 10.3 Bilinear Interpolation

We have already seen that a projective warp can map a rectangular image to any quadrilateral shape. If the warp maps a rectangle into a parallelogram, then the projection is a simple affine projection with elements $a_{31}$ and $a_{32}$ of matrix $M$ both set to zero. However, if these terms are non-zero, we have a perspective warp, and the resulting shape is a quadrilateral whose opposite sides are not necessarily parallel. This is fine if we want a true perspective of the original image, but there are times when we would like to do a warp which preserves uniform spacing between image features. A perspective warp will have a foreshortening effect, making image features shrink as they “recede” away from the viewer. This effect can be seen in Figure 10.3.

Often this foreshortening effect is not desired, and a mapping like that shown in Figure 10.4 would be preferred. This kind of map is known as a bilinear warp. Just like a perspective warp, it will allow for arbitrary mapping from any quadrilateral shape to any other quadrilateral shape, but will preserve spacing.

The bilinear warp uses linear interpolation to map from the $(u, v)$ source image plane to the $(x, y)$ destination image plane. It is obtained by the following construction:

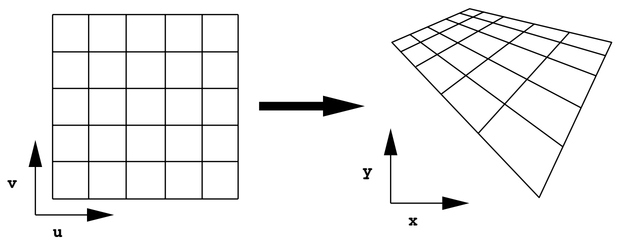


Figure 10.3: Foreshortening due to Perspective

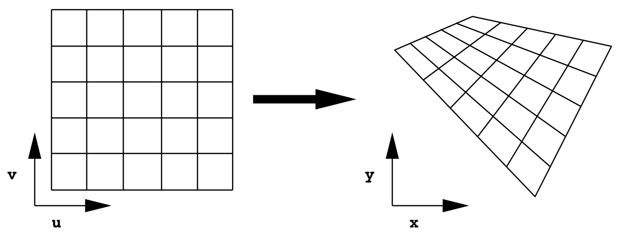


Figure 10.4: Even Spacing under Bilinear Warp

1. number the corners of the source image, and the corresponding corners of the destination image with the same subscripts, as shown in Figure 10.5.

Now, for each point $(u, v)$ in the source image, find the corresponding point $(x, y)$ in the destination image, as shown in Figure 10.6:

2. measure the fraction of distance $u$ along the $u$-axis of the source, and the same fraction along the $[\overline{x_0}, \overline{x_3}]$ and $[\overline{x_1}, \overline{x_2}]$ edges of the destination, to place points $\overline{x_{03}}$ and $\overline{x_{12}}$.

3. measure the fraction of distance $v$ along the $v$-axis, and the same fraction along the $[\overline{x_0}, \overline{x_1}]$ and $[\overline{x_3}, \overline{x_2}]$ edges, to place points $\overline{x_{01}}$ and $\overline{x_{23}}$.

4. the lines $[\overline{x_{03}}, \overline{x_{12}}]$ and $[\overline{x_{01}}, \overline{x_{23}}]$ intersect at the point $(x, y)$ that the input point $(u, v)$ maps to.

Although the bilinear map eliminates the foreshortening effect, it is not the best for all purposes. Its main problem is that, unlike the perspective warp which maps straight lines to straight lines, the bilinear warp maps straight lines to curved (parabolic) lines. This can be seen in Figure 10.7,

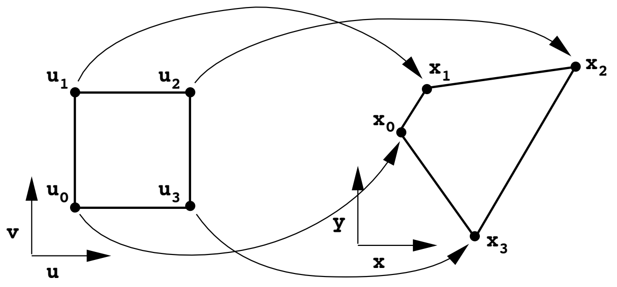


Figure 10.5: Establishing Correspondence Between Corners

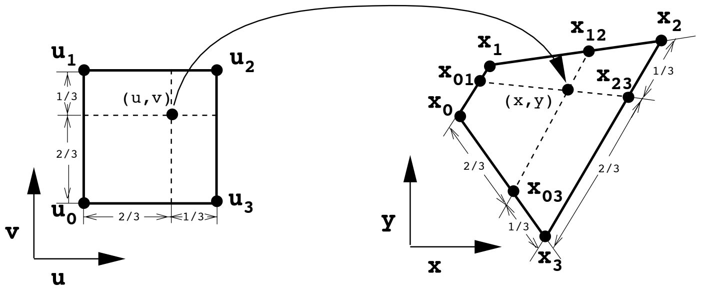


Figure 10.6: Example Map of Point $(u, v) = (2/3, 2/3)$ to Point $(x, y)$

which shows how the diagonals of a rectangle are mapped under both perspective and bilinear maps. The mapping of straight lines to curved lines can lead to quite annoying image distortions for certain kinds of images.

### 10.3.1 Algebra of Bilinear Map

In order to fully understand the bilinear warp, it is necessary to explore its algebra. For simplicity, we will first normalize the coordinates of the input image to the range $[0, 1]$. If the original coordinates $(s, t)$ are on the range

$$s_0 \le s \le s_1,\;\; t_0 \le t \le t_1,$$

then let

$$u = \frac{s - s_0}{s_1 - s_0},$$

and

$$v = \frac{t - t_0}{t_1 - t_0}.$$

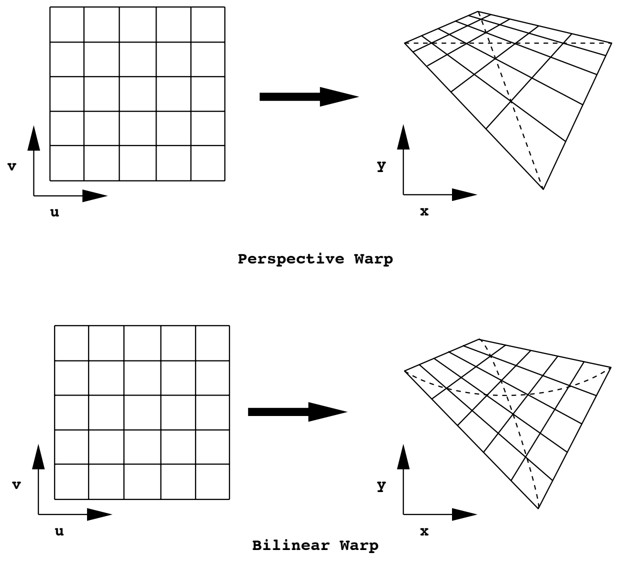


Figure 10.7: Mapping of Diagonals Under Perspective and Bilinear Warps

Now the $u$ and $v$ coordinates can be thought of as fractions of the distance along the original $s$ and $t$ axes.

Note that in the original construction of the bilinear map, steps 2, 3 and 4 are equivalent to constructing the line $[\overline{x_{03}}, \overline{x_{12}}]$ and then measuring fraction $v$ along the line from $\overline{x_{03}}$. This is given by the equations for the forward map:

$$\overline{x_{03}} = (\overline{x_3} - \overline{x_0})u + \overline{x_0},$$

$$\overline{x_{12}} = (\overline{x_2} - \overline{x_1})u + \overline{x_1},$$

and

$$\overline{x} = (\overline{x_{12}} - \overline{x_{03}})v + \overline{x_{03}},$$

where each vector equation really stands for two scalar equations, one for the $x$ coordinate and one for the $y$ coordinate. To compute the inverse map, first combine these six equations into the two equations

$$x = a_0 + a_1 u + a_2 v + a_3 uv$$

$$y = b_0 + b_1 u + b_2 v + b_3 uv,$$

where $a_0 = x_0$, $a_1 = x_3 - x_0$, $a_2 = x_1 - x_0$, $a_3 = x_2 - x_1 - x_3 - x_0$, $b_0 = y_0$, $b_1 = y_3 - y_0$, $b_2 = y_1 - y_0$, $b_3 = y_2 - y_1 - y_3 - y_0$.

With some effort, these equations can be solved for $u$ and $v$ to yield

$$v = \frac{-c_1}{2c_2} \pm \frac{1}{2c_2}\sqrt{{c_1}^2 - 4c_2c_0} \tag{10.2}$$

$$u = \frac{x - a_0 - a_2v}{a_1 + a_3v}, \tag{10.3}$$

subject to the constraints $0 < u < 1$, $0 < v < 1$ and where

$$c_0 = a_1(b_0 - y) + b_1(x - a_0),$$

$$c_1 = a_3(b_0 - y) + b_3(x - a_0) + a_1b_2 - a_2b_1,$$

and

$$c_2 = a_3b_2 - a_2b_3.$$

The decision about adding or subtracting the second term in Equation 10.2 for $v$ is made by applying the constraints that $u$ and $v$ lie in the range $[0, 1]$.

## 10.4 Scanline Approach to Inverse Mapping

Whether we are using a projective map or a bilinear map, we are still left with the problem of how to efficiently compute the output image from the input. Inverse mapping is usually the best approach, and this can be done using a scanline algorithm. First, determine the bounds of the output pixmap by finding the minimum and maximum x and y coordinates of the corners, as shown in Figure 10.8. This will have to be done using the forward map, or may already be known if this is part of an interactive warping tool.

Now, represent each edge in parametric form. For example, for edge $[\overline{x_0}, \overline{x_1}]$ we write

$$x = (x_1 - x_0)t + x_0,$$

and

$$y = (y_1 - y_0)t + y_0,$$

where t is a parameter measuring the fraction of the distance along the edge, measured from $\overline{x_0}$. The intersection of this edge with a specific scanline at $y = y_i$ is then given by solving for $t$

$$t = \frac{y_i - y_0}{y_1 - y_0}$$

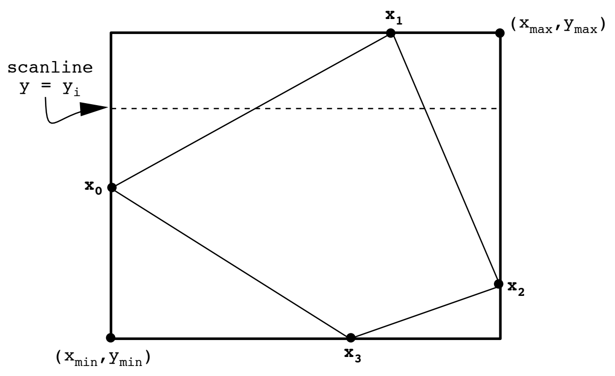


Figure 10.8: Bounding Pixmap Around Corners of Image

and then applying this to the equation for $x$, giving

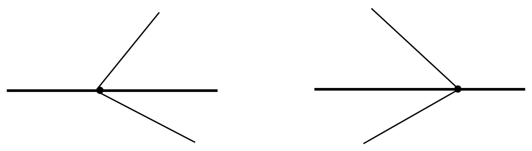
$$x = (x_1 - x_0)\frac{(y_i - y_0)}{y_1 - y_0} + x_0,$$

which is the equation for the $x$ coordinate of the intersection of scanline $y_i$ with the line.

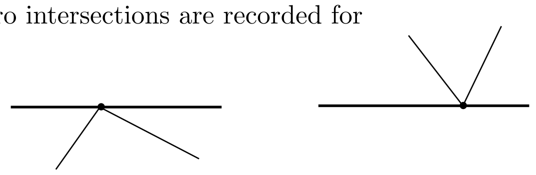
Given this simple way to find the intersection between scanlines and image edges, we can compute the output image with the following algorithm:

```
for(y = y_min; y < y_max; y++){
  list the 2 intersections with scanline y;
  sort intersections by x coordinate, giving x_low, x_high;
  for(x = x_min; x < x_low - 1; x++)
    pixel[x][y] = black;
  for(x = x_low; x < x_high; x++){
    use inverse map to compute (u,v) coordinates in input image
      that correspond to coordinates (x,y) in output image
    pixel[x][y] = pixel value at location (u,v) in source image;
  }
  for(x = x_high + 1; x < x_max; x++)
    pixel[x][y] = black;
}
```

This simple looking algorithm has a few pitfalls. First, `x_low` and `x_high`, as well as `u` and `v` should be rounded to the nearest integer value to form proper indices. Second, care must be taken to deal with situations when a scanline intersects exactly with a corner, so that only one intersection is recorded for



but two or zero intersections are recorded for



If two intersections are recorded, we can draw the corner pixel and if zero intersections are recorded, we can skip the corner pixel. This can be a stylistic decision. There are also some nice ways to organize the algorithm and the data to get some efficiencies (see Homework Nuts and Bolts below).

This approach works for maps from a rectangular image to an arbitrary quadrilateral or from a quadrilateral to a rectangle (inverse). This leads to a general quadrilateral to quadrilateral algorithm with an intermediate rectangular step.

## 10.5 Homework Nuts and Bolts: Projective Transformation

### 10.5.1 Building the transformation matrix M

A composite transformation matrix can easily be built up from a series of simpler transformations. Start by initializing $M$ to the identity matrix

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Then, for each new operation $T$ that is to be concatenated onto the transform, premultiply the matrix $M$ by $T$, replacing $M$ by the product $TM$

$$M \Longleftarrow TM,$$

### 10.5.2 Constructing the output pixmap

Let $\overline{u_0}$, $\overline{u_1}$, $\overline{u_2}$, $\overline{u_3}$ be the coordinates of the four corners of the input image. Apply transform $M$ to each of these in turn, using the steps of a projective transformation covered last class. This will give you the output corners $\overline{x_0}$, $\overline{x_1}$, $\overline{x_2}$, $\overline{x_3}$, as shown in Figure 10.5.

Now, as long as the transformation $M$ is projective, it is certain that the minimum and maximum $x$ and $y$ coordinates of the corners $\overline{x_0}$, $\overline{x_1}$, $\overline{x_2}$ and $\overline{x_3}$ will determine the bounding box around the output image, as shown in Figure 10.9. The bounding box can then be used to determine the necessary width and height of the pixmap for the output image.

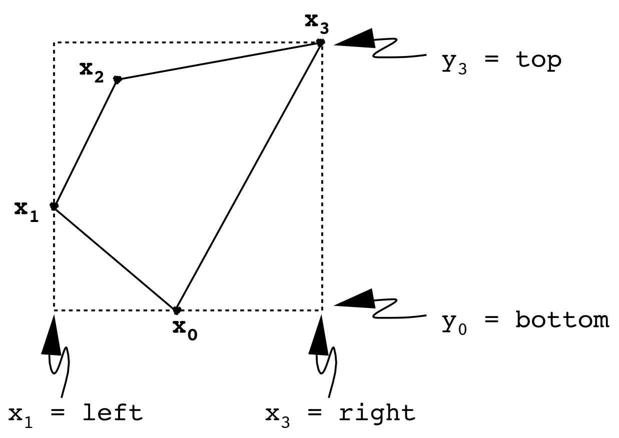


Figure 10.9: Constructing a Bounding Box

### 10.5.3 Finding the inverse transformation

This is simply the inverse of the matrix $M$, as given in Figure 10.2.

### 10.5.4 Constructing an edge table for the inverse map

We color in the output image by using the inverse map to determine which input pixel maps to each output pixel, by iterating over the output pixels. This can cause problems for certain inverse maps, which will send pixels in the output pixmap, that are not within the boundaries of the projected image outline, back to the input pixmap.

To prevent coloring in output pixels that are outside of the image boundary, you will need to be sure to do the inverse map only on pixels within the projected image boundary. The most convenient way to do this is with an edge table.

For each of the projected edges: $[\overline{x_0}, \overline{x_1}]$, $[\overline{x_1}, \overline{x_2}]$, $[\overline{x_2}, \overline{x_3}]$ and $[\overline{x_3}, \overline{x_4}]$ store the following:

- $y_{\text{min}}$ — minimum $y$ coordinate of the edge

- $y_{\max}$ — maximum $y$ coordinate of the edge
- $dx/dy$ — inverse of the slope of the edge, e.g., $\frac{x_1-x_0}{y_1-y_0}$
- $x_0$ — $x$ coordinate of the endpoint having $y$ coordinate $y_{\min}$

Since a horizontal edge, i.e. an edge $ij$ with $(y_i - y_j) = 0$, will not intersect a scanline, each such edge should be discarded. Now, sort the remaining edges by $y_{\min}$, breaking ties using $dx/dy$.

Note: in determining ties and checking for horizontal lines, it is important to allow for some roundoff or truncation error in the floating point numbers. The test `x1 == x2` will only be true if `x1` exactly equals `x2`. Instead, use the test

```
fabs(x1 - x2) < EPSILON
```

where `EPSILON` is some very small number. I would suggest

```
#define EPSILON 1.0e-5
```

i.e. one hundred thousandth.

### 10.5.5 Painting in the output image

The first 2 entries in your edge table will mark the leftmost and rightmost pixels in the projected image on the bottom scan line. For each scan line, color in black up to the first edge, inverse map between edges, and color in black from the second edge, as shown in Figure 10.10.

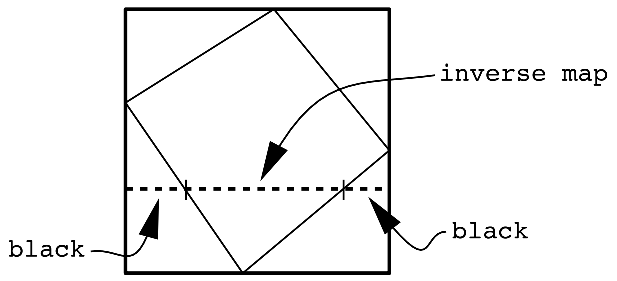


Figure 10.10: Coloring a Scanline in the Output Image

In order to do the bookkeeping on this, with each "active" edge (i.e. each edge which the current scan line crosses) store the $x$ coordinate of the crossing point of the scanline with the edge. This is easy to do. At the start, the crossing point is given by $x_0$. Each time the scanline is advanced, add $dx/dy$ to the crossing $x$ coordinate – that's it! The only remaining bit of bookkeeping is to note when you have reached the top of an "active" edge. At this point, deactivate the edge, and activate the next edge in the table.

### 10.5.6 Data structures and algorithms

The following data structures will be convenient for storing edge information:

```
typedef struct _edge{
  int Ymin, Ymax;
  int X0;
  double dxdy;
  double X;
}Edge;

Edge edge_table[4];
```

The following selection sort algorithm on an array of N integers will put the smallest value in `table[0]`, the second smallest in `table[1]`, etc.

```
int table[N], temp;
int i,j, k;

for(i = 0 ; i < N - 1 ; i++){
  for(k = i, j = i + 1; j < N; j++)
    if(table [j] < table[k])
      k = j;

  if(k != j){
    temp = table[i];
    table[i] = table[j];
    table[j] = temp;
  }
}
```

Chapter 11

# Filtering to Avoid Artifacts

## 11.1 The Warping Process

If you have made some progress on the warping assignment, you will have noticed some artifacts that are introduced into the warped image. Figure 11.1 depicts a typical image warp, and shows how the same warp can produce both maginification and minification in the output image. Where the warped image is magnified compared with the input, the artifacts are the result of oversampling. Where the warped image is minified compared with the input, the artifacts are aliasing errors due to undersampling. In order to understand where these errors are coming from and how to correct them, we need to have a better conceptual view of what we are doing when we warp a digital image.

## 11.2 Sampling and Reconstruction

First, remember that the original image consists of samples of some real world (or virtual) scene. Each pixel value is simply a point sample of the scene. When we view such an image on a screen, we are really viewing a reconstruction that is done by spreading each sample out over the rectangular area of a pixel, and then tiling the image with these rectangles. Figures 11.2a-c show the steps in the process from sampling to reconstruction, looked at in one-dimensional form. When viewed from a distance, or when the scene is sufficiently smooth, the rectangular artifacts in the recon-

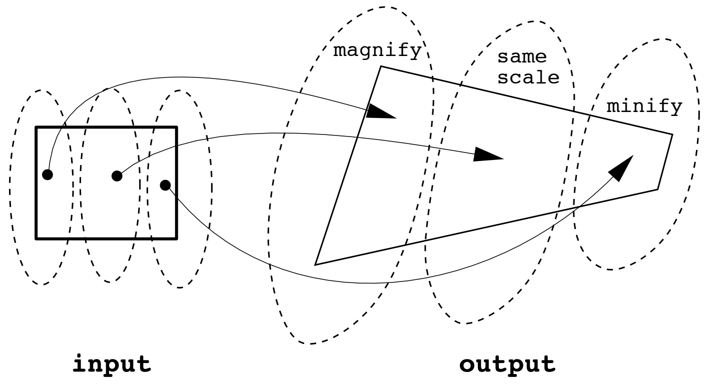


Figure 11.1: Magnification and Minification in the Same Warp

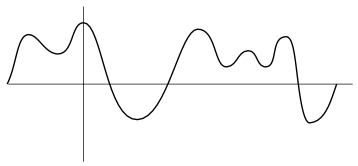
struction are negligible. However, under magnification they become very noticeable as *jaggies* and *blockiness* in the output. Figure 11.2d shows how the extra resampling when magnifying results in multiple samples with the same value, effectively spreading single pixel values in the original image over multiple pixels in the magnified image.

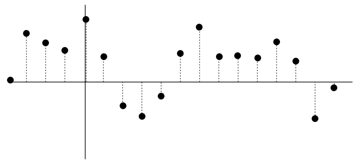
> **Lesson 1**: To reduce magnification artifacts we need to do a better job of reconstruction.

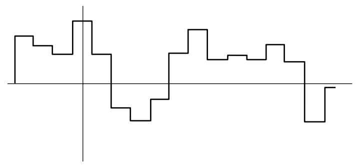
## 11.3 Resampling and Aliasing

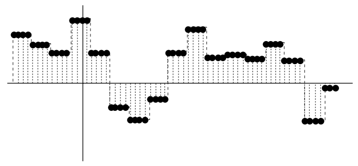
When we minify an image, however, the resampling process causes us to miss many of the reconstructed samples in the original image, leading to missing details, *ropiness* of fine lines, and other aliasing artifacts which will appear as patterning in the output. In the worst case, the result can look amazingly unlike the original, like the undersampled reconstruction shown in Figure 11.2e. The problem here is that the resampling is being done too coarsely to pick up all the detail in the reconstructed image, and worse, the regular sampling can often pick up high frequency patterns in the original image and reintroduce them as low frequency patterns in the output image.

> **Lesson 2**: To reduce minification artifacts we must either 1) sample more finely than once for each output pixel, or 2) smooth the reconstructed input before sampling.


a) brightness along a scanline across the original scene


b) brightness samples along the same scanline


c) pixel-like reconstruction of original line from the samples


d) resampling under magnification

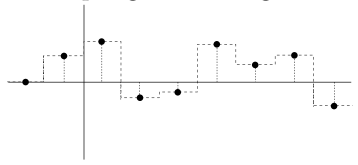

e) resampling under minification

Figure 11.2: The Sampling, Reconstruction and Resampling Processes

## 11.4 Sampling and Filtering

Although we have not yet developed the mathematics to state this precisely, it will be shown in a later chapter that under certain conditions it is theoretically possible to exactly recover an original unsampled image given only the samples in the pixmap. It will be possible to do this when the original image is smooth enough given the sampling density that we used to sample the original image to capture it in the pixmap. The criterion, loosely stated, is that the original image should not have any fluctuations that occur at a rate greater than 1/2 of the sampling rate. In other words, if the original image has an intensity change that from dark to light and back to dark again (or *vica versa*), that there should be at least two samples taken in the period of this fluctuation. If we have more, that is even better. Another way of saying this is that any single dark to light or light to dark transition should be wide enough that at least one sample is taken in the transition. This criterion must hold over the whole image, for the shortest transition period in the image.

If we let the sampling period (i.e. the space between samples) be $T_S$, then this is useful in two ways in considering the image warping problem:

1. A perfect reconstruction could be obtained by prefiltering the sampled image with a filter that smooths out all fluctuations that occur in a space smaller than $2T_S$.

2. Aliasing under resampling could be avoided by filtering the reconstructed image to remove all fluctuations with periods smaller than twice the resampling period, before doing the resampling.

In simple terms, we need to: 1) Do a nice smooth reconstruction, not the crude one obtained by spreading samples over a pixel area, and 2) possibly further smooth the reconstruction so that when we resample, we are always doing the resampling finely enough to pick up all remaining detail.

## 11.5 The Warping Pipeline

Conceptually, what we are trying to do when we warp a digital image is the process diagrammed in Figure 11.3. It consists of three steps:

1. reconstruct a continuous image from the sampled input image, by filtering out all fluctuations with period less than twice the original sampling period $T_S$,

2. warp the continuous reconstruction into a new shape,

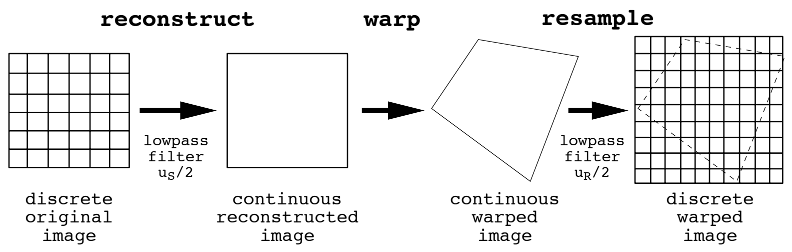


Figure 11.3: A Conceptual View of Digital Image Warping

3. filter the warped image to eliminate fluctuations with period less than twice resampling period to be used in the next step, and

4. resample the warped image with a resampling period $T_R$ to form a new discrete digital image.

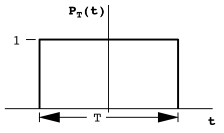
However, in fact we do not really have any way in the computer of reconstructing and manipulating a continuous image, since by its nature everything in the computer is discrete. Now, since the reconstructed and warped continuous versions of the image never really get constructed, it makes sense to think of combining the reconstruction and low-pass filtering operations of steps 1 and 3 into a single step, that somehow would take into account the warp done in step 2. To understand just how this might work, it would help to have a view of the filtering process that would operate over the original *spatial domain* image, rather than on the image after it has been transformed into the frequency domain.

## 11.6 Filtering as Convolution

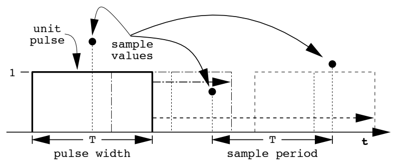
In order to understand what a filter might look like in the spatial domain, let us start by looking at the process of reconstruction – remembering that in the frequency domain this turned out to be a filtering process that discarded high frequencies. To do this we will need to expand our notion of convolution to include the convolution of continuous functions. Converting our image samples into pixels – i.e. reconstructing an image from samples – is mathematically a convolution of the samples with a continuous square unit pulse function. A one-dimensional, analogy follows.

Define a one-dimensional unit pulse function of width $T$ to be

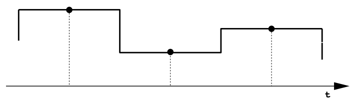
$$P_T(t) = \begin{cases} 1, & -T/2 < t < T/2 \\ 0, & \text{otherwise,} \end{cases}$$



a) unit pulse function



b) *sliding* unit pulse over samples



c) resulting reconstruction

Figure 11.4: Reconstruction by Convolution of Samples with Unit Pulse

as shown in Figure 11.4a. Now, think of the pulse as sliding back and forth along the scanline, as in Figure 11.4b. Wherever you want a reconstructed value from the samples, simply center the pulse over the point along the scanline at which you want the image value and select the value of the sample that the pulse overlaps. Another way of thinking of this is that we multiply the value of the shifted pulse by all of the samples that it overlaps and add all of the products. If we choose a pulse whose width is exactly the same as the sampling period used to capture the original samples, the non-zero portion of the pulse will overlap only one of the samples at a time, and the reconstruction process will simply spread each sample value out 1/2 the sampling period $T$ in each direction. This will result in a reconstructed scanline as shown in Figure 11.2c. The unit pulse serves as a *weighting* function, weighting each sample's contribution to the final reconstruction. A weighting function used in this way is also known as a *convolution kernel*.

The convolution process is described mathematically by the integral

$$f_R(t) = \int_{-\infty}^{\infty} f_T(\lambda) h(t-\lambda) d\lambda,$$

where: $f_R$ is the reconstructed function, $f_T$ is the function $f(t)$ sampled with sampling period $T$, $h$ is the convolution kernel, and $\lambda$ is a shift parameter that determines where the kernel $h$ is centered.

If $f_T$ is zero except at integer multiples of its sampling period $T$, and there are $N$ samples numbered 0 through $N-1$, then the integral reduces to the sum

$$f_R(t) = \sum_{n=0}^{N-1} f_T(nT)h(t-nT).$$

For example, if $T = 1$, then

$$f_R(2) = f_T(0)h(2) + f_T(1)h(1) + f_T(2)h(0) + \cdots + f_T(N-1)h[2-(N-1)].$$

Now, in this example, if the kernel function $h$ is of width 1, then all the terms in the sum will be zero except the term $f_T(2)h(0)$, giving the same kind of squared off result as in Figures 11.2c and 11.4c.

If we choose the "tent" function of base width two shown in Figure 11.5a as our convolution kernel $h$, then doing a convolution as in Figure 11.5b, all terms would be zero except $f_T(2)h(0)$ only if the kernel is centered exactly over a sample. If we choose a value of $t$ that is not exactly at a sample point, the tent function will overlap two samples, and we will get a weighted average of adjacent sample points. For example,

$$f_R(2.5) = f_T(2)h(0.5) + f_T(3)h(-0.5) = 0.5f_T(2) + 0.5f_T(3)$$

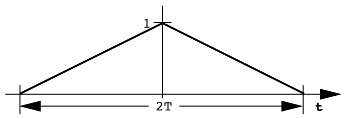
This yields a linear weighted reconstruction as shown in Figure 11.5c, essentially connecting the samples by straight lines. Obviously this is a much "smoother" result than that obtained with the unit pulse kernel, and in practical applications is often enough to eliminate the most obvious reconstruction artifacts in a magnified image.

There is a whole science behind selecting and designing convolution kernels for filtering, and this will be the subject of the following sections.

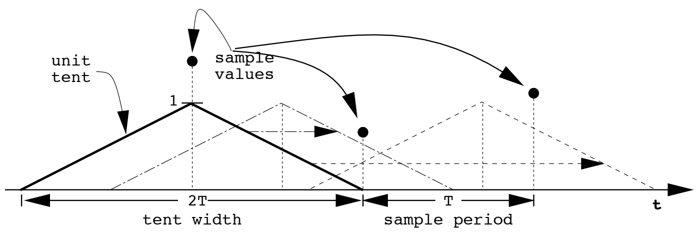
## 11.7 Ideal vs. Practical Filter Convolution Kernels

### 11.7.1 The Ideal Low Pass Filter

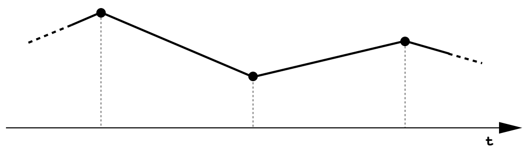
Assume that we have sampled continuous function $f(t)$ with a sampling interval $T$, to obtain the sampled signal $f_T(t)$. Assuming that the original continuous signal were smoothed so that it has no fluctuations with size smaller than $2T$, then we know that we should be able to reconstruct $f(t)$



a) unit tent function



b) *sliding* unit tent over samples



c) resulting reconstruction

Figure 11.5: Reconstruction by Convolution of Samples with Unit Tent

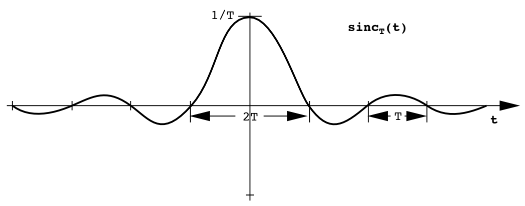


Figure 11.6: Unit Sinc Function of Period $2T$

exactly. Although the mathematics behind precisely how to do this must wait for a later chapter, we can, at this point, develop a good working technique for at least doing a good approximate reconstruction.

It is a well known result in the theory of filtering that an ideal low-pass filter, that cuts off all fluctuations of period less than $2T$ is given by the convolution kernel

$$h(t) = \frac{\sin \pi t/T}{\pi t},$$

where the function $\frac{\sin\theta}{\theta}$ is known as the *sinc* function. The graph of this function is shown in Figure 11.6. Now, to filter out fluctuations that are shorter than $2T$ we need to convolve with the octopus-like sinc function of period $2T$. This is not such a happy result, because the sinc never goes to a constant zero value. Thus, we are left with the need to convolve all of our image samples with a convolution kernel of infinite extent – a bit hard to do!

### 11.7.2 Practical Filter Kernels

Thus, practical image filtering cannot be done using the ideal low-pass filter. However, there are many ways in which we can approximate the ideal sinc kernel. In fact, the quality of a low-pass filter will be judged by how nearly it approximates the sinc, recognizing that there is always the trade-off between wanting the computational speed advantages of a narrow convolution kernel versus wanting a high quality result. Some of the ways of approximating the sinc are shown in Figure 11.7. The simplest is to truncate the sinc spatially as in Figure 11.7a. A second approach is to use another function that has the same general local shape as the sinc, such as the spatial pulse and the tent kernels that we discussed in the last section. Figure 11.7b shows that these can both be seen as approximations to the sinc, albeit poor ones!. The trick that tends to yield the best tradeoff between simplicity and quality is to *window* the sinc by multiplying by a smooth function that goes to zero beyond a certain distance, as shown in Figure 11.7c.

Another technique is to approximate the windowed sinc by a spline curve. These and other ideas are given a very thorough treatment in Wolberg, Chapter 5.

### 11.7.3 Practical Convolution Filtering

We have seen the mathematical representation of the convolution, but now need to move to a more practical level and learn how to efficiently compute a convolution. The convolution problem is shown in Figure 11.8. Put in simple terms, the problem is: given a convolution kernel $h$, and a set of uniform samples $f_T$, compute the value of the convolution of $h$ and $f_T$ for an arbitrary coordinate $t$. This essentially allows us, in one step, to both reconstruct and resample our function at any arbitrary resampling period.

The mathematical expression for the convolution of kernel $h$ with sampled function $f_T$, evaluated at coordinate $t$ is

$$f^*(t) = \sum_{n=-\infty}^{\infty} f_T(nT)h(t - nT),$$

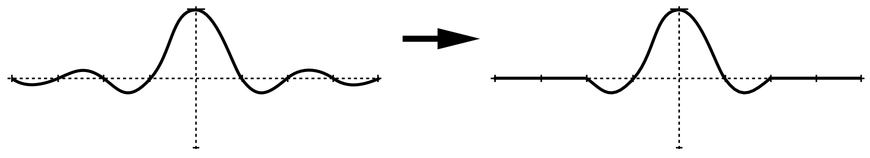
where $n$ is the sample number, $T$ is the sample period, and $f_T$ is taken to be zero beyond the region for which samples are stored.

Now, if we assume that $h(t)$ has a finite extent, say 3 units of the original sampling period $T$ on either side of $t$, then the sum reduces to centering the convolution kernel over the position $t$, multiplying it by each of the 3 samples on either side of $t$, and then summing the result. This can be written as the sum

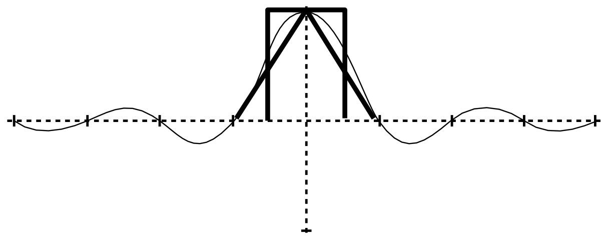
$$f^*(t) = \sum_{m=-2}^{3} f_T[(\lfloor t/T \rfloor + m)T]h[t/T - (\lfloor t/T \rfloor + m)T],$$

Here, $\lfloor t/T \rfloor$ is the greatest integer less than or equal to $t/T$, or in other words it is simply the sample number of the sample just to the left of position $t$. Thus, $f_T[(\lfloor t/T \rfloor + n)T]$ gives the sample value just to the left of $t$ when $n = 0$, and the two to the left of this when $n = -1, -2$, and the three to the right when $n = 1, 2, 3$. This concept is diagrammed in Figure 11.9.

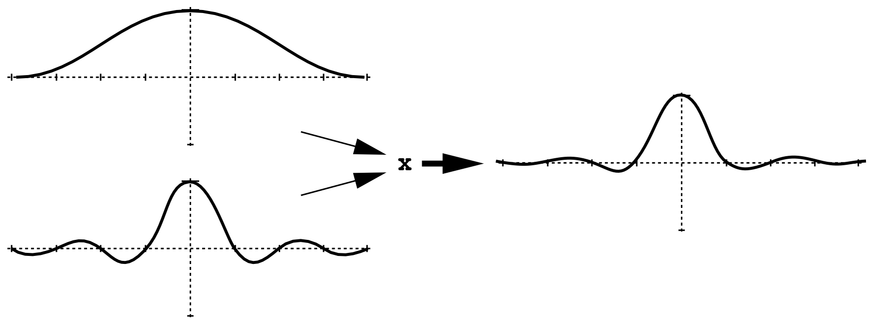
Now, with respect to the array indices in a pixmap, the sample period $T$ is just 1, i.e. horizontally or vertically a move of 1 unit moves 1 pixel across a scan line or along a column.



a) simple truncation of sinc to width $4T$



b) unit pulse and tent as approximations to the sinc



c) windowing the sinc with a finite function

Figure 11.7: Approximating the Infinite Sinc Function to Achieve a Finite Convolution Kernel

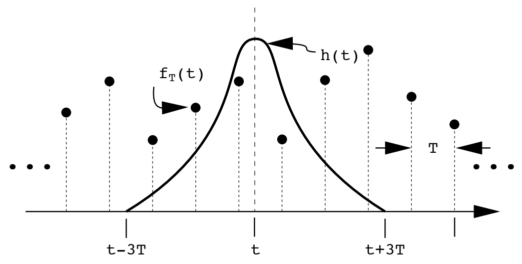


Figure 11.8: Convolution of Kernel $h$ with samples $f_T$ at Position $t$

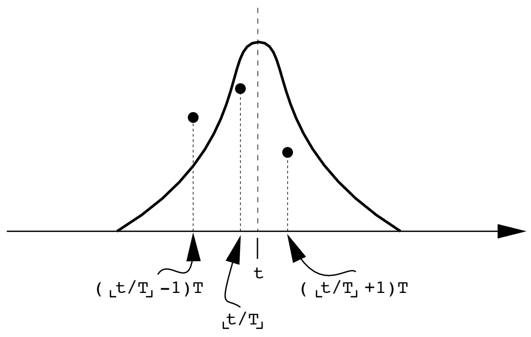


Figure 11.9: Values of $h$ are Needed Only at Integer Valued Offsets from $t$

Thus, sampling the filtered pixmap is simple. If $W(u,v)$ is the warp function, and $W^{-1}(x,y)$ is its inverse, then

$$\begin{bmatrix} u \\ v \end{bmatrix} = W^{-1}(x,y).$$

For a kernel of width 6, we sample the input image along scanline $\lfloor v \rfloor$ at positions $\lfloor u \rfloor - 3$ , $\lfloor u \rfloor - 2$ , $\lfloor u \rfloor - 1$ , $\lfloor u \rfloor$ , $\lfloor u \rfloor + 1$ , $\lfloor u \rfloor + 2$ , and $\lfloor u \rfloor + 3$; and multiply each of the samples by

$$h[u - (\lfloor u \rfloor + n)],$$

or

$$h[(u - \lfloor u \rfloor) + n],$$

where $n$ is the sample index from $-2$ to 3. (Note that $(u - \lfloor u \rfloor)$ is simply the fractional part of the real number u.) The sum of all of these six products is our approximation to the pixel in the reconstructed warped image resampled at the point $(x,y)$. Of course, this operation needs to be done both along the columns and along the scanlines that fall under the filter kernel if the image warp involves distortion in both directions. The horizontal filter should be done to an intermediate image, and then that image should be vertically filtered into the final image, so that the actual convolution is over a rectangular area about the point $(u,v)$.

You should note that this calculation requires computing the kernel function $h$ six times for each $(x,y)$ coordinate pair under the inverse map. This can be avoided by an initialization step that samples $h$ finely, computing $h$ once for each of these samples and storing the resulting sample values in a table, as diagrammed in Figure 11.10. This can be done very finely (e.g. 500 samples) since it only need be done once. Then when a value for $h$ is needed it can be obtained from the table by simply selecting the sample value in the table whose index is nearest to the coordinate for which a value for $h$ is needed. Now, the whole process of taking a single filtered sample is simply a series of multiplies and adds.

## 11.8 Antialiasing: Spatially Varying Filtering and Sampling

### 11.8.1 The Aliasing Problem

Remember that in the image warping problem there are two places where filtering is important:

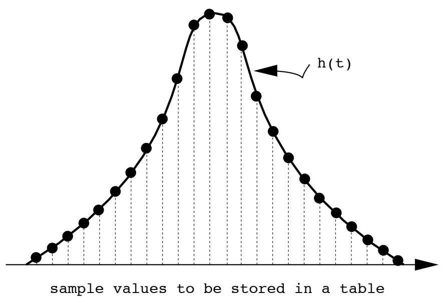


Figure 11.10: Precalculation of Many Samples of Convolution Kernel $h$

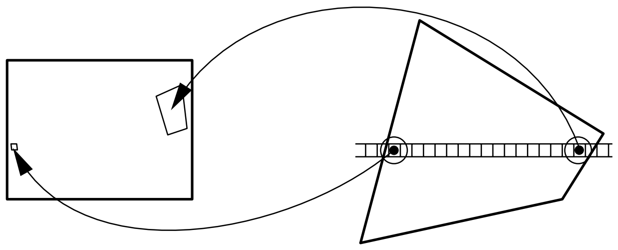


Figure 11.11: Area Sampling, Pixel Area Projected Back Into Input Image

1. reconstruction filtering (critical when magnifying)

2. antialiasing filtering (critical when minifying).

In the last section we looked at the general idea of practical filtering, with special attention paid to the reconstruction problem. Here we will focus specifically on the aliasing problem.

Recall that aliasing will occur when resampling an image whose warped reconstruction has frequency content above 1/2 the resampling frequency. To prevent this we can either:

1. filter the reconstructed and warped image so that its highest frequencies are below 1/2 the resampling rate,

2. adjust the resampling rate to be at least twice the highest frequency in the reconstructed warped image.

In general, the problem in both cases is that a warp is non-uniform, meaning that the highest frequency will vary across the image, as schematized in Figure 11.11.

We can deal with this by filtering the entire image to reduce the highest frequencies found anywhere in the image, or sample everywhere at a rate twice the highest frequency found anywhere in the image. Either solution can be very wasteful (sometimes horribly wasteful) of computation time in those areas of the image that do not require the heavy filtering or excess sampling. The answer is to use either spatially varying filtering or adaptive sampling.

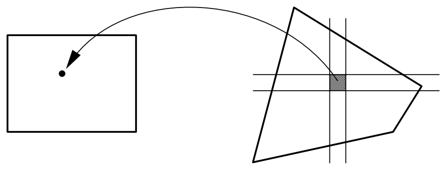
### 11.8.2 Adaptive Sampling

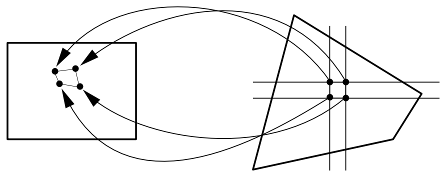
We can think of the inverse mapping process described in Chapter 7, and shown in Figure 11.12a, as a point sampling of the input image, over the inverse map, guided by a uniform traversal over the output image. This technique is simple to implement but, as we have discovered, leaves many artifacts, and we need to look at more powerful approaches.

One method for doing sampling without aliasing is *area sampling*. This technique is diagrammed in Figure 11.12b. Area sampling projects the area of an output pixel back into the input image and attempts to take a weighted average of covered and partially covered pixels in the input. Area sampling produces good results under minification, i.e. aliasing is eliminated. However, to do area sampling accurately is very time consuming to compute and hard to implement. Fortunately, experience shows that good success can be had with simpler, less expensive approximations to area sampling.

A common approximation to area sampling is known as *supersampling*. Here, we still take only point samples, but instead of sampling the input only once, or over an entire area as with area sampling, multiple samples are taken per output pixel. This process is diagrammed in Figure 11.12c. A sum or weighted average of all of these samples is computed before storing in the output pixel. This gives results that can be nearly as good as with area sampling, but results vary with the amount of minification. The problem with this method is that if you take enough samples to eliminate aliasing everywhere, you will be wasting calculations over parts of the image that do not need to be sampled so finely. Although supersampling can be wasteful, some economies can be had. Figure 11.13 shows how some careful bookkeeping can be done to minimize the number of extra samples per pixel. If samples are shared at the corners and along the edges of pixels, there is no need to recompute these samples when advancing to a new pixel.

*Adaptive supersampling* improves on simple supersampling by attempting to adjust the sampling density for each pixel to the density needed to avoid aliasing. The process used in adaptive supersampling is as follows:


a) point sampling


b) area sampling

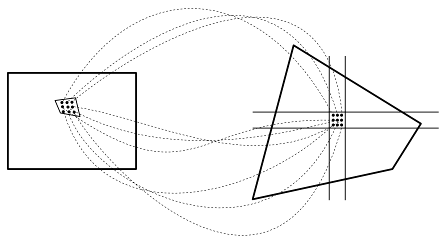

c) supersampling

Figure 11.12: Sampling Techniques Under Inverse Mapping

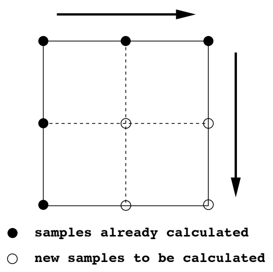


Figure 11.13: Sharing of Samples Under Inverse Mapping. Scan is Left to Right, Top to Bottom

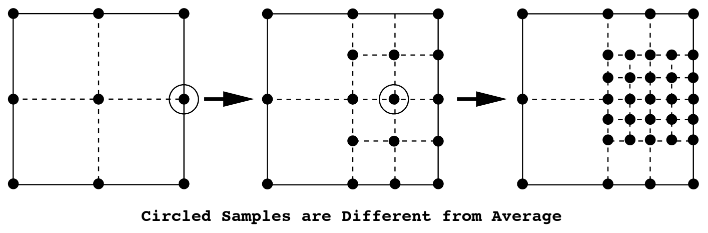


Figure 11.14: Adaptive Supersampling

1. Do a small number of test samples for a pixel, and look at the maximum difference between the samples and their average.

2. If any samples differ from the average by more than some predefined threshold value, subdivide the pixel and repeat the process for each sub-pixel that has an extreme sample.

3. Continue until all samples are within the threshold for their sub-pixels(s) or some maximum allowable pixel subdivision is reached.

4. Compute the output pixel value as an area-weighted average of the samples collected above.

Any regular sampling approach will be susceptible to aliasing artifacts introduced by the regular repeating sampling pattern. A final refinement to the adaptive supersampling technique is to *jitter* the samples so that they no longer are done on a regular grid. As shown in Figure 11.15, an adaptive sampling approach is taken but each sample is moved slightly by a random perturbation from its regular position within the output pixel before the inverse map is calculated. Irregular sampling effectively replaces the low frequency patterning artifacts that arise from regular sampling with high frequency *noise*. This noise is usually seen as a kind of graininess in the image and is usually much less visually objectionable than the regular patterning from aliasing.

### 11.8.3 Spatially Variable Filtering

Antialiasing via filtering is a fundamentally different approach from antialiasing via adaptive supersampling, and leads to another basic problem. Since the frequency content of an image will vary due to the warp applied, efficiency issues require that whatever filtering is done be adapted to this changing frequency content. The idea behind spatially varying filtering is

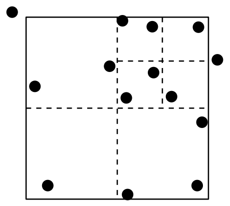
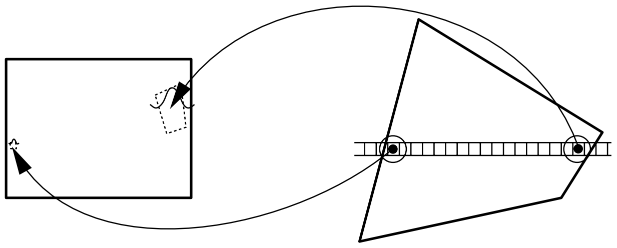


Figure 11.15: Jittered Adaptive Supersampling



Kernel size expands to match sampling rate. For high sampling rate, kernel size is small, for low rate it is large.

Figure 11.16: Spatially Varying Filtering

to filter a region of a picture only as much as needed, varying the size of the filter convolution kernel with the variance in frequency content.

*Spatially variable filtering* schemes attempt to pre-filter each area of the input picture, using only the level of filtering necessary to remove frequencies above either 1/2 the sampling rate or 1/2 the resampling rate, whichever is lower under the inverse map. The idea here is that the same convolution kernel is used across the whole image but it changes scale to accommodate the resampling rate. Where the output is minified, the scale of the kernel being convolved with the input image is increased. Where the output is magnified, it is decreased. Remember, of course, that the bigger the kernel the more computation is involved, so the problem with this approach is that it gets very slow over minification.

A crude but efficient way of implementing spatially varying filtering is the *summed area table* scheme. It is crude in that it only allows for a box convolution kernel, but fast in that it does a $128 \times 128$ convolution as fast as a $4 \times 4$ convolution! The trick is to first transform the input image into a data structure called a summed area table (SAT). The SAT is a rectangular array that has one cell per pixel of the input image. Each cell in the SAT

| | | | |
|---|---|---|---|
| 2 | 1 | 3 | 2 |
| 1 | 3 | 2 | 1 |
| 3 | 2 | 1 | 1 |
| 1 | 2 | 1 | 3 |

input image

$\Longrightarrow$

| | | | |
|---|---|---|---|
| 7 | 15 | 22 | 29 |
| 5 | 12 | 16 | 21 |
| 4 | 8 | 10 | 14 |
| 1 | 3 | 4 | 7 |

summed area table

Figure 11.17: Summed Area Table

corresponds spatially with a pixel in the input, and contains the sum of the color primary values in that pixel and all other pixels below and to the left of it. An example of a $4 \times 4$ SAT is shown in Figure 11.17.

The SAT can be computed very efficiently by first computing its left column and bottom row, and then traversing the remaining scan lines doing the following calculation:

```
SAT[row][col] = Image[row][col] + SAT[row-1][col] +
                    SAT[row][col-1] - SAT[row-1][col-1];
```

where `SAT` is the summed area table array and `Image` is the input image pixmap, and array indexing out of array bounds is taken to yield a result of 0.

Once the SAT is built, we can compute any box-filtered pixel value for any integer sized box filter with just 2 subtracts, 1 add and 1 divide. The computation of the convolution for one pixel is

```
BoxAvg = (SAT[row][col] - SAT[row-w][col] -
            SAT[row][col-w] + SAT[row-w][col-w]) / ^2;
```

where `w` is the desired convolution kernel width and `(row, col)` are the coordinates of the upper right hand corner of the filter kernel. Note, that the size `w` of the convolution kernel has no effect on the time to compute the convolution. The required size of the convolution can be approximated by back projecting a pixel and getting the size of its bounding rectangle in the input, as shown in Figure 11.18.

Looked at mathematically, if the inverse map is given by functions $u(x, y)$ and $v(x, y)$, then the bounding rectangle size is given by

$$du = \max(\partial u/\partial x, \partial u/\partial y)$$

in the horizontal direction, and

$$dv = \max(\partial v/\partial x, \partial v/\partial y)$$

in the vertical direction, as shown in Figure 11.19. Alternatively, the euclidian distances

$$(du)^2 = (\partial u/\partial x)^2 + (\partial u/\partial y)^2$$

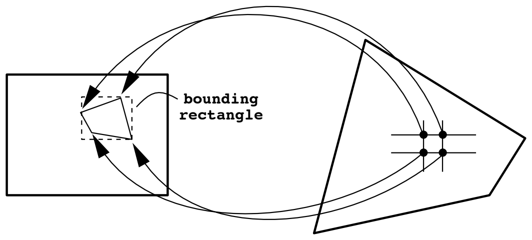


Figure 11.18: Finding the Input Bounding Rectangle for an Output Pixel

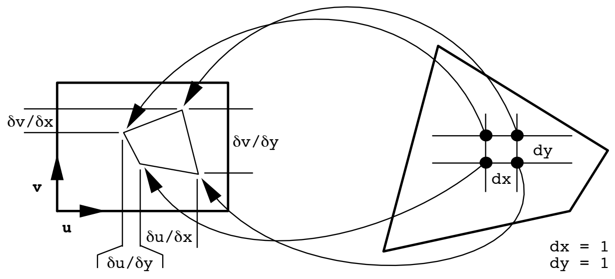


Figure 11.19: Measuring the Required Kernel Size Under an Inverse Map

and

$$(dv)^2 = (\partial v/\partial x)^2 + (\partial v/\partial y)^2$$

can be used.

### 11.8.4 MIP Maps and Pyramid Schemes

A final and very popular technique for speeding the calculation of a varying filter kernel size is to use a Multiresolution Image Pyramid or *MIP-Map* scheme for storing the image. The idea here is to get quick access to the image prefiltered to any desired resolution by storing the image and sequence of minified versions of the image in a pyramid-like structure, as diagrammed in Figure 11.20. Each level in pyramid stores a filtered version of the image at 1/2 the scale of the image below it in the pyramid. In the extreme, the top level of the pyramid is simply a single pixel containing the average color value across the entire original image. The pyramid can be stored very neatly in the data structure shown in Figure 11.21, if the image is stored as an array of RGB values.

The MIP-Map is somewhat costly to compute, so the scheme finds most use in dealing with textures for texture mapping for three-dimensional com-

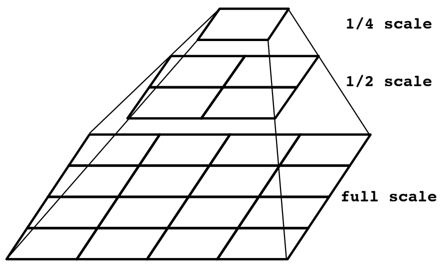


Figure 11.20: An Image Pyramid

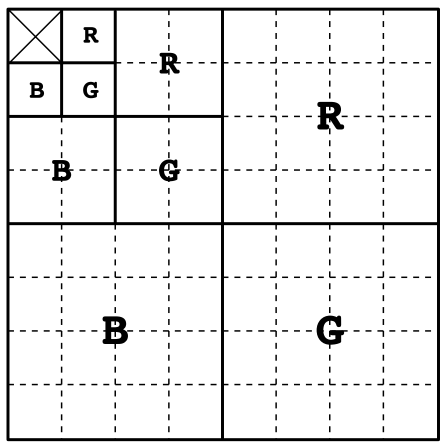


Figure 11.21: Data Structure for a MIP-Map

puter graphics rendering. Here, the MIP-Map can be computed once for a texture and stored in a texture library to be used as needed later. Then when the texture is used, there is very little overhead required to complete filtering operations to any desired scale.

From the MIP-Map itself, it may seem that it is only possible to do magnification or minification to scales that are powers of 2 from the original scale. For example if we want to scale the image to 1/3 its original size, the 1/2 scale image in the MIP-Map will be too fine, and the 1/4 scale image will be too coarse. The solution to this problem is to interpolate across pyramid levels, to get approximation to any resolution image. We would sample at the levels above and below the desired scale, and then take a weighted average of the color values retrieved. The idea is simply to find the pyramid level where an image pixel is just larger than the projected area of the output pixel, and the level where a pixel is just smaller than this area. Then we compute the pixel coordinates at each level, do a simple point sample at each level, and finally take the weighted average of the two samples.

# Chapter 12

# Scanline Warping Algorithms

## 12.1 Scanline Algorithms

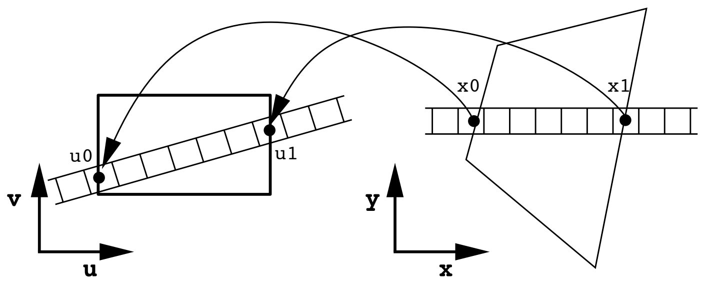
A scanline algorithm is one in which the main control loop iterates over the scanlines of an image and an inner loop iterates across each scanline, as in the code fragment of Figure 12.1. Often these algorithms alternate passes over the scanlines and then the columns of the image.

```
...               // loop initialization
for(y = 0; v < height; y++){
  ...             // scanline initialization for scanline y
  for(x = 0; x < width; x++){
    ...           // operation on pixel x on scanline y
  }
  ...             // scanline y wrap-up
}
```

Figure 12.1: Basic Organization of a Scanline Algorithm

A scanline approach to constructing image manipulation algorithms can often allow the realization of speedups that exploit coherencies that may exist along scanlines or down columns. The observation that neighboring pixels have similar values is just one such coherency that can lead to important savings in computation. Another idea that can save time is to assume that a movement across a pixel is so small that certain nonlinearities can be ignored and sometimes simple linear interpolations can be used in advancing from pixel to pixel across a scanline.

For example, one such trick would be to assume that an inverse map is linear across a short span. Then, as diagrammed in Figure 12.2, instead of computing the inverse map for every pixel, compute it for the two ends of a scanline span (`x0` and `x1` in the figure), yielding two corresponding $(u, v)$ coordinates pairs in the input image (`u0` and `u1` in the figure). Then, while advancing across the output scanline, calculate intermediate input $(u, v)$ values by linear interpolation, as shown in the code fragment in Figure 12.2.



```
for(y = 0; y < height; y++){
  determine x0 and x1 of a span on scanline y;
  (u0, v0) = inv_map(x0, y);      // inverse map of span left end
  (u1, v1) = inv_map(x1, y);      // inverse map of span right end
  dudx = (u1 - u0) / (x1 - x0);   // change in u per pixel
  dvdx = (v1 - v0) / (x1 - x0);   // change in v per pixel
  u = u0;
  v = v0;
  for(x = x0; x <= x1; x++){
    out[x][y] = in[u][v];
    u += dudx;                    // interpolate to next u value
    v += dvdx;                    // interpolate to next v value
  }
}
```

Figure 12.2: Approximation of the Inverse Map by Linear Interpolation

This trick will work exactly for affine warps, since these maps and their inverses are simply linear functions and translations in $x$ and $y$. For non-affine warps, this approach will lead to errors in the image mapping, that will be more or less severe, depending upon the degree of the nonlinearity and the length of the span being interpolated in the input image space $(u, v)$. However, the approach tends to be much faster than the exact computation, which involves computing the inverse map for each pixel.

A further trick that will often make the linear interpolation idea work well even for nonlinear warps is to subdivide the input image, using the forward projection of the subdivision to subdivide the output image. This idea is illustrated in Figure 12.3. When interpolations are done, they are done across the subregions, thus making the spans shorter and minimizing the effects of nonlinearities.

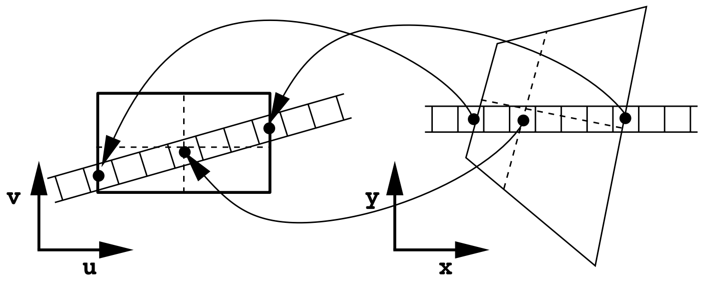


Figure 12.3: Subdivision to Minimize the Effect of Nonlinearities

For the special case of a perspective warp, a trick which allows interpolation to compute the exact inverse map is to do the interpolation in $(u, v, w)$ space, before dividing by $w$ to normalize the $(u, v)$ coordinates. In other words, instead of interpolating only $u$ and $v$, interpolate $u$, $v$ and $w$. This trick works because the inverse map in three-dimensional homogeneous coordinates is linear, up until normalization by the $w$ coordinate. Figure 12.4 shows the code of Figure 12.2 modified to do interpolation in $(u, v, w)$ space.

## 12.2 Separable Scanline Algorithms

A separable algorithm is one in which the map can be factored into two or more seperate maps, each of which operates only over scanlines or over columns. This idea is diagrammed in Figure 12.5. The scanline operations change only the horizontal coordinate and the column operations change only the vertical coordinate.

Algebraically, what is meant here is this. Given a warp

$$(x, y) = f(u, v) = [X(u, v), Y(u, v)],$$

```
for(y = 0; y < height; y++){
  determine x0 and x1 of a span on scanline y;
  (u0, v0, w0) = inv_map(x0, y);  // inverse map of span left end
  (u1, v1, w1) = inv_map(x1, y);  // inverse map of span right end
  dudx = (u1 - u0) / (x1 - x0);   // change in u per pixel
  dvdx = (v1 - v0) / (x1 - x0);   // change in v per pixel
  dwdx = (w1 - w0) / (x1 - x0);   // change in w per pixel
  u = u0;
  v = v0;
  w = w0;
  for(x = x0; x <= x1; x++){
    nu = u / w;
    nv = v / w;
    out[x][y] = in[nu][nv];
    u += dudx;                    // interpolate to next u value
    v += dvdx;                    // interpolate to next v value
    w += dwdx;                    // interpolate to next w value
  }
}
```

Figure 12.4: Linear Interpolation in $(u, v, w)$ Space before Normalization by $w$

we say that the warp is two-pass separable if we can find a function $G(u, v)$ such that

$$f(u, v) = [X(u, v), G(X(u, v), v)].$$

In other words, if we can find a function $G$ such that

$$Y(u, v) = G(X(u, v), v).$$

If we can find such a function, then we can first apply the map

$$(u, v) \mapsto (X(u, v), v) = (x, v),$$

keeping the vertical coordinate fixed and changing only the horizontal coordinate and then apply the map

$$(x, v) \mapsto (x, G(x, v)) = (x, y)$$

to the result, keeping the horizontal coordinate fixed and changing only the vertical.

In general, the seperable property holds when the function $X(u, v)$ can be partially inverted to yield a solution for $u$ in terms of $x$ and $v$. Whenever

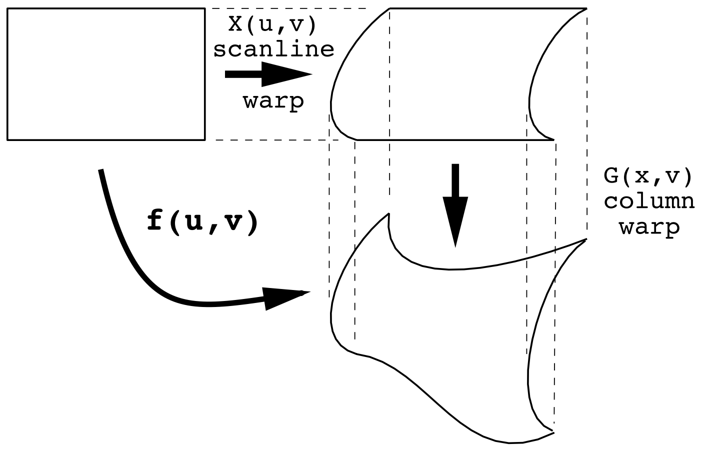


Figure 12.5: Separation of a Warp into Sequential Scanline and Column Warps

this can be done, then it is a simple matter to use this solution for $u$ as the input to the function $Y(u, v)$, to yield $G(x, v)$.

### 12.2.1 Separation of projective warps

The projective warps can all be implemented via separable algorithms. Note that all of the projective warps are expressed as a multiplication of the input image coordinates $(u, v)$ by a $3 \times 3$ homogeneous projection matrix $M$, i.e.

$$M \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}.$$

For the special case of an affine warp, where $w$ is 1, separation is achieved by simply factoring the matrix $M$ into the product of two matrices, the first of which affects only the horizontal coordinate $u$, and the second of which affects only the vertical coordinate $v$. The portion of the matrix which affects the output $w$ coordinate can be placed in either of the two factors, or distributed between them. If the matrix $M$ is given by

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix},$$

then one possible factorization would be

$$M = VU,$$

where

$$U = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and

$$V = \begin{bmatrix} 1 & 0 & 0 \\ d/a & (ea - bd)/a & (af - dc)/a \\ 0 & 0 & 1 \end{bmatrix}.$$

In the case of a general projective warp, where we cannot assume that $w$ is 1,

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}.$$

Now we have

$$X(u, v) = \frac{au + bv + c}{gu + hv + 1} \tag{12.1}$$

and

$$Y(u, v) = \frac{du + ev + f}{gu + hv + 1}. \tag{12.2}$$

Equation 12.1 can be solved for $u$ to yield

$$u = \frac{v(b - hx) + c - x}{gx - a},$$

which when substituted for $u$ in equation 12.2 gives

$$G(x, v) = \frac{d(v(b - hx) + c - x) + ev(gx - a) + f(gx - a)}{g(v(b - hx) + c - x) + hv(gx - a) + (gx - a)}$$

Thus, any of the projective warps can be seen to be separable.

### 12.2.2 Paeth-Tanaka rotation algorithm

The Paeth-Tanaka algorithm is a very clever example of a separable warp algorithm. It handles only pure image rotations, which is a special case of the projective warps, and operates in three phases – 1) scanline, 2) column, and 3) scanline. Paeth and Tanaka realized that any rotation in two dimensions can be factored into three successive shears, as illustrated in Figure 12.6.

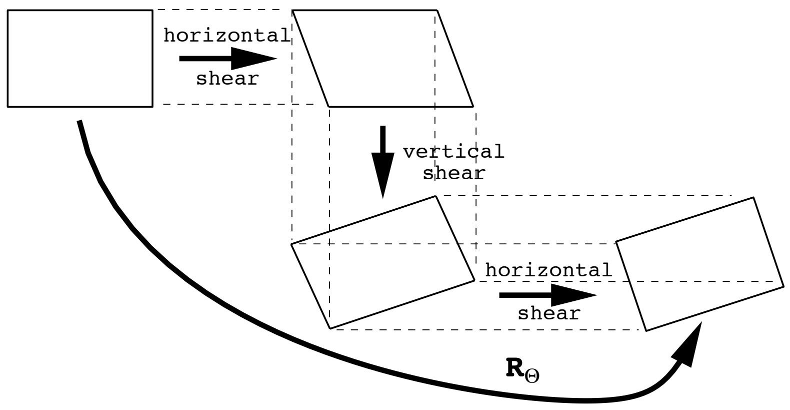


Figure 12.6: Image Rotation Implemented as a Sequence of Three Shears

The factorization of the rotation matrix used in the algorithm is

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin\theta & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix}. \tag{12.3}$$

Checking this factorization is left as an exercise to the student. While checking, you will need the trigonometric identities

$$\tan\theta/2 = \frac{\sin\theta}{1+\cos\theta},$$

and

$$\tan\theta/2 = \frac{1-\cos\theta}{\sin\theta}.$$

Note that under the factorization of Equation 12.3, the multiplication

$$R_\theta \begin{bmatrix} u \\ v \end{bmatrix}$$

becomes

$$(\begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix} (\begin{bmatrix} 1 & 0 \\ \sin\theta & 1 \end{bmatrix} (\begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}))),$$

which is a horizontal shear, followed by a vertical shear, followed by a second horizontal shear.

The beauty of this formulation is that under each of the shears, no scale is required and at each step in the algorithm, pixel motion is in one direction only. Thus, each output scanline from the first shear is just a horizontal displacement of the entire input scanline. The output from the second shear is simply a vertical displacement of each column of the intermediate result, and the output from the third shear is again a horizontal displacement of each scanline of the second intermediate result. Since the forward map does no scales and only applies translations, this allows us to use a forward mapping algorithm rather than an inverse map to compute the rotation. Further, we can apply a simple tent reconstruction filter very efficiently by simply averaging each adjacent pair of input pixels in the source into a single pixel in the destination according to a fixed ratio on each scanline, as depicted in Figure 12.7

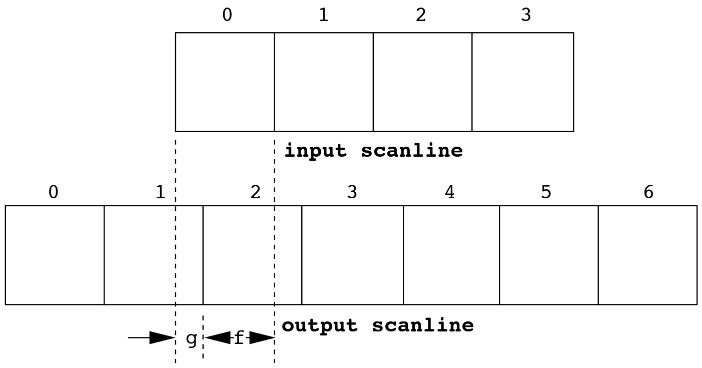
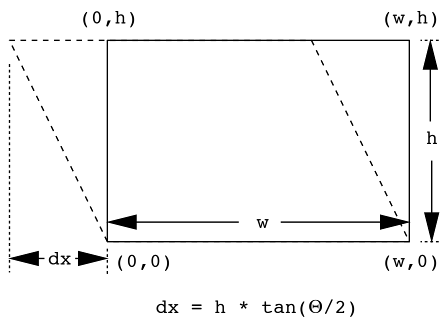


Figure 12.7: Image of Input Scanline Overlaid on Output Scanline

The entire algorithm reduces to three simple steps to compute the three successive shears:

1. Loop across each scanline, displacing each pixel on the line and weighting each adjacent pair of pixels into each output pixel,
2. Loop on each column, displacing each pixel in the column and weighting each adjacent pair of pixels into each output pixel,
3. Repeat of Step 1.

The only complication is that before each step, it is necessary to compute the output array size to store the intermediate image. This can be done quite simply, since at each step either the number of rows or number of columns stays fixed so only the extents of the other coordinates need to be determined. The geometry of this calculation is shown in Figure 12.8a, and Figure 12.8b gives a detailed algorithm for implementing step 1.



a) geometry of step 1 of the Paeth-Tanaka algorithm

```
x_off = height * tan(theta/2);    // offset of lower lefthand corner
if(x_off < 0) x_off = 0;

for(v = 0; v < height; v++){      // v = input scanline number
  x0 = -v * tan(theta/2) + x_off; // starting x, for u = 0
  f = fabs(x0 - (int)x0);         // overlap fraction
  g = 1.0 - f;

  for(j = 0; j < (int)x0; j++)    // zero output scanline up to x0
    out[j][v] = 0;

  out[j++][v] = g * in[0][v];     // tent filter input pixels into output
  for(i = 0; i < width - 1; i++)
    out[j++][v] = f * in[i][v] + g * in[i+1][v];
  out[j++][v] = f * in[i+1][v];

  for(; j < width + x_off; j++)   // zero rest of scanline
    out[j][v] = 0;
}
```

b) algorithm for step 1 of the Paeth-Tanaka algorithm

Figure 12.8: Implementation Details for Step 1 of the Paeth-Tanaka Rotation Algorithm

## 12.3 Mesh Warp Algorithm

In a mesh warp, we superimpose a regular grid over the input image, and then allow the user to move grid corners to deform the grid. The image is then warped, usually using forward bilinear interpolation, to map the input subimage in each grid square to the corresponding deformed quadrilateral in the deformed grid.

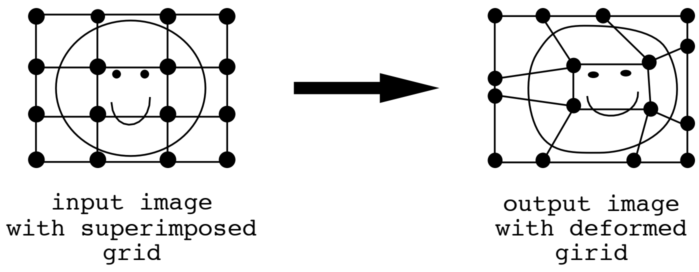


Figure 12.9: Image Deformation via a Mesh Warp

Typical constraints in a mesh warp are that

1. image corners cannot be moved,
2. grid vertices on horizontal and vertical edges may only move along the edge,
3. grid edges cannot be made to cross over each other.

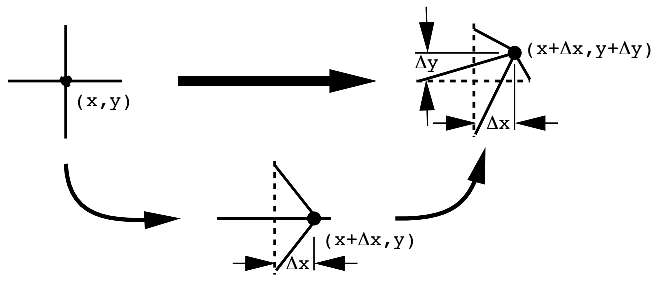
These constraints assure that input and output images are the same size and shape, and that the image will deform like a rubber sheet, without tearing or folding over itself (i.e., it keeps its original topology).

The edges of the mesh can either be straight lines or spline curves. We will look first at a mesh warp where mesh edges are straight lines.

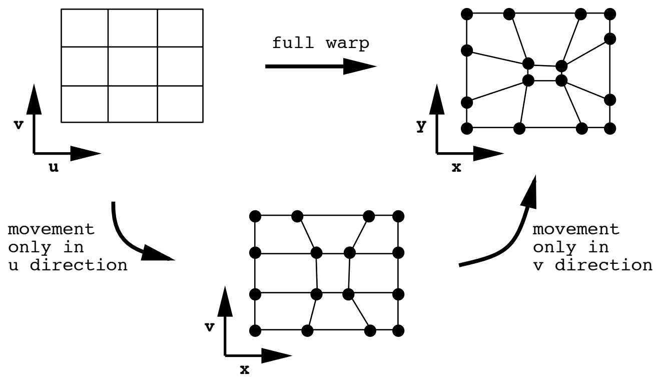
### 12.3.1 Separable mesh warp

The mesh warp can be implemented easily using a two-pass separable scanline algorithm. The idea is to treat each vertex perturbation as a translation in the horizontal direction, followed by a translation in the vertical direction. This leads quite naturally into a decomposition of the warp into a set of horizontal movements followed by a set of vertical movements as diagrammed in Figure 12.10a, which leads to the full mesh warp shown in Figure 12.10b.

The algorithm, based on this idea, goes as follows:



a) displacement of a single mesh point in two steps



b) full mesh warp in two steps

Figure 12.10: Mesh Warp Implementation Using Horizontal and Vertical Displacements

1. Apply horizontal movement to each mesh point in the $u$ direction to give $(x, v)$ coordinates for each mesh point.

2. For each scanline in the intermediate output image, note the intersection of all grid lines with this scanline as shown in Figure 12.11, and build an array of these intersection points. If there are $M$ vertical grid lines, and $N$ scan lines, then the array will be $N \times M$ in size, as there will be $M$ entries for each of $N$ scanlines.

3. Resample across each scanline to the full image resolution, using forward mapping, accounting for overlap by using box-filter averaging where the scanline is being minified (compressed) and filling in holes using tent-filter resampling where the scanline is being magnified (stretched).

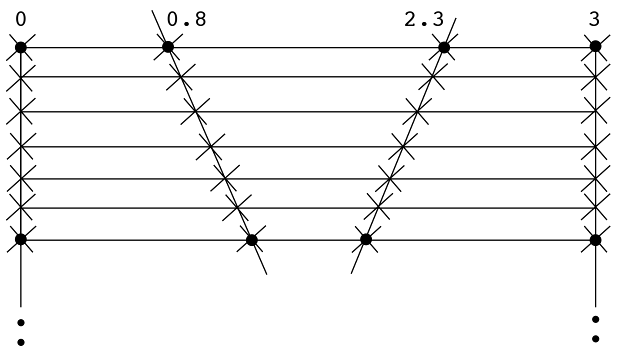


Figure 12.11: Crossings of Mesh Lines with Scanlines

4. Repeat this process from the intermediate image to the final output image, but doing the process over image columns rather than scanlines, and doing movements in the vertical direction to change $(x, v)$ coordinates into $(x, y)$ coordinates.

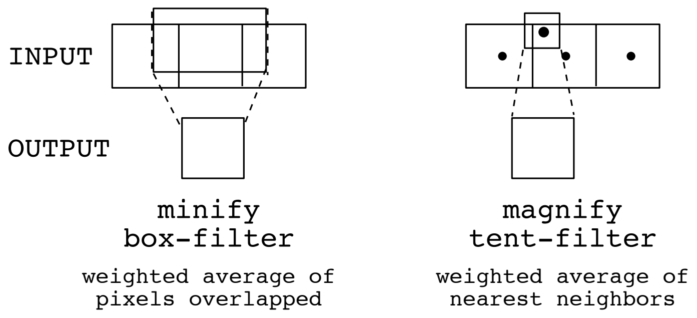
### 12.3.2 Mesh Warp Using Spline Interpolation

The warp algorithm as described can easily be extended to warp an image via a mesh with smoothly curved edges. The only extension needed here is to interpolate between mesh control points using a higher order interpolating curve. A typical choice is a cubic interpolation polynomial. The difference between simple linear interpolation and cubic interpolation is illustrated in Figure 12.13.

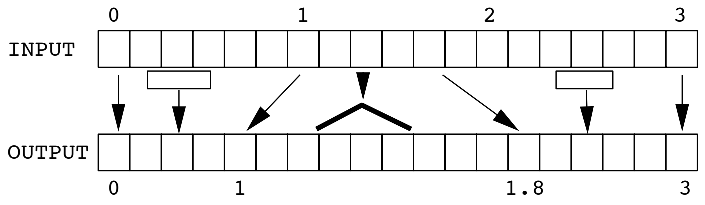
A cubic interpolating polynomial is constructed by fitting curves described by cubic polynomials between the control points in such a way that they join together smoothly to create a continuous curve across all of the points. Choosing to use cubic curve segments between mesh control points allows us to organize the calculation in a straightforward way. In reading the following description, please refer to Figure 12.14, which shows how such an interpolating curve would be constructed down a column (in the $v$ direction) in the input image to govern horizontal displacements (in the $x$ direction) along a scanline in the output image. For each segment $i$ of the curve (e.g. segment 1 in the figure) define a distance parameter

$$t_i = \frac{v - v_i}{v_{i+1} - v_i} \tag{12.4}$$

which varies from 0 to 1 as $v$ varies from $v_i$ to $v_{i+1}$. Then the displacement



a) filters for magnification and minification



b) different filters applied along the scanline

Figure 12.12: Filtering Input Image into Output Image Along a Scanline

of the curve in the $x$ direction for segment $i$ is given by

$$X_i(t_i) = a_i t_i^3 + b_i t_i^2 + c_i t_i + d_i, \tag{12.5}$$

i.e. a cubic curve in $t_i$. The coefficients $a_i$, $b_i$, $c_i$, and $d_i$ are determined by imposing constraints on the curve segment.

Certainly we want the curve segment to pass through the control points at each of its two ends. Letting $x_i$ and $x_{i+1}$ be the control point horizontal coordinates for $v_i$ and $v_{i+1}$ , we have

$$X_i(0) = x_i \text{ and } X_i(1) = x_{i+1}.$$

Also, we would like the curve segment to join smoothly with its neighbors to the left and right. This is often enforced by requiring that the slopes and curvatures must match where the segments join. In other words, their first and second derivatives must match. This gives

$$X'_{i-1}(1) = X'_i(0) \text{ and } X''_{i-1}(1) = X''_i(0).$$

These constraints, plus some additional ones to handle conditions on the extreme left and right ends of the entire curve serve to completely determine the coefficients $a_i$, $b_i$, $c_i$ and $d_i$ for each curve segment.

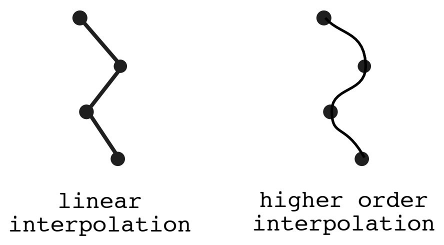


Figure 12.13: Linear Interpolation vs. Cubic Polynomial Interpolation

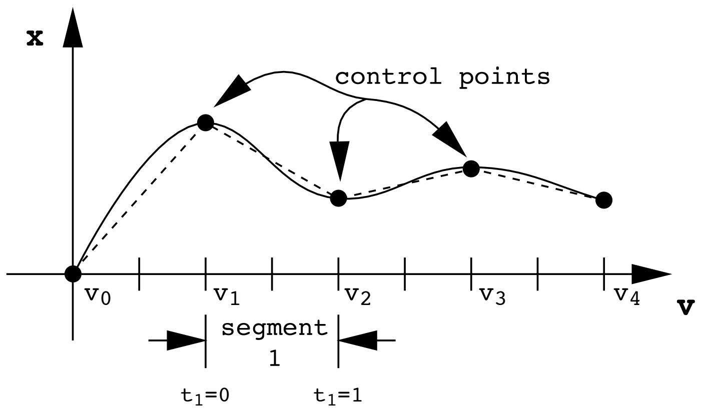


Figure 12.14: Cubic Spline Down Input Column Giving Output Horizontal Displacements

Once these coefficients are determined, Equations 12.4 and 12.5 can be used to find the intersections of the curves with the scanlines and columns. Equations 12.4 and 12.5 are set up to determine horizontal displacement $x$ in the output image as a function of vertical displacement $v$ in the input image. They can be used for pass-one of the algorithm to determine where each curve crosses the scan lines of the image. From here, it is an easy exercise to construct similar functions to be used for pass-two, which give the vertical displacements $y$ in the output image as a function of horizontal displacements $u$ in the input image.

# Chapter 13

# Morphing

A *morph* from one image to another involves 1) warping both of the images to some intermediate shape where they can be superimposed upon each other, and 2) blending the two images together to produce a third image. An example is given in Figure 13.1, which shows a step in a morph from a brick into a ball. First, we find an intermediate shape that we can deform both into, warp both images to this intermediate shape, and then blend the intermediate images to produce an image that is a mix of the two originals.

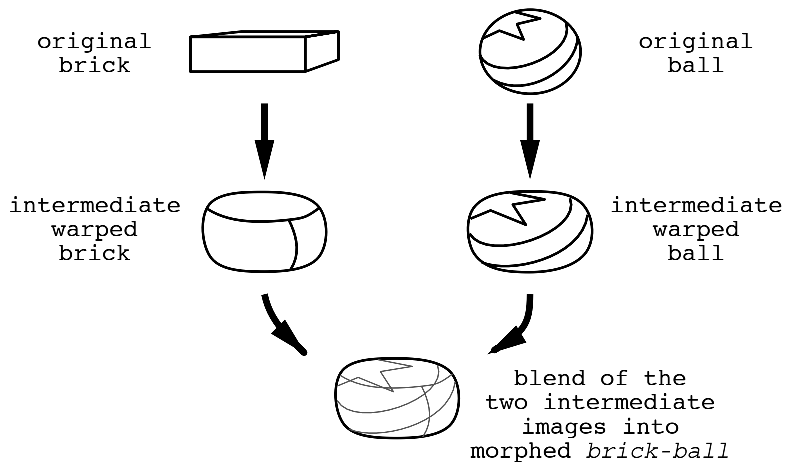

original brick
original ball
intermediate warped brick
intermediate warped ball
blend of the two intermediate images into morphed *brick-ball*

Figure 13.1: A Step in a Morph of a Brick into a Ball

Five steps in the complete brick-to-ball morph sequence are shown in Figure 13.2. The complete sequence consists of a series of intermediate warped shapes that go from one of the original shapes to the other, and the blend is controlled by giving proportionally more strength to the original image that is closest to the deformed shape. If this is done with some artistry, when the sequence is played back it can look like a smooth transition from one shape and coloring to the other.

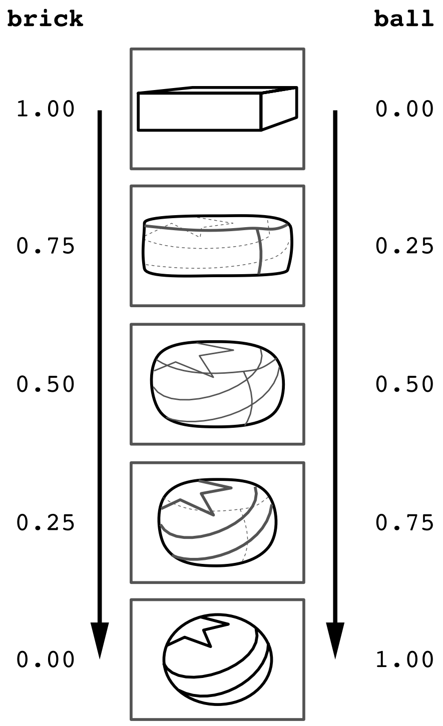


Figure 13.2: Steps in Morph Sequence from Brick to Ball

If the two objects being morphed are also moving, then each warp in the sequence must be done between corresponding frames in the original motion sequence. For this reason, it is important that a morph tool for animation or video gives the animator good control over the warp between the images while allowing for the exploitation of continuity from frame to frame – to minimize the amount of hand work that must be done for each frame.

## 13.1 Morphing Algorithms

We will look at three schemes for doing morphing; a scan line approach, a triangular area fill approach, and a feature based approach.

### 13.1.1 A Scanline Morphing Algorithm

The scanline algorithm extends the separable mesh warping algorithm that we studied in the last chapter. The idea is to allow the user to deform a mesh over each of the two images, so that the mesh approximately fits over the image features that are to be warped and so that corresponding mesh elements enclose areas of the two images that are to be warped and blended into each other. Figure 13.3 shows how a pair of meshes might be set up for the brick-to-ball morph, although generally this will require a much finer grid and careful line placement than that shown in the figure.

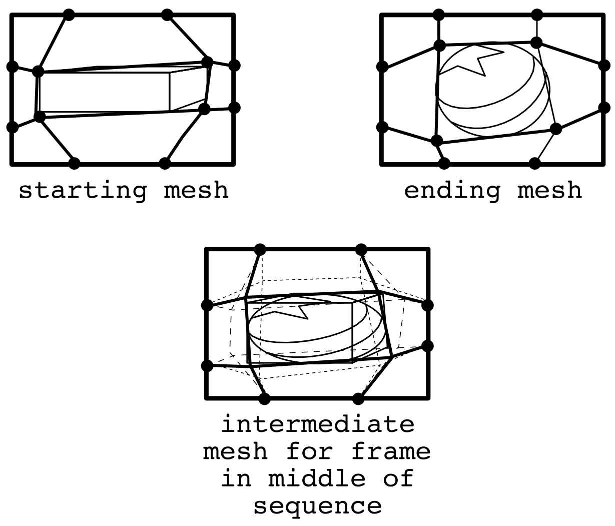


Figure 13.3: Rectangular Meshes Morphing a Brick into a Ball

Now, each image will be warped to an intermediate form that can be derived by interpolating corresponding vertices in the mesh over image 1 into the vertices in the mesh over image 2. This is very simple to compute – for the frame at fractional position $f$ along the sequence ($f = 0$ at the starting frame, and $f = 1$ at the ending frame), simply set each vertex $P_{i,f}$ in the mesh for frame $f$ to

$$P_{i,f} = fP_{i,1} + (1 - f)P_{i,0}.$$

This process of mesh interpolation is also shown in Figure 13.3, which shows how the starting and ending meshes are interpolated to form the mesh in the middle of the sequence.

Then, each image is warped to this intermediate mesh using the separable scan line mesh warp algorithm, and the images are blended, also using $f$ as the blending factor. The interested student is referred to the text by Wolberg for complete details of the algorithm.

### 13.1.2 Triangle Mesh Warp Morph

The rectangular mesh technique is by no means the only mesh-oriented technique that can be used to do a morph. For example, a triangle mesh warp algorithm was developed by Darrin Butts for the *Midas* morphing program. It gives the user free control over the number and placement of the vertices of a polygonal outline in the initial image, and then requires the user to triangulate the region enclosed by the polygon by connecting groups of three vertices with edges. It then copies the outline to the final image and allows the user to reposition the vertices, thus warping and repositioning the triangles. The program creates intermediate meshes by linear interpolation based on frame number, just as with the rectangular mesh warp. Triangles in the original two images are warped into the triangles in the intermediate images using an approach very similar to bilinear interpolation for rectangular grid elements. The triangular mesh warp process used by *Midas* is illustrated in Figure 13.4.

### 13.1.3 Feature Based Morph

A feature based approach to defining a morph was by Thad Beier of SGI and Shawn Neely of PDI (SIGGRAPH '92). This technique was used for the ending segment of the Michael Jackson "Black and White" music video, which features a sequence of morphs of dancers heads into each other. This technique works by having the user draw pairs of lines, one in each of the two images that are to be morphed, that locate and identify significant features of the two images that should correspond to each other in the morphed image. The idea is that corresponding lines are both to map to an "averaged" line in the intermediate image. This is implemented mathematically by using these lines to define an *influence function* which spreads a kind of gravitational influence of each line out across the image. At each pixel in the image, the sum of the influence functions of all of the feature lines determines the value of a global "field", which is used to warp the image into the intermediate image. The basic idea of this algorithm is depicted in Figure 13.5.

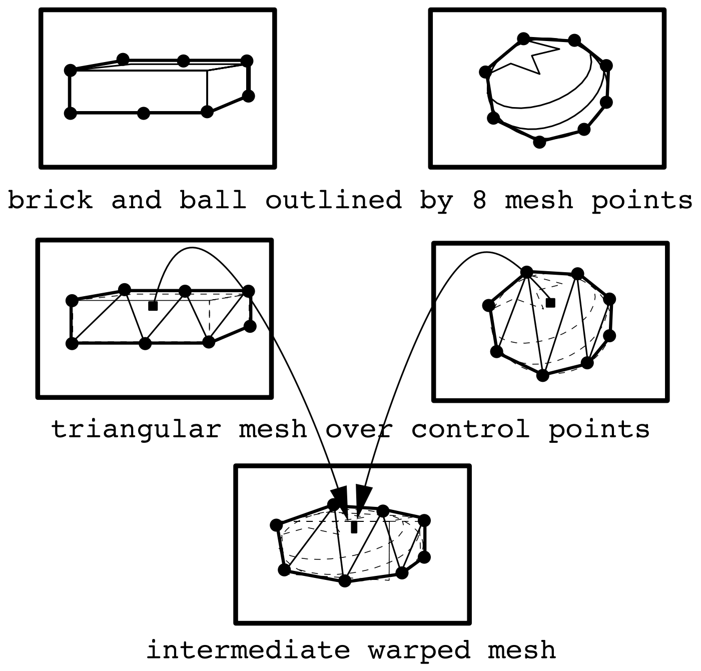

brick and ball outlined by 8 mesh points
triangular mesh over control points
intermediate warped mesh

Figure 13.4: Triangular Meshes Morphing a Brick into a Ball

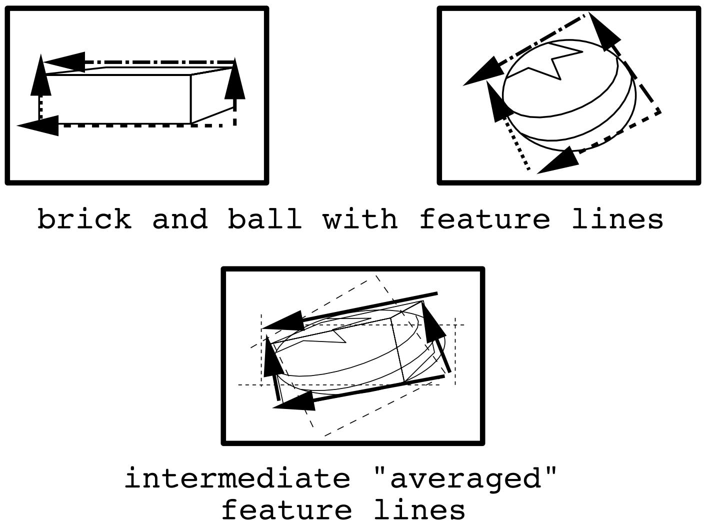

brick and ball with feature lines
intermediate "averaged" feature lines

Figure 13.5: Feature Lines used to Morph a Brick into a Ball

# Chapter 14

# Frequency Analysis of Images

## 14.1 Sampling, Aliasing and Reconstruction

We have already informally looked at the problem of doing any sort of image warp (change in geometry) when the image is a discretely sampled raster representation. When the image is warped, changes in size and orientation make the map from input pixels to output pixels complicated to determine, and in general not 1-to-1, i.e. there is not usually precisely one input pixel corresponding to each output pixel.

This problem is fundamental to digital processing of any sort of information such as images, sound, simulation data, etc., that is sampled as part of the information capture process. We can view such sampling abstractly as the selection of a sequence of evenly spaced values from a continuous signal or function, as depicted in the graphs of Figure 14.1. When we take any sort of continuous signal and store a representation of it as a sequence of samples, we have made some very fundamental changes in the information. Obviously, we have thrown away a lot of potential detail. It is this potential for loss of important information that leads to problems later. Sampling leads to two general classes of problem, reconstruction errors and aliasing.

We can recognize aliasing as the introduction of pattern artifacts into the image or sound that we are trying to reproduce. Moire patterns (that are often made for artistic effect) are the result of the same kind of error that causes aliasing – basically that we are sampling at a rate that is below the level of detail of what we are trying to capture. Such aliasing artifacts are symptomatic of the sampling process itself, and cannot be avoided unless

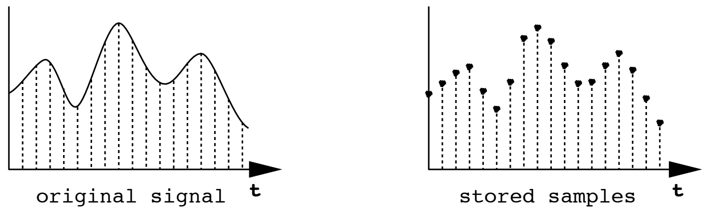


Figure 14.1: Sampling a Continuous Function

care is taken to do sampling in a way that is appropriate to the data being sampled. A demonstration of Moire patterns created by radial lines drawn on a CRT raster can be seen by running the program `moire` that can be found in the directory `/usr/local/misc/courses/viza654/examples/`.

Another problem with sampled data is that in order to reproduce the original input from the samples, we must somehow reconstruct the input using only the samples as our information source. For example, in a digital image we reconstruct an original continuous image by spreading each sample value across the area of a rectangular pixel on the screen. The artifacts due to the reconstruction process are known as reconstruction artifacts, and for digital images result in such errors as staircasing or *jaggies*, as shown in Figure 14.2.

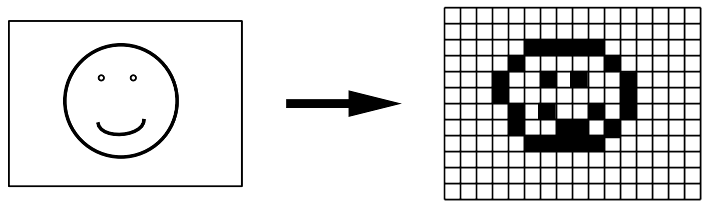


Figure 14.2: Jaggies Due to Rectangular Pixel Reconstruction

The questions that we need to address if we are to be able to produce high quality warped images are

1. How should we do sampling to ensure that we have retained enough data to avoid aliasing artifacts?

2. How should we do reconstruction, given a set of samples, to avoid reconstruction artifacts?

Our goal in attempting to answer both of these questions will be to see how successful we can be in avoiding – or at least minimizing – both aliasing and reconstruction artifacts.

## 14.2 Frequency Analysis

Before we can answer either of these questions, we need to develop an understanding of how the smoothness of a data set relates to how finely it must be sampled, and how much information is contained in a set of samples. Frequency analysis will give us the tools we need to do this.

We are used to thinking of sound as having a frequency spectrum, and can readily identify the low frequency tuba from the high frequency piccolo. The notion of frequency is normally associated with a time varying signal (like sound), but finds important use in both developing tools to operate on images and in providing a framework for the investigation of image qualities. In fact, it provides the foundation of theory supporting the design and refinement of most image processing techniques.

### 14.2.1 Fourier series

What do we mean by frequency analysis? The notion here is that any finite duration or periodic "signal", can be represented as an infinite weighted sum of sines and/or cosines. This representation is known as the *Fourier Series* representation. Given

$$f(t) = g(t) = \begin{cases} 0,\ t > T \\ g(t),\ 0 \leq t \leq T \\ 0,\ t < 0, \end{cases}$$

then, on the interval $(0, T)$ we have

$$f(t) = 1/2\, C_0 + \sum_{n=1}^{\infty} C_n \cos(\frac{2\pi n}{T} t - \phi_n) \tag{14.1}$$

where the amplitude coefficients $C_n$ and the phase angles $\phi_n$ are determined by first computing auxiliary sine and cosine coefficients

$$A_n = \frac{2}{T}\int_0^T f(t) \cos\frac{2\pi n}{T} t dt, \tag{14.2}$$

$$B_n = \frac{2}{T}\int_0^T f(t) \sin\frac{2\pi n}{T} t dt, \tag{14.3}$$

and then computing

$$C_n = \sqrt{A_n^2 + B_n^2}, \tag{14.4}$$

and

$$\phi_n = \tan^{-1}\frac{B_n}{A_n}. \tag{14.5}$$

Figure 14.3 shows how the Fourier Series representation can be used to represent a square wave. It shows how the coefficients for the series are calculated, and shows how the series gradually converges to the shape of the original wave as more and more terms are added to the sum.

What does this mean intuitively?

$$f(t) = 1/2\, C_0 + \sum_{n=1}^{\infty} C_n \cos(\frac{2\pi n}{T} t - \phi_n)$$

On the range from 0 to $T$, our function $f(t)$ can be represented as the sum of a constant $1/2\, C_0$, which is simply the average value of $f(t)$, and a sum of sine waves (shifted to the left by $\pi/2$ which changes the sines to cosines), each weighted by the weights $C_n$ and shifted to the right by the distance $\frac{T}{2\pi n}\phi_n$, and with frequencies that are integer multiples of the *fundamental* frequency $2\pi/T$. The integer multiples of the fundamental frequency (shown in Figure 14.4) are known as its *harmonics*.

To determine the cosine weight for a particular value of $n$: 1) multiply the function $f(t)$ by the cosine, 2) find the area under the resulting curve, 3) divide by 1/2 the fundamental interval $T$. In other words,

$$A_n = \frac{2}{T} \int_0^T f(t) \cos \frac{2\pi n}{T} t dt.$$

A similar process is used to find the sine weights, yielding the equation

$$B_n = \frac{2}{T} \int_0^T f(t) \sin \frac{2\pi n}{T} t dt.$$

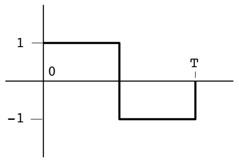
If $f(t)$ is our signal, we can now think of the signal in terms of its frequency content. The combined sine and cosine coefficient

$$C_n = \sqrt{A_n^2 + B_n^2}$$

gives the total weight of the $n$th harmonic in determining the shape of the signal $f(t)$. In other words, the weights $C_n$ tell you how important the $n$th harmonic is in determining the overall shape of the signal or function.

### 14.2.2 Fourier transform

A more general transformation from a time varying to a frequency-based or *frequency domain* signal is given by the *Fourier Transform.* Unlike the Fourier series, the Fourier transform does not require the signal to be of



$$A_n = \frac{2}{T}\int_0^T f(t)\cos\frac{2\pi n}{T}tdt$$
$$= \frac{2}{T}\left[\int_0^{T/2}\cos\frac{2\pi n}{T}tdt - \int_{T/2}^{T}\cos\frac{2\pi n}{T}tdt\right]$$
$$= \frac{1}{\pi n}\left[\sin\frac{2\pi n}{T}t\Big|_0^{T/2} - \sin\frac{2\pi n}{T}t\Big|_{T/2}^{T}\right] = 0,$$

i.e. all cosine coefficients are 0.

$$B_n = \frac{2}{T}\int_0^T f(t)\sin\frac{2\pi n}{T}tdt$$
$$= \frac{2}{T}\left[\int_0^{T/2}\sin\frac{2\pi n}{T}tdt - \int_{T/2}^{T}\sin\frac{2\pi n}{T}tdt\right]$$
$$= \frac{1}{\pi n}\left[-\cos\frac{2\pi n}{T}t\Big|_0^{T/2} + \cos\frac{2\pi n}{T}t\Big|_{T/2}^{T}\right]$$
$$= \frac{2(1-\cos\pi n)}{\pi n},$$

i.e. all sine coefficients are 0 for $n$ even, and $\frac{4}{\pi n}$ for $n$ odd. Thus

$$f(t) = \sum_{n\ \mathrm{odd}}\frac{4}{\pi n}\sin\frac{2\pi n}{T}t = \sum_{n\ \mathrm{odd}}\frac{4}{\pi n}\cos(\frac{2\pi n}{T}t - \pi/2).$$

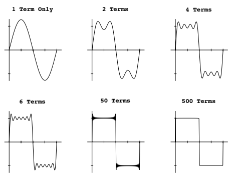


Figure 14.3: Square Wave Example of Fourier Series

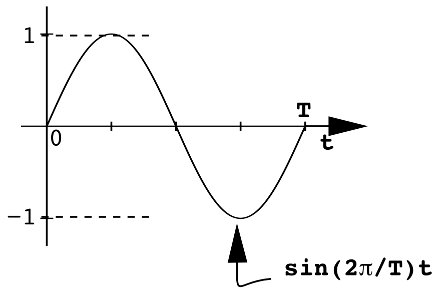


Figure 14.4: Fundamental Sine Wave of Period $T$

finite duration or to be periodic, and it produces a continuous frequency spectrum, rather than discrete harmonic weights. It is given by

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt, \tag{14.6}$$

and its inverse is

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega t}d\omega, \tag{14.7}$$

where $t$ is the measure used along the horizontal axis, $\omega$ is frequency measured in radians per unit of the horizontal axis, and $i = \sqrt{-1}$, and $e^{i\theta}$ is the complex exponential. The complex exponential is defined by its complex number equivalent in trigonometric form:

$$e^{i\theta} = \cos\theta + i\sin\theta.$$

The relationship between the Fourier Transform and the Fourier Series may seem obscure, but due to the equivalence between the complex exponential and the trigonometric form, we could rewrite Equation 14.6 as

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\cos\omega t dt - i\int_{-\infty}^{\infty} f(t)\sin\omega t dt. \tag{14.8}$$

In this form, the Fourier Transform bears a close resemblance to the Fourier series (compare Equation 14.8 with Equations 14.2 and 14.3 for the Fourier series coefficients). Anyway, the point is that we are back to integrals of products of our function and sines and cosines. The principal differences between the Fourier Series and Fourier Transform representations are that

1. $F(\omega)$ is continuous, rather than a collection of discrete harmonic weights, and

2. $F(\omega)$ is a complex function, rather than a real function.

If $F(\omega)$ is the Fourier Transform of function $f(t)$, we say that $|F(\omega)|$ (i.e. the magnitude of the Fourier Transform) is its *Fourier Spectrum.* Figure 14.5 shows what such a spectrum might look like for a typical input signal. Like the coefficients of the Fourier Series, the Fourier Spectrum gives a measure of how much each frequency contributes to the overall shape of the signal or function.

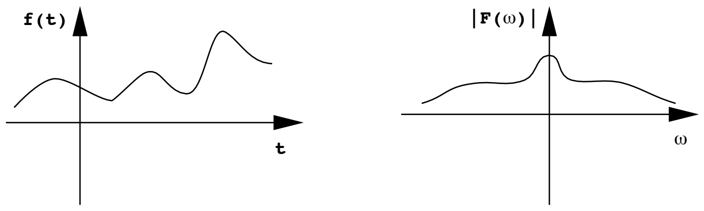
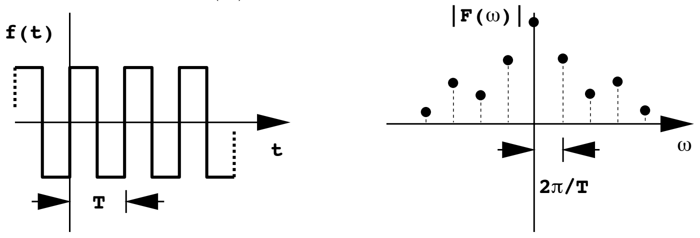


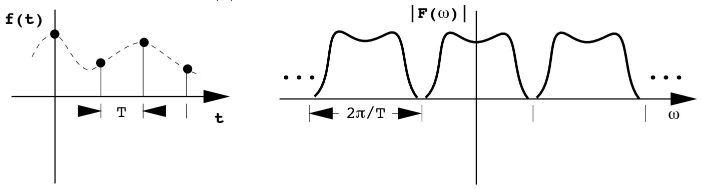
Figure 14.5: $|F(\omega)|$ is the Fourier Spectrum

Fourier transforms of real signals obey the following laws:

- If $f(t)$ is a real signal, $|F(\omega)|$ is symmetric about 0.
- $f(t)$ and $F(\omega)$ are two complete representations of the same signal and are completely recoverable from each other.
- If $f(t)$ is periodic then $F(\omega)$ is discrete.



- If $f(t)$ is discrete (sampled), then $F(\omega)$ is periodic with period equal to the sampling rate of $f(t)$.



- If $f(t)$ is both discrete and periodic then $F(\omega)$ is also both discrete and periodic.

### 14.2.3 Discrete Fourier transform

There is a discrete version of the Fourier Transform based on the last law, that is especially useful for dealing with images or any kind of sampled data. When our function $f(t)$ is both discrete and periodic, then its Fourier Transform $F(\omega)$ is also. It is clear that any discrete periodic function requires only a fixed number of samples or values to represent it. If the function is periodic, the discrete values that describe it repeat over and over, and there is no need to record the repetitions.

The same is true for the function's Fourier Transform. Take a function $f(t)$ sampled on a unit interval and periodic with period $N$, then it is described by $N$ discrete sample values, and its Fourier transform is also described by $N$ discrete sample values. Also, if there is a finite number of samples, then the Fourier Transform is given by a finite sum rather than an integral. This form of the Fourier Transform is called the *Discrete Fourier Transform* (DFT) and is given by

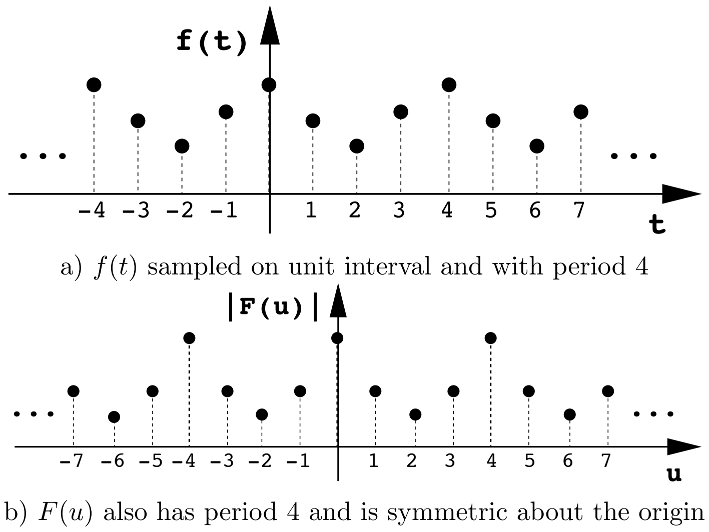
$$F(u) = 1/N \sum_{t=0}^{N-1} f(t) e^{-i2\pi ut/N};\ u = 0, 1, 2, ..., N-1, \tag{14.9}$$

with inverse

$$f(t) = \sum_{u=0}^{N-1} F(u) e^{i2\pi ut/N};\ t = 0, 1, 2, ..., N-1. \tag{14.10}$$

Note that in these equations $t$ is still used as the time or distance measure, but it is a sequence of integers, and $u$ is an integer frequency measure in cycles per unit of the total span $N$, unlike $\omega$ that was measured in radians per unit.

Figure 14.6 depicts a sampled periodic signal and its DFT. Note that the DFT itself is sampled and periodic with period $N$, but more – it is symmetric about the origin, $u = 0$. This symmetricity is an attribute of the Fourier Transform of any real valued signal. Since $F(u)$ is both periodic and symmetric, we need to store only 1/2 of the $N$ samples to completely describe $F(u)$, but remember that $F(u)$ is complex and so carries both amplitude and phase information at each sample. Thus we need two numbers to describe each sample so that we still have to store a total of $N$ numbers – we haven't gained any "compression" simply by doing the DFT. However, this symmetry is very important in giving us a way to compute the DFT efficiently, as we will see later when we discuss the fast algorithm for computing the DFT, known as the Fast Fourier Transform.



a) $f(t)$ sampled on unit interval and with period 4

b) $F(u)$ also has period 4 and is symmetric about the origin

Figure 14.6: A Discrete Periodic Function and its Fourier Transform

### 14.2.4 Discrete Fourier Transform of an image

This discrete form of the Fourier Transform permits us to transform a digital image. Given an image sampled with $N$ samples per scanline, and $M$ scanlines, and letting $x$ represent horizontal distance, $y$ vertical distance (i.e. column and row numbers) and $u$ and $v$ represent frequency in cycles per scanline or cycles per column, the two-dimensional Discrete Fourier Transform is given by

$$F(u,v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-i2\pi(\frac{ux}{N} + \frac{vy}{M})},$$

with inverse transform

$$f(x,y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{i2\pi(\frac{ux}{N} + \frac{vy}{M})},$$

Here $x$, $y$, $u$, and $v$ are integers, with $0 \le x, u < N$, and $0 \le y, v < M$.

Because of the law $x^{(a+b)} = x^a x^b$ governing sums of exponents, these equations can be factored into a handy computational form that allows computation of the DFT in two parts – first by computing the DFT of each scanline, and then by computing the DFT of each column of this intermediate form. This factorization gives us

$$F(u,v) = \frac{1}{MN} \sum_{y=0}^{M-1} \left[ \sum_{x=0}^{N-1} f(x,y) e^{-i2\pi ux/N} \right] e^{-i2\pi vy/M}, \tag{14.11}$$

and

$$f(x,y) = \sum_{v=0}^{M-1} \left[ \sum_{u=0}^{N-1} F(u,v) e^{i2\pi ux/N} \right] e^{i2\pi vy/M}, \tag{14.12}$$

Now, we have waded through a lot of mathematics, and you might be asking yourself at this point, what does the Discrete Fourier Transform buy us? In a word, it gives us a new way of looking at our image data. Specifically,

1. it gives us a different space within which to study images, known as the *frequency domain.*

2. $|F(u,v)|$ gives the frequency spectrum, or spectral content of the image, i.e. how much "information" in the image is concentrated at various frequencies. This can tip us off as to what periodicities exist in the image, and what ranges of frequencies are most represented?

3. The frequency content of an image is strongly related to what kind of features might exist in the image. In general, hard edges and high contrast in an image are equivalent to having strong high frequency content in the Fourier Transform. Soft edges and low contrast are equivalent to having strong low frequency content.

4. Certain filtering operations become trivial to implement in the frequency domain. The filtering process can be seen as

$$f(t) \overset{\substack{\text{fourier}\\\text{transform}}}{\Longrightarrow} F(\omega) \overset{\substack{\text{apply}\\\text{filter}}}{\longrightarrow} F'(\omega) \overset{\substack{\text{inverse}\\\text{transform}}}{\Longrightarrow} f'(t)$$

5. The Fourier transform holds the key to understanding and coping with sampling issues.

### 14.2.5 A demonstration of frequency based image construction

The demonstration program `frequency` in the `examples` directory under the course directory allows the user to construct images by specifying the amplitude and phase of the harmonic frequencies along scanlines and down columns. The program sums all of the harmonics, using one such sum for all of the scanlines, and another such sum for all of the columns. The interface to the program is simply a menu of three buttons. The *open* button is used to select an input data file describing the image size and harmonic content. the *save* button allows the user to save an image to a file once it is created, and the *quit* button is used to exit the program.

Input to the program is via a data file organized as shown in Figure 14.7. The first line of the file contains the image width and height, which must be integer numbers between 1 and 1,024. The second line contains a number between 0.0 and 1.0 which gives the average image intensity. The third line contains triples describing scanline (horizontal) harmonics and the fourth line has an identical format but describes column harmonics. Each harmonic triple consists of a positive integer harmonic number between 1 and one-half the scanline (or column) width, a number between 0.0 and 1.0 giving the amplitude of the harmonic, and a number between 0 and 360 giving the phase angle of the harmonic in degrees. The two lines describing harmonics each start with an interpolation flag that must be either 0 or 1. If the flag is 0, each specified harmonic is taken as an explicit discrete value. If the flag is 1, harmonics are taken in pairs, with amplitude and phase of the harmonics lying between these pairs linearly interpolated between the two harmonics in the pair.

| Data | | | | | Description |
|---|---|---|---|---|---|
| width | height | | | | image size |
| midlevel | | | | | average grey level |
| interpolation-flag | harmonic | amplitude | phase | ... | scanline harmonics |
| interpolation-flag | harmonic | amplitude | phase | ... | column harmonics |

Figure 14.7: `frequency` Program Input Data File Format

The image is constructed by taking products across the scanline and column harmonic sums. Given the arrays `S` and `C` contining the computed sums of scanline and column harmonics, the pixel in image `I` at location (`row`, `col`) is computed by

```
I[row][col] = S[col] * C[row];
```

Figure 14.8 shows how this computation works. The pixel with the • in the output image on the left is the product of the scanline and column array cells containing •'s. Likewise, the pixel with the # is the product of the scanline and column cells containing #'s.

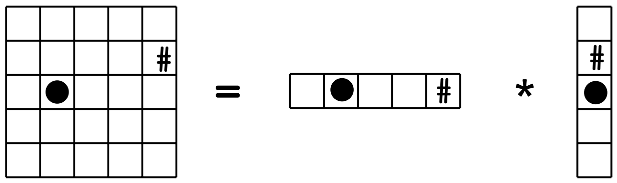


Figure 14.8: Row-Column Product used by Demonstration Program `frequency`

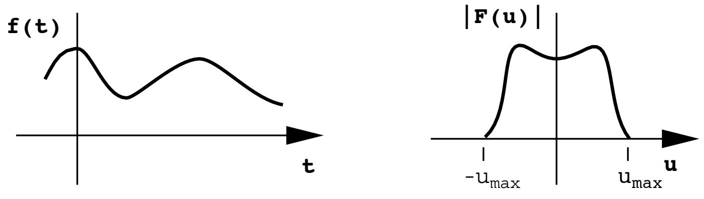
This is not the same as the fully general approach to building up an image, given by the Fourier Transform, since unlike in a two-dimensional Fourier Transform, each row has the same spectrum, and each column has the same spectrum. Nevertheless, one gets the idea of how patterns can be built up, and how a Fourier Transform can represent an image.

## 14.3 The Sampling Theorem

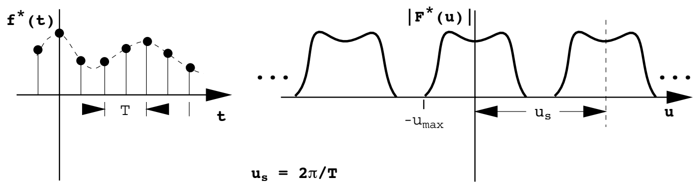
The Sampling theorem gives the following very surprising result. It states that if a signal is sampled at a sampling rate greater than twice the highest frequency contained in the signal, then it is possible to perfectly reproduce the signal from the samples. This is precisely what we want to be able to do with images – we would like our pixels (samples) to provide enough information so that we can exactly reproduce the original unsampled scene (or image). The reasoning in the theorem is as follows.

Assume that we start with a signal $f(t)$ that is *band-limited*, i.e. it has a maximum frequency $u_{\max}$ beyond which its Fourier Spectrum $|F(u)|$ is zero. Such a band limited signal and its spectrum are shown in Figure 14.9a. Now, sample that signal at some sampling frequency $u_s$ to form a new discrete signal $f^*(t)$ and take its Fourier Transform. It turns out that the Fourier Transform $F^*(u)$ of the sampled signal $f^*(t)$ is the mathematical sum of an infinite number of copies of the Fourier Transform $F(u)$ of the original signal $f(t)$, with each copy of $F(u)$ shifted from its neighboring copies by exactly the sampling frequency, as shown in Figure 14.9b. Now, since $F(u)$ is band limited, the copies of $F(u)$ in the sampled transform

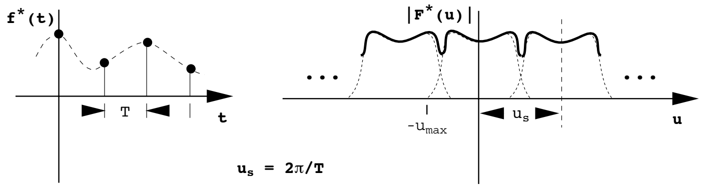
$F^*(u)$ will not overlap as long as the sampling frequency, $u_s$ is greater than twice the highest frequency in $F(u)$. If the sampling frequency is less than this value, as depicted in Figure 14.9c, the lobes overlap and when they are summed, the shape of the Fourier Transform is distorted in the region of overlap. You can see from this figure that high frequencies get shifted into and added together with low frequencies. This is the source of the aliasing phenomenon – because of sampling, artifacts get introduced into the sampled data that cannot be removed – they come from the sampling process itself.



a) signal and its Fourier spectrum



b) well sampled signal and its Fourier spectrum



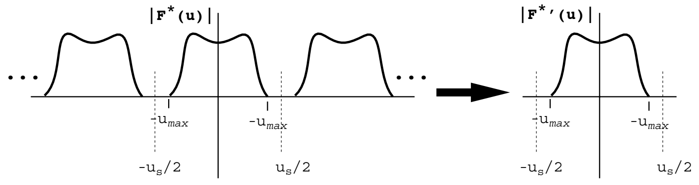
c) undersampled signal and its Fourier spectrum

Figure 14.9: Fourier Spectra of Continuous and Sampled Band-limited Signals

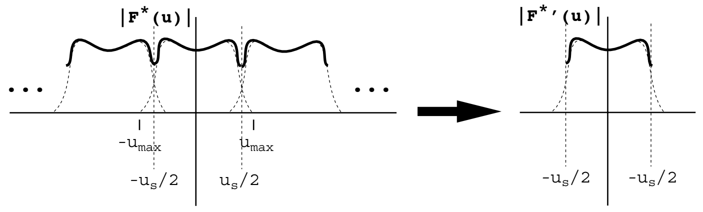
The reasoning in the sampling theorem continues – if we truncate the Fourier Transform of a sampled signal at exactly $u_s$, as shown in Figure 14.10a, to obtain the filtered Fourier Transform $F^{*'}(u)$, then we have back the Fourier Transform $F(u)$ of the original signal! In other words,

$$F^{*'}(u) = F(u), \ \textbf{if} \ u_s/2 > u_{\max}.$$

All we have to do is to take the inverse Fourier Transform to perfectly recover the full original signal from our sampled data. However, if we have not sampled fast enough so that $u_s/2$ is larger than $u_{\max}$, as is the case in Figure 14.10b, the resulting Fourier Transform $F^{*'}(u)$ will be misformed, and our reconstructed signal will have aliasing artifacts no matter what we do.



a) filtering to remove shifted copies of $F(u)$



b) deformation of result when sampling frequency is too low

Figure 14.10: Filtering in Frequency to Recover Fourier Transform of Original Signal

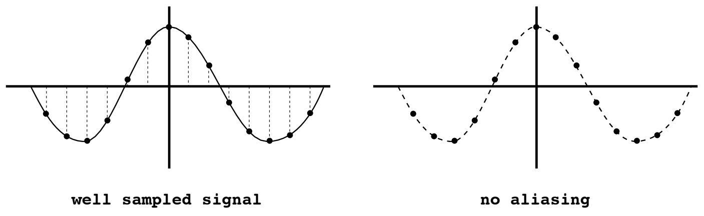
Here is the moral again:

> If you want to sample sound, image data, anything at all, you had better sample at more than twice the highest frequency in the original signal, otherwise you will have aliasing artifacts in the result **no matter what you do later**.

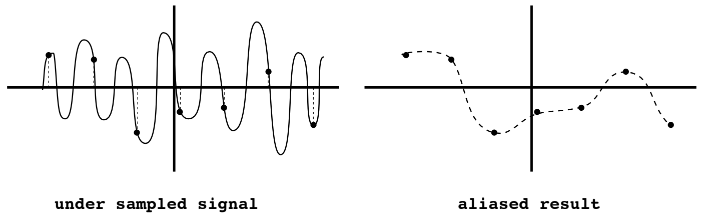
Figure 14.11 shows this result in simple graphical form.

## 14.4 Ideal Low-Pass Filter

In the chapter on removing aliasing and reconstruction artifacts, we stated that the ideal low-pass filter was the sinc function. The secret to the derivation of this result is the rather interesting fact that convolution in the spatial domain is equivalent to multiplication in the frequency domain. It is left as



a) plenty of samples per period



b) not enough samples per period

Figure 14.11: Aliasing Due to Undersampling

an exercise for the student to prove this to yourself by taking the Fourier Transform of the convolution of two functions.

If we denote the convolution operation by the operator $*$, and the pairing of a function with its fourier transform by the operator $\Leftrightarrow$, this result can be expressed algebraically by

$$h(t) * f(t) \Longleftrightarrow H(u)F(u),$$

where $f(t)$ is the function, $F(u)$ its Fourier transform, $h(t)$ is the convolution kernel and $H(u)$ its Fourier transform. The law

$$h(t)f(t) \Longleftrightarrow H(u) * F(u)$$

also holds but is not useful here.

Now in the frequency domain, an ideal low-pass filter is easy to describe. It is simply a unit pulse, centered at the origin in the frequency domain, with width twice the cutoff frequency. If we multiply any frequency domain function by the unit pulse, it will be set to zero outside of the extent of the pulse, and will be unchanged under the pulse. The relationship between multiplication and convolution means that if we multiply the Fourier Transform $F_T(u)$, of a function $f(t)$ sampled at sampling period $T$, by the unit pulse and then take the inverse transform, we should get the same result as if we convolved the inverse Fourier Transform of the pulse with the original

$$
\begin{aligned}
p_T(t) &= \int_{-\infty}^{\infty} P_T(u) e^{i2\pi ut} du \\
&= \int_{-1/(2T)}^{1/(2T)} e^{i2\pi ut} du \\
&= \frac{e^{i\pi t/T} - e^{-i\pi t/T}}{i2\pi t}
\end{aligned}
$$

$$p_T(t) = \frac{\sin \pi t/T}{\pi t}$$

so for $T = 1$, we have

$$p_1(t) = \frac{\sin \pi t}{\pi t}$$

which is the *sinc* function with period 2 and maximum amplitude 1 at $t = 0$.

Figure 14.12: Inverse Fourier Transform of Unit Pulse of Width $2\pi/T$ Centered at Origin

sampled signal $f_T(t)$. Thus, the way to get the correct convolution kernel for the ideal low-pass filter in the spatial domain would be to compute the inverse Fourier Transform of the unit pulse in frequency. This derivation is shown in Figure 14.12.

## 14.5 Fast Fourier Transform

The Discrete Fourier Transform in the form given by Equations 14.9 and 14.10, and the two-dimensional version given by Equations 14.11 and 14.12 are easy to program. Let us assume that we have defined a type `Complex` for working with complex numbers, and that we have defined a function

`Complex W(int n, int N)`

which returns $e^{-i2\pi n/N}$ as a complex number. Then, an algorithm based on Equation 14.9 to compute the Discrete Fourier Transform of the array `f` of `N` elements is:

```
Complex *DFT(float f[], int N){
  int u, n;
  Complex *dft;

  dft = new Complex[N];
  for(u = 0; u < N; u++){
    dft[u] = 0;
    for(n = 0; n < N; n++)
      dft[u] += f[n] * W(u * n, N);
  }
  return dft;
}
```

Now, this algorithm is extremely simple, but it has a very real problem. That is that the two nested for-loops, each iterating $N$ times means that the algorithm will have a computation time proportional to the square of $N$. In the parlance of complexity analysis, we say that the algorithm is $O(N^2)$. Now, this is not a problem if the number of samples $N$ is small, but imagine a two dimensional version of this algorithm operating on a $512 \times 512$ image. Since the total number of pixels is approximately 1/4 of a million, then if we know the computation time to compute a one pixel Fourier Transform, we would have to expect that the total computation time will be on the order of 60 billion times greater! This is not likely to be a useful algorithm for working on real images.

Fortunately, there are several hidden symmetries in the calculation of the Discrete Fourier Transform that can be exploited for big time savings. The name given to the fast version of the DFT algorithm that exploits this is the *Fast Fourier Transform.* This is a bit of a misnomer, since what it computes is exactly the Discrete Fourier Transform, it just does it in an amount of time proportional to $N \log N$. This means that our $512 \times 512$ DFT that took 60 billion units of time using the naiive algorithm above, can be done in about 1.5 million units of time, a speed-up of about 40,000 times! Clearly, we need this algorithm.

The Fast Fourier Transform algorithm uses a *Divide and Conquer* strategy to exploit symmetries in the Discrete Fourier Transform calculation. What we mean by this is that at each stage in the calculation, the problem is separated into two problems, each 1/2 the size of the original problem. This separation is done recursively, for example breaking a size 4 problem into two size 2 problems, then dividing each of these size 2 problems into two size 1 problems. When the problem is reduced to a small enough size (size 1 in this case) so that it can be computed easily, the computation is done. Then, after computing each subproblem separately, the results are recombined to eventually form the final solution. By doing this division in a clever way, redundancies in the calculation of the DFT are uncovered and easily exploited.

First, let us define the function

$$W_N = e^{-i2\pi/N}.$$

Then substituting $W_N$ for the complex exponential and the variable $n$ for $t$, we can rewrite Equation 14.9 for the DFT as

$$F(u) = \sum_{n=0}^{N-1} f(n) W_N^{nu}. \tag{14.13}$$

The summation in Equation 14.13 can be subdivided into two summations, one that sums over the even numbered terms and one that sums over the odd numbered terms. This gives

$$F(u) = \sum_{n=0}^{N/2-1} f(2n) W_N^{2nu} + \sum_{n=0}^{N/2-1} f(2n+1) W_N^{(2n+1)u},$$

the second summation of which can be factored to yield

$$F(u) = \sum_{n=0}^{N/2-1} f(2n) W_N^{2nu} + W_N^u \sum_{n=0}^{N/2-1} f(2n+1) W_N^{2nu}. \tag{14.14}$$

But

$$W_N^2 = e^{-i2\pi/(N/2)} = W_{N/2},$$

so Equation 14.14 can be rewritten

$$F(u) = \sum_{n=0}^{N/2-1} f(2n) W_{N/2}^{nu} + W_N^u \sum_{n=0}^{N/2-1} f(2n+1) W_{N/2}^{nu}. \tag{14.15}$$

The summations in Equation 14.15 can be recognized as simply two DFT's, each of a function 1/2 the size of the original function. Let us define these two DFT's to be

$$F_e(u) = \sum_{n=0}^{N/2-1} f(2n) W_{N/2}^{nu},$$

representing the even numbered terms in the original DFT and

$$F_o(u) = \sum_{n=0}^{N/2-1} f(2n+1) W_{N/2}^{nu}$$

representing the odd numbered terms in the original DFT. Then we can rewrite Equation 14.15 to give the equation that forms the heart of the Fast Fourier Transform Algorithm

$$F(u) = F_e(u) + W_N^u F_o(u). \tag{14.16}$$

Equation 14.16 gives us a recursive formulation of the DFT calculation, i.e. the DFT is defined in terms of itself. All that we need to complete this recursive formulation is to determine a *base case* to end the recursion. Note that when $N = 1$, the DFT of the single sample is just the single sample itself, since $W_1 = e^{-i2\pi} = 1$. Thus the base case is that the DFT of a signal consisting of only a single sample is the sample itself. The following version of the DFT algorithm (which has a hidden flaw) is based on Equation 14.16 and this base case:

```
Complex *FFT(float f[], int N){
  int n, u;
  float *fe, *fo;
  Complex *Fe, *Fo;
  Complex *fft;

  fft = new Complex[N];
  if(N == 1)
    fft[0] = f[0];
  else{
    fe = new float[N/2];
    fo = new float[N/2];
    for(n = 0; n < N/2; n++){
      fe[n] = f[2*n];
      fo[n] = f[2*n + 1];
    }

    Fe = FFT(fe, N/2);
    Fo = FFT(fo, N/2);

    for(u = 0; u < N; u++)
      fft[u] = Fe[u] + W(u, N) * Fo[u];

    delete fe; delete fo; delete Fe; delete Fo;
  }

  return fft;
}
```

Did you find the flaw? On careful analysis it is made obvious by the array sizes – the DFT's $Fe$ and $Fo$ for the odd and even functions are represented in the algorithm by arrays `Fe` and `Fo`, each of which has only $N/2$ elements. However, the for-loop that merges the two partial calculations of the FFT needs to fill in $N$ entries in the output array `fft`. Three simple observations will give us all that we need to fix the problem, and give us a working algorithm.

First, note that the DFT of a function containing $N$ samples is really an infinite periodic function, with period $N$. This relationship can be seen in Figure 14.13, which depicts the complex exponential function $W_4$ as a sequence of unit vectors rotating around the origin of the complex plane. This periodicity can also be seen algebraically, because $W_N^{n+N} = W_N^n W_N^N$, but

$$W_N^N = e^{-i2\pi} = 1,$$

yielding

$$W_N^{n+N} = W_N^n.$$

Now, if $W_N$ is periodic with period $N$, then $W_{N/2}$ must be periodic with period $N/2$, and we can simply index the arrays `Fe` and `Fo` modulo $N/2$.

Second, note that $W_N^n$ and $W_N^{n+N/2}$ are mirror reflections of each other. Like the periodicity, this relationship can also be seen in Figure 14.13. The mirroring is because $W_N^{n+N/2} = W_N^n W_N^{N/2}$, but

$$W_N^{N/2} = e^{-i\pi} = -1,$$

yielding

$$W_N^{n+N/2} = -W_N^n.$$

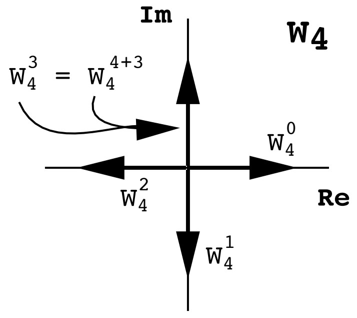


Figure 14.13: $W_4^n$ is Periodic in $n$ with Period 4

Finally, note that choosing a base case of size 2 instead of size 1 also leaves us with only a simple calculation. Since $W_2^0 = 1$ and $W_2^1 = -1$, we have

$$F[0] = f[0] + f[1],$$

and

$$F[1] = f[0] - f[1].$$

Using these three observations, we can recode the FFT algorithm to get the final working version:

```
Complex *FFT(float f[], int N){
  int n, u;
  float *fe, *fo;
  Complex *Fe, *Fo;
  Complex *fft;

  fft = new Complex[N];
  if(N == 2){
    fft[0] = f[0] + f[1];
    fft[1] = f[0] - f[1];
  }
  else{
    fe = new float[N/2];
    fo = new float[N/2];
    for(n = 0; n < N/2; n++){
      fe[n] = f[2*n];
      fo[n] = f[2*n + 1];
    }

    Fe = FFT(fe, N/2);
    Fo = FFT(fo, N/2);

    for(u = 0; u < N/2; u++){
      fft[u] = Fe[u] + W(u, N) * Fo[u];
      fft[2*u] = Fe[u] - W(u, N) * Fo[u];
    }

    delete fe; delete fo; delete Fe; delete Fo;
  }
  return fft;
}
```

There is one last thing to note about our algorithm. It is really only good for values of $N$ that are integer powers of 2. This is because our recursive subdivision and recombination process expects to be able to eventually subdivide the array `f` down to chunks, each of which is of size 2. This can only be done if $N$ is a power of 2. If $N$ is not a power of 2, then one or the other of the chunks will eventually reach an odd size and when it is subdivided the resulting chunks will not be of the same size. This restriction on the algorithm is not really a big limitation of the algorithm since we can always pad our image out so that it fits into the smallest rectangle the lengths of whose sides are powers of 2.

Now, let us look at the computational complexity of the algorithm. My claim was that we would be able to achieve $O(N \log n)$ performance. The old algorithm had two nested loops, each of which iterated $N$ times, yielding $O(N^2)$ performance. In the new algorithm the nesting of loops is gone, being replaced by subdivisions and recursive calls. At each level of recursion, the main calculation loop iterates $N/2^L$ times, where N is the size of the original problem, and $L$ is the level of recursion, i.e. $L = 1$ at the top level, 2 after one subdivision, etc. But, at level $L$ the problem is subdivided into $2^{L-1}$ chunks, so the total number of iterations at each level is $(N/2^L)2^{L-1} = N/2$. Now, if $N = 2^M$, then the deepest level of subdivision will be $L_{max} = \log_2 N = M$. So, the algorithm does $N/2$ iterations at each of $\log_2 N$ levels, yielding an algorithm whose performance will be proportional to $N \log N$.