CS 351 Computer Graphics, Spring 2017

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

Computer graphics deals with the manipulation and creation of digital imagery. We cover drawing algorithms for two-dimensional graphics primitives, 2D and three-dimensional matrix transformations, projective geometry, 2D and 3D model representations, clipping, hidden surface removal, rendering, hierarchical modeling, shading and lighting models, shadow generation, special effects, fractals and chaotic systems, and animation techniques. Labs will focus on the implementation of a 3D hierarchical modeling system that incorporates realistic lighting models and fast hidden surface removal.

Prerequisites: CS 251 or permission of instructor. Linear algebra recommended.

Desired Course Outcomes

A. Students understand and can implement the fundamental concepts of creating and manipulating images.

B. Students understand and can implement the fundamental concepts of rendering, including matrix transformations, shading, and hidden surface removal.

C. Students understand and can implement the fundamental concepts of modeling objects and scenes and hierarchical modeling systems.

D. Students work in a group to design and develop 3D modeling and rendering software.

E. Students present methods, algorithms, results, and designs in an organized and competently written manner.

F. Students write, organize and manage a large software project.

1 Graphics

If you could make a picture, what would it be?

1.1 Graphics is...

The field of computer graphics impacts many disciplines and activities. In some cases its use is specialized to a particular application, such as scientific visualization. In other cases, its impact is broad and ubiquitous, such as windowing systems on a computer.

- Vector graphics: **rendering** images using points and lines.

  (Rendering: to create the actual image, to make it visible to the user. "cause to be or become; make.")

  Some types of images are built from lines and shapes. Common examples include logos, fonts, and wire-frame models. An image like a logo or a letter should look the same regardless of its size. The important characteristics of these images are not their pixels, but the geometric relationship of the image elements, which can be described as a set of points, lines, vectors, arcs, polygons, or other shapes. If we represent these types of images as geometric entities, converting them to pixels only as the image is being rendered to a device, then we are free to rotate, scale, translate, and otherwise transform them arbitrarily, manipulating only the points and vectors being used to represent them. If we had to represent a letter as a set of pixels, then scaling it to a larger size would result in blocky artifacts (the **jaggies**).

- 2-D shapes and shape manipulation: presentation software

  When we are creating documents we often want to manipulate the elements of the document in arbitrary ways. Again, we are best served by thinking about the elements of the document as mathematical objects described by points and vectors, which are amenable to mathematical transformations. Selecting, moving, scaling, rotating, and transforming these elements are all enabled by the fundamental concepts of CG.

- Drawing and photo manipulation programs: from MacPaint to Photoshop

  Drawing and photo manipulation programs often blend vector graphics with pixel-based manipulation. Manipulating pre-existing photos is a branch of computer graphics focused on both creative manipulation and post-processing for photo enhancement. Much of the field of photo manipulation falls in the category of image processing, or signal processing more generally, where we think of the image as a 2-D signal or function to be filtered and modified.

- Graphical User Interfaces [GUI] and computer windowing interfaces

  Graphical user interfaces are the primary method we have of interacting with most computers (e.g. windowing systems, widgets, buttons and sliders, point and click, click and drag). Rendering the windows in the proper order and ensuring the user sees the correct information are core computer graphics concepts. Clipping–identifying when one graphic item occludes another–is a key concept underlying windowing systems, and fast clipping algorithms are critical to making GUIs run fast enough to be practical. Likewise, all of the various effects from bouncing applications to sliding buttons to flowing windows are the result of CG algorithms.

- **Rendering** pictures of 3-D scenes from models

  Taking a 3-D model and creating a 2-D image of it from a selected viewpoint is the canonical computer graphics problem. It encompasses model representation and creation, the definition of a view, the definition of the viewing geometry, the projection of a 3-D world onto the 2-D viewing geometry, and the rendering of geometric objects on to a raster representation. In each stage, the efficiency of the representation and the speed of the process are critical to making CG practical.

- **Animation** of models

  Making things appear to move realistically is hard to do. While rigid objects can be usually be controlled through Newtonian mechanics, once things have multiple jointed connections, become floppy or springy, can fall apart, can interact with other objects, or deform under a variety of forces, the equations and interactions can become extremely complex. In addition, when making things move we often want them to be in certain places at certain times, such as when animating a character in a movie. How do we make the motion natural, while still achieving a specific outcome? There are a large number of tools available not only for creating motions from scratch, but also for capturing the motion of real things and then mapping that motion onto virtual objects. After rendering, animation is the second fundamental task in CG.

- **Modeling** of physical or virtual phenomena

  In addition to animating objects, modeling real-world or other-world phenomena–static or dynamic–is the third fundamental task of CG. Examples include creating realistic wear on stone monuments, the accurate modeling of reflections on shiny surfaces or sub-surface scattering in skin or semi-transparent liquids, and realistic smoke, fire, and water. There are many visual phenomena that are difficult to model and require complicated algorithms or equations to simulate properly.

- Computer-Aided Design [CAD] or Computer-Aided Modeling [CAM]

  Almost all manufactured devices available today were initially designed and modeled using a computer graphics program. CAD design is any process where a computer is the primary design tool, and computer-aided modeling enables testing of device or object functionality prior to physically creating the object. With the advent of 3D printers, it is now possible to create fast prototypes, or final versions, of almost any design in many different types of materials.

- Training and simulation, such as flight simulators

  Training people for a complex task can be expensive, if not outright dangerous. Flight simulators were one of the first commercial applications of computer graphics because training people in real airplanes, and putting those airplanes through dangerous maneuvers is an expensive proposition (time, money, and lives). One common application of CG is to provide useful, often realistic, interactive training systems without requiring real world resources and exposing the trainees to dangerous situations.

- Virtual reality and immersive 3D environments

  Useful for training, research, and entertainment, virtual reality [VR] and immersive 3D environments enable people to experience a more complete version of a 3D world. there are many different types of VR systems with a range of capabilities. But the key difference between a VR system and a passive system is that the system must respond to the motions of the viewer to maintain the illusion that the viewer is inside the virtual world and their motions and actions have the correct effects on that world.

- Visualization of data, such as medical imaging or web networks

  How to visualize data in ways that enable people to see patterns is an important sub-field of CG. There are a variety of alternative projection schemes that map data into different types of spaces for visualization. A simple example is visualizing data on a sphere. We can unwrap the sphere to create a 2-D map, or we can try to create an interactive visualization of the data on a sphere that can be moved and rotated by the user so that it appears to retain its 3D shape–even though it is being viewed on a 2D image.

- Games, both 2D and 3D

  Computer games are now the largest segment of the visual entertainment industry. Almost all visual aspects of games fall into the realm of CG, including the realistic modeling and AI systems necessary to create challenging opponents that move and act appropriately. (Note the blurry lines between CG and other domains when modeling things in the world starts to require a wide variety of knowledge and algorithms from other disciplines.)

1.2 Representing images and colors

At its core, graphics is about creating images from some kind of input data. A video is simply a coherent sequence of images. Images consist of a combination of spectral and spatial information. In vector graphics, the information is encapsulated as line segments or other crisp geometric entities. In raster graphics, the image is represented by a set of pixels. Normally, the pixels are laid out on a grid, such as a regular rectangular grid, and each pixel is represented by a 3-tuple $(x, y, \vec{C})$, where $\vec{C}$ is the spectral information. In most consumer devices and sensors, the spectral information consists of a 3-tuple (R, G, B). However, $\vec{C}$ may be a scalar value for greyscale devices or hundreds of values for hyper-spectral imaging devices.

What is color?

One important concept to understand as a CG developer or artist is that color is a perception. The perception of color is caused by electromagnetic [EM] radiation hitting sensing cells in our eyes. While different EM spectra will cause different responses in our sensors, it is not the case that different EM spectra will always cause different perceptions of colors. In fact, it is possible to adjust a person's color perception of a particular EM spectrum by modifying the EM spectra of surrounding areas. This has been shown in many psychophysical studies, and CG can take advantage of this fact in order to create particular experiences for the user.

For example, if we want to make the user think there is a light source in the scene, then we have to control the surrounding context to make the light source appear brighter than what is around it. Given that we are limited to a fixed gamut of colors, and the screen is limited to a maximum amount of energy at each pixel, the only way to make something appear to be emitting light is to make the rest of the scene darker. The amazing thing is that we do not necessarily perceive the room to be darker. Once we perceive that the light source is self-emitting, our brains adjust our perception of the room so we still perceive the rest of the room to be well lit. Understanding the psychophysics research related to color perception and other human visual system phenomena enable us to make more effective use of the tools that we have to create the perceptions we want to occur in our users.

How do we sense color?

Humans have (at least) four different kinds of sensors in the retina.

- One type, called the rods, sense EM radiation across a broad spectrum. Since they do not differentiate between different frequencies, rods sense in what we perceive as greyscale.
- The other type, called the cones, sense particular parts of the EM spectrum. There are three types of cone sensors that sense long, medium and short wavelengths. These correspond to what we perceive as the red, green, and blue parts of the spectrum. Different spectra produce different responses in the three types of cones. Those different patterns get converted by the brain into colors.
- There is evidence for a fourth type of sensor in the retina that is sensitive to a very small part of the spectrum that corresponds to a piece of the daylight spectrum. These sensors appear to be linked to circadian rhythms and may or may not play a role in vision or color perception.

The EM spectrum from 380nm to 780nm, which covers the range of visible light, has an infinite number of potential spectra. However, the physics of reflection and emission mean that the set of EM spectra we are likely to encounter in our lifetimes is a small subset of all possible spectra. Experiments have shown that sampling the EM spectrum and representing it with three values is sufficient to differentiate most materials and represent most spectra in the physical world.

There are, however, materials that will look identical in one lighting situation to our eyes, but different under a second lighting situation. This phenomenon is called metamerism, and it is caused by the fact we cannot sense the details of the EM spectrum.

The neurons in the human retina also have the capability to inhibit one another, so a cone of one color can end up with a negative response to certain stimuli. The relevance to comptuer graphics is that there are colors we can perceive–ones with negativ responses in one of the color channels–that cannot be represented on a physical display device that uses additive emission to generate colors.

How do we generate color with a computer?

Color is generated on a computer monitor using some method of generating different EM spectra.

- Cathode Ray Tube (CRT): these use electron guns to activate different kinds of phosphors arranged in an array on a screen. There are generally three different colors of phosphor (red, green, and blue) and they mix together to generate different spectra.
- Liquid Crystal Display (LCD): these use small liquid crystal gates to let through (or reflect) differing amounts and colors of light. Usually, the gates are letting light pass through what are effectively R, G, or B painted glass. LCD displays are currently the most common types of displays in use.
- Plasma Display (PDP): these use very small R, G, B fluorescent lamps at each pixel. The lamps are filled with a noble gas and a trace amount of mercury. When a voltage is applied to the lamp, the mercury emits a UV photon that strikes a phosphorescent material that emits visible light. Plasma colors match CRT colors, because both use the same phosphorescent materials.
- Organic Light Emitting Diodes [OLED]: these are materials that emit photos of varying wavelengths when a voltage is applied in the correct manner. Like plasma displays, OLEDs are an emissive technology that uses collections of small OLEDs to generate a spectrum of colors.

How do we represent colors with a computer?

- The internal representation of colors matches the display (and viewing) mechanisms: mix three intensity values, one each for red, green, and blue.
- The RGB system is additive: with all three colors at maximum intensity you get white.
- There are many other color spaces we can use to represent color
  - HSI: hue, saturation, intensity
  - YIQ: intensity (Y) and two color channels (in-phase and quadrature), the original US broadcast TV standard
  - YUV: intensity (Y) and two color channels (UV), used by PAL, the original European broadcast standard
- We need to use a certain number of bits per color channel to generate different colors. Common formats include: 1, 5/6, 8, and 16 bits/pixel. High end cameras now capture in up to 14 bits/pixel.
- Internally, we will likely be using floating point data to represent pixels.

How do we represent pictures as files?

- Somehow we have to store the information in the image, which may include colors, but may also include shapes, text, or other geometric entities.
- Raster image format: each pixel gets a value
  - TIFF: Tagged Image File Format, non-lossy, any data depth is possible
  - GIF: Graphics Interchange Format, lossy, at most 256 colors, supports animation
  - JPG: long name, lossy method (can't recreate the original data), compacts images well
  - PNG: Portable Network Graphics, non-lossy compression, retains original data
  - PPM: Portable Pixel Map, very simple image representation with no compression
  - RLE: Run-Length Encoded, simple, non-lossy (but not very good) compression
- Vector image representations: images are collections of objects with geometric definitions
  - SVG: Scalable Vector Graphics, created for the web, based on XML
  - GXL: Graphics eXchange Language, also created for the web and based XML
- Element list representations: images are collections of objects and pictures
  - PICT
  - PDF
- Graphical language representations: Postscript

1.3 Getting Started...

We're going to use the PPM library for reading/writing/storing images. There are convenient tools for converting PPM images into any other format you wish, so the fact that they are not compressed or readable by a browser is not a big deal. We'll use C/C++ to write graphics programs because it's fast.

1.3.1 C/C++ Basics

C is a language not too far removed from assembly. It looks like Java (or Java was based on C syntax) but it's not. The key difference between Java and C/C++ is that in C you have to manage memory yourself. In C you have access to any memory location, you can do math on memory locations, you can allocate memory, and you can free memory. This gives you tremendous power, but remember the Spidey rule: with great power comes great responsibility. The corollary of the Spidey rule is: with great power come great screw-ups.

C is a functional language, which means all code is organized into functions. All executable C programs must have one, and only one function called main. Execution of the program will begin at the start of the main function and terminate when it returns or exits. You can make lots of other functions and spread them around many other files, but your executable program will always start with main.

C++ is an object oriented language that allows you to design classes and organize methods the same way you do in Java. C++ still retains its functional roots, however, and your top level program still has to be main. You can't make an executable function out of a class as you can in Java.

Code Organization

There are four types of files you will create: source files, header files, libraries, and object files.

- Source files: contain C/C++ code and end with a .c or .cpp suffix (.cc is also used for C++)
- Header files: contain type denitions, class declarations, prototypes, extern statements, and inline code. Header files should never declare variables or incorporate real code except in C++ class definitions.
- Libraries: contain pre-compiled routines in a compact form for linking with other code.
- Object files: object les are an intermediate step between source les and executables. When you build an executable from multiple source les, using object les is a way to speed up compilation.

The main function and command line arguments

One of the most common things we like to do with executable shell functions is give them arguments. C makes this easy to do. The main function should always be defined as below.

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

  return(0);
}
```

The main function returns an int (0 for successful completion) and takes two arguments: argc and argv. The argument argc tells you how many strings are on the command line, including the name of the program itself. The argument argv is an array of character arrays (strings). Each separate string on the command line is one of the entries in argv. Between the two arguments you know how many strings were on the command line and what they were.

Data Types

The basic C data types are straightforward.

- char / unsigned char: 8 bits (1 byte) holding values in the range [-128, 127] or [0, 255]
- short / unsigned short: 16-bits (2 bytes) holding values in the range [-32768, 32767] or [0, 65535]
- int / unsigned int: 32 bits (4 bytes) holding signed or unsigned integers up to about 4 billion
- long / unsigned long: 32 (4 bytes) or 64 bits (8 bytes), depending on the processor type holding very large integers
- float: 32-bit (4 byte) IEEE floating point number
- double: 64-bit (8 byte) or longer IEEE floating point number

There aren't any other basic data types. There are no native strings. You can create structures that are collections of basic data types (the data part of classes in Java). You can create union data structures where a single chunk of memory can be interpreted many different ways. You can also create arrays of any data type, including structures or unions.

```
  int a[50];
  float b[20];
```

The best way to create a structure is to use the typedef statement. With the typedef you can create new names for specific data types, including arrays of a particular size. The following creates a data type Vector that is an array of four floats.

```
typedef float Vector[4];
```

Example: Defining a structure

```
typedef struct {
  short a;
  int b;
  float c;
} Fred;
```

The above defines Fred to be a structure that consists of three fields a, b, and c. The syntax for accessing the fields of Fred is dot-notation. The following declares two variables of type Fred. The first is initialized in the declaration, the second is initialized using three assignment statements.

```
  Fred tom = {3, 2, 1.0};
  Fred f;

  f.a = 6;
  f.b = 3;
  f.c = 2.0;
```

C does not pre-initialize variables for you (Java does). Whatever value a variable has at declaration is the result of random leftover bits sitting in memory and it has no meaning.

Strings

C does not have a built-in string type. Generally, strings are held in arrays of characters. Since an array does not know how big it is, C strings are null-terminated. That means the last character in a string must be the value 0 (not the digit 0, but the value 0). If you create a string without a terminator, something will go wrong. String constants in C, like "hello" will be null-terminated by default. But if you are manually creating a string, don't forget to put a zero in the last place. The zero character is specified by the escape sequence '\0'.

Since strings in C are null-terminated, you always have to leave an extra character for the terminator. If you create a C array of 256 characters, you can put only 255 real characters in it.

Never allocate small strings. Filenames can be up to 255 characters, and pathnames to files can get large very quickly. Overwriting the end of small string arrays is one of the most common (and most difficult to debug) errors I've seen.

C does have a number of library functions for working with strings. Common ones include:

- strcpy(char *dest, char *src) - copies the source string to the destination string.
- strcat(char *dest, char *src) - concatenates src onto the end of the destination string.
- strncpy(char *dest, char *src, size_t len - copies at most len characters from src into dst. If src is less than len characters long, the remainder of dst is filled with '\0' characters. Otherwise, dst is not terminated. This is a safer function than strcpy because you can set len to the number of characters that can fit into the space allocated for dest.
- strncat(char *dest, char *src, size_t count) - appends not more than count characters from src onto the end of dest, and then adds a terminating '\0'. Set count appropriately so it does not overrun the end of dest. This is a safer function than strcat.

To find out about a C library function, you can always use the man pages. Typing `man strcpy`, for example, tells you all about it and related functions.

Header Files

You will want to create a number of different types for your graphics environment. In C the best way to put together new types is the typedef statement. In C++, use classes. Both types of declarations should be placed in header les. As an example, consider an Image data type. In C, we might declare the Image data type as below.

```
typedef {
  Pixel *data;
  int rows;
  int cols;
} Image;
```

The difference with C++ and using a class is not signicant, except that in C++ you can associate methods with the class.

```
class Image {
  public:
    Pixel *data;
    int rows, cols;

    Image();
    Image(int r, int c);
};
```

Prototypes of functions also belong in header les. Prototypes describe the name and arguments of a function so that it is accessible in any source le and the compiler knows how to generate the code required to call the function.

```
Pixel *readPPM(int *rows, int *cols, int *colors, char *filename);
```

Extern statements are the appropriate method for advertising the existence of global variables to multiple source les. The global variable declarations themselves ought to be in source les. Initialization of the global variables also need to be in the source les. If the declaration itself is made in the header le, then multiple copies of the global variable may exist. Instead, an extern statement advertises the existence of the variable without actually instantiating it.

```
extern int myGlobalVariable;
```

Inline functions are small, often-used functions that help to speed up code by reducing the overhead of function calls. Rather than use a typical function call that requires pushing variables onto the stack, inline functions are copied into the function from which they were called. Because the functions are copied by the compiler, the compiler must have access to inline functions during compilation. Therefore, inline functions must be located in the header les. They are the only C code that belongs in a header le. In C++, methods dened within the class declaration are implicitly inline, but not necessarily. It is a good idea to only dene methods explicitly declared as inline in the header file, especially for large projects.

Useful include files

Standard include les for C provide denitions and prototypes for a number of useful functions such as printf(), provided by stdio.h, malloc, provided by stdlib.h, and strcpy(), provided by string.h. In addition, all math functions such as sqrt() are provided b math.h. A good template for include les for most C programs is given below.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

When using C++, if you want to use functions like printf(), you should use the new method of including these les, given below. In addition, the include le iostream is probably the most commonly used include le for C++.

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include <iostream>
```

Pointers

Variables hold data. In particular, variables hold collections of ordered bits. All variables hold nothing but bits, and what makes them different is how we interpret the bits.

In C there are two major subdivisions in how we interpret the value in a variable. Some variables hold bits that we interpret as data; it has meaning all by itself. Some variables hold addresses: they point to other locations in memory. These are pointers.

When you declare a variable, it gives a label to some memory location and by using the variable's name you access that memory location. If the variable is a simple data type (e.g. char, int, float, double) then the memory location addressed by the variable can hold enough bits for one of those types. If the variable is a pointer (e.g. char *, int *, float *, double *) then the memory location addressed by the variable can hold enough bits for the address of a memory location.

Until you allocate memory for the actual data and put the address of that allocated location into the pointer variable, the pointer variable's address is not meaningful.

You can declare a pointer variable to any data type, including types that you make up like arrays, structures and unions. Adding a * after the data type means you are declaring a pointer to a data type, not the actual data type itself. That means you have created the space for an address that will hold the location of the specified type.

To allocate space for actual data, use the malloc function, which will allocate the amount of memory request and return a pointer to (the address) of the allocated memory. The sizeof function returns the number of bytes required to hold the specified data type.

Example: Declaring and allocating pointers

```
int *a; // declare a pointer to an integer

a = malloc(sizeof(int)); // allocate memory for the integer

*a = 0; // assign a value to the integer

free(a); // free the allocated memory (the address in a is no longer valid)
```

The above first declares an int pointer and allocates space for it. The next line says to dereference the pointer (*a), which means to access the location addressed by a, and put the value 0 there. The final line frees the space allocated in the malloc statement.

Every malloc statement should be balanced by a free statement. Good coding practice is to put the free statement into your code when you make the malloc. The power of C is that you can, if you are sure about what you're doing, access, write and manage memory directly.

Arrays

Arrays in C are nothing but pointers. If you declare a variable as `int a[50];` then the variable `a` holds the address of a memory location that has space for 50 ints. The difference between `int *a;` and `int a[50];` is that the former just allocates space for the address of one (or more) integers. The latter allocates space for 50 integers and space for their address and puts the address in `a`.

The benefit to making arrays using simple pointers is that you can set their size dynamically. Because arrays are pointers, you cannot copy their value from one array to another using a simple assignment. That just copies the address of one array into the variable holding the address of the second array (which is bad). You have to copy arrays element by element.

Example: creating an array

```
int *a;
int size = 50;
int i;

a = malloc(sizeof(int) * size);

for(i=0;i<size;i++) {
  a[i] = 0;
}

free(a);
```

The above creates an array of 50 ints and puts the address of that memory in `a`. It then puts the value 0 in each of the memory locations and then frees the memory. If you want to create 500 ints, all you have to do is change the value of the variable `size`.

You can declare multi-dimensional arrays in C.

```
int a[4][4]; // creates a 4x4 array
```

However, multi-dimensional arrays in C don't act like true 2-D arrays. For a fixed-size data type, like a 4x4 matrix, they work fine. But you can't directly allocate a multi-dimensional array using malloc. You have to build multi-dimensional arrays yourself, as shown in the next section.

1.3.2 Creating, Reading, Modifying and Writing Images

Images are big 2D arrays. The common way to address a particular image location is (row, column) notation. This is different than traditional mathematical notation, which generally puts the horizontal axis (x-axis) first and accesses 2D locations using (x, y). The reason is that data on a screen is organized in row-major order. All of the pixels on the first row (top row) of an image come first in order from left to right. Just remember that x is a column and y is a row.

Any image can be represented as a single big array. Just take all the rows from top to bottom and concatenate them together. The index of pixel (r, c) is then `[r * cols + c]`. To allocate an image as a big array, we can just figure out how many pixels there are an allocate enough for the image.

We can also create real 2-D arrays by first creating an array of row pointers and then setting their value to the proper addresses in a pixel array. The latter makes accessing random row-column locations easier to code in many situations. Allocating, setting up, and freeing 2-D arrays takes a bit more effort. Both types of image creation and access are shown in the example below.

To read and write images we are going to use a collection of functions that read and write PPM (color) and PGM (greyscale) images. These functions in located in ppmIO.c and prototyped in ppmIO.h, which you will want to include in your programs. The function readPPM reads in an image, gets the number of rows and columns, and returns a pointer to the image data as a single array. The function writePPM writes out image data to a file as a PPM image.

```
#include <ppmIO.h>

int main(int argc, char *argv[]) {
  Pixel *image;
  int rows, cols, colors;
  int i;

  // read the image
  image = readPPM( &rows, &cols, &colors, argv[1] );

  // swap blue and red channels
  for(i=0;i<rows*cols;i++) {
    unsigned char tmp = image[i].r;
    image[i].r = image[i].b;
    image[i].b = tmp;
  }

  // write out the new image
  writePPM(image, rows, cols, colors, argv[2] );
}
```

Example: creating an image

The following creates an image as a 1-D array and initializes it to black.

```
int rows = 50;
int cols = 50;
int size = rows * cols;
int i;
Pixel *image;

image = malloc(sizeof(Pixel) * size);

for(i=0;i<size;i++) {
  image[i].r = 0;
  image[i].g = 0;
  image[i].b = 0;
}

free(image);
```

The following creates a 2-D array and initializes it to black.

```
int rows = 50;
int cols = 50;
int size = rows * cols;
int i, j;
Pixel **image; // note the image is a double pointer

image = malloc(sizeof( Pixel *) * rows ); // allocate row pointers
image[0] = malloc(sizeof(Pixel) * size); // allocate the pixels
for(int i=1;i<rows;i++) {
  image[i] = &( image[0][i*cols] ); // set the row pointers
}

for(i=0;i<rows;i++) {
  for(j=0;j<cols;j++) {
    image[i][j].r = 0;
    image[i][j].g = 0;
    image[i][j].b = 0;
  }
}

free(image[0]); // free the pixel data
free(image); // free the row pointer data
```

1.3.3 Compilation Basics

C/C++ files must be compiled before you can run your program. We will be using gcc/g++ as our compiler. As noted above, you can distribute your functions/classes among many files, but only one of the files you compile together into an executable can have a main function. It's good coding style to distribute functions across a number of files in order to keep things modular and organized. As in Java, it's common to use a separate file for each class.

You can always list all of the C/C++ files you want to compile together on the command line and tell gcc to build your executable. The example below compiles the three C files and links them together into a single executable called myprogram (the -o flag tells gcc what to call the output.

```
gcc -o myprogram file1.c file2.c file3.c
```

The problem with this approach is that every time you make a change to one of the files, all of the files get recompiled. With a large project, recompiling all of the files can be time consuming. The solution to this is to create an intermediate file type called an object file, or .o file. An object file is a precompiled version of the code ready to be linked together with other object files to create an executable. The -c flag for gcc/g++ tells it to create an object file instead of an executable. An object file is always indicated by a .o suffix. To build the same program as above, you would need to precompile each of the C files and then link the object files.

```
gcc -c file1.c
gcc -c file2.c
gcc -c file3.c
gcc -o myprogram file1.o file2.o file3.o
```

This seems like a lot of extra work, until you make a change to file1.c and then want to recompile. You can rebuild the executable using the commands

```
gcc -c file1.c
gcc -o myprogram file1.o file2.o file3.o
```

This can same you lots of time when you are working with a large project, because the files you didn't touch don't have to be recompiled. gcc/g++ just uses the object file to link things together.

Sometimes you also want to link in libraries such as the standard math library or libraries that you built yourself. The -l flag lets you link in existing libraries that are in the compiler's search path. For example, to link in the math library, you would change the last line of the above example to be the following.

```
gcc -o myprogram file1.o file2.o file3.o -lm
```

When you want to link in a library you've built yourself, you need to not only link it with the -l flag, but also tell the compiler where to look with the -L flag. The example below tells the compiler to go up one directory and down into a subdirectory named lib to look for libraries. If the file libmylib.a is located there, it will successfully link.

```
gcc -o myprogram file1.o file2.o file3.o -L../lib -lmylib -lm
```

You may also need to tell the compiler where to find include files (.h files). If, for example, all of your include files in an include subdirectory, you will want to add a -I (dash capital I) option to the compile line when compiling the .c files. Linking is too late for includes, as they must be available during compilation. The following compiles the file file1.c into an object file file1.o and tells the compiler to look in a neighboring include subdirectory called include.

```
gcc -c file1.c -I../include
```

There are many other flags that can be used by gcc. Two you will see this semester are -Wall, which turns on all warnings, and -O2, which sets the optimization level to 2, which is fairly aggressive. It will likely make your code a bit faster, which is good.

Development Tools

The mac does not come with development tools in a standard install, although they are included on the installation disks. You need the Xcode package to get gcc/g++ and standard development tools like make. If you also want X-windows and other Unix tools and programs, then you will want to download XQuartz (version 2.3 as of September 2008) and MacPorts. MacPorts is a package manager and lets you install things like ImageMagick, gimp, and xv. Install Xcode, the XQuartz, the MacPorts. It will take a while, but none of it is particularly difficult.

Windows also does not come with development tools standard. The cygwin package is the most common way developers get an X-windows environment and all of the standard unix development tools. Note that there may be additional packages you need to download in addition to the standard install.

1.3.4 Makefiles

The make program is a way of automating the build process for programs. It's most useful for large programs with many files, but it's also useful for smaller programs. It can save you from typing a lot of stuff on the command line and keeps you from having to remember all the flags every time.

The simplest makefile is just two lines. The first line defines the name of the rule and its dependencies, the second line defines the rule's action.

```
myprogram: file1.c file2.c file3.c
    gcc -o myprogram file1.c file2.c file3.c -I../include -lm
```

The first line defines a rule called myprogram and lists the files upon which the rule depends. If any of those files change, the rule should be executed. The second line defines the action of the rule. If you type `make myprogram` the rule will execute. In fact, since make executes the first rule by default, all you need to type is `make` and the rule will execute.

When you're just starting out, create simple makefiles where the rules are spelled out explicitly. There are many more things you can do with make, and it incorporates a complete scripting language for automating complex tasks. For more information, see the makefile tutorial linked to the course website.

1.4 The Art of Graphics

Graphics is ultimately about creating images. A good artist knows about their canvas, their paints, and their brushes and uses the appropriate tools depending upon the scene they want to create. The same is true for computer graphics. The more you understand about your canvas, paints, and brushes the better images you can create.

Unlike physical artists, a digital artist is not limited in their vision of a brush. Any digital tool can be used as a building block to generate imagery, including hierarchical combinations of tools and procedural or algorithmic tools.

1.4.1 Canvas

Digital images or image sequences are most easily displayed electronically on a 2D raster screen such as a CRT, LCD, or plasma display.

- Raster images are collections of discrete pixels.
- Electronic displays use an additive color system (RGB).
- Each pixel contains all the data required to display itself.
- The information represented in the image is constant; each refresh requires a read through the entire **frame buffer**–the memory in which the image to be displayed is stored.
- Each pixel represents a finite area that will be a single color, which can make it difficult to render high frequency details without high spatial resolution. The spatial resolution of the screen, and the shape of the pixel itself creates a situation where we need to think about how to render high spatial frequency information on a lower spatial frequency device. The most common artifacts in computer graphics–**the jaggies**, or stair-step artifacts on what should be smooth continuous shapes–come from this property of raster displays.
- The retina display–a raster display where the pixels are so small that normal human vision cannot distinguish them–creates a fundamental change in the way we think about rendering graphics. When the display is at a higher resolution than the material we need to display, it removes the need to worry about artifacts like jaggies.
- The **display refresh rate** is an important value to know. The image is updated at a rate between 60-75Hz (Hz = 1/s). Therefore, when creating real time animation, your system has between 13-16ms to generate the next frame.
- The system generally has two **frame buffers**: a frame buffer is the memory in which the image to display is stored. Normally, one frame buffer is being transferred to the display while the other frame buffer is being written by the graphics engine.

Vector display systems, which are rare today, provide a different kind of canvas.

- Vector displays have a single electron gun and single color long-glow phosphor. The phosphorous on the screen has an extremely fine grain size, meaning the spatial resolution is at or beyond the limit of our eyes to distinguish individual elements.
- Vector displays generally draw lists of lines represented as point pairs.

- For each line, the electron gun moves from the starting point to the ending point, drawing a line that is the width of the electron beam.
- The system draws exact lines between high resolution locations; the coordinate system does not need to be integral, but exists at the resolution of the electron gun control system.
- The speed of the system determines the maximum number of lines that can be drawn per second. Therefore, the more lines there are in the image, the longer it takes to draw the entire list of lines. The speed of animation sequences, therefore, is determined by the complexity of the scene.
- The only parts of the screen affected are the lines, and they are drawn at the resolution of the electron gun focus and phosphor grain.

Hard-copy devices such as laser printers, inkjet printers, and plotters are common ways to view images.

- Most hard copy devices are raster devices with a high resolution, but not necessarily high enough to avoid spatial artifacts.
- Plotters that worked like vector displays used to be more common. They take longer to print, but can produce artifact-free copy.
- Hard-copy devices must use a subtractive system (CMYK) where each kind of ink subtracts one band (R, G, or B).
- Synchronizing printer output and an electronic display is non-trivial. People who depend upon high quality output use calibration equipment to guarantee that the color on the display is the color that comes out of the printer.
- 3-D printing devices are becoming more common, enabling the physical realization of 3D models.

1.4.2 Paints

While the internal representation of an image is almost always RGB (3-channel) or greyscale (1-channel), there are other ways to describe colors.

- RGB–red, green, and blue–is an additive color system and is the most common internal representation of colors. It roughly matches the output display device representation, so the complexity of the transformation of color information from internal representation to display is minimized. RGB color space is usually visualized as a cube. While RGB is useful as a data representation, it is not intuitive as a color selection space (e.g. what is orange?).
- HSI/HSV–hue, saturation, and intensity or value–is usually visualized as a cylinder. The intensity axis is the cylinder's central axis, saturation is distance from the central axis, and hue is the angle around the circle. HSI is a more intuitive color space for people because intensity and color are more clearly separated and the hue captures what we generally think of as color names (red, orange, yellow, green, blue, indigo, violet).
- Ad hoc representations: think about color picker options and all the various ways colors can be represented (e.g. crayons).
- High spectral resolution representations: maintain continuous or high frequency discrete representations of spectra throughout the rendering process.

1.4.3 Brushes

Input devices have always brought out tremendous creativity in engineers and scientists. In computer graphics, a brush is any device that can capture information from the world or generate it synthetically. That is a pretty large set of tools.

- Buttons (keyboard)
- Dials
- Position measurement (mice, trackpads, tablets, data-gloves, body suits, mo-cap systems)
- 2-D scanners
- 3-D scanners
- Motion capture
- Digital images and videos
- Stereo imagery
- Satellite/radar data
- Haptic inputs
- Microphones/music

Manual paint tools (e.g. Canvas, Photoshop, Gimp)

- Create 2D images
- Create 3D images
- Modify existing pictures
- Create textures

Plotting pixels (write code or use MacPaint)

- Tough to create sophisticated images this way
- Pixels are the most basic unit of creativity in graphics, so an artist will almost always want access to individual pixels to create a finished product.

Graphics Primitives

- Lines, curves, rectangles, ellipses, splines
- One level of abstraction above pixels
- Filling shapes affects large areas of pixels

3D Models

- 3D lines, spheres, cylinders, prisms, polygons, blobs
- Teapots, trees, buildings, other shapes in a library
- Free-form surfaces: Bezier surfaces, NURBS, subdivision-surfaces
- Another level of abstraction above graphics primitives
- Permits design in a 3D space, but requires a path to view the space

Procedural Models

- Landscape brushes, vegetation models
- Building cities from algorithms
- Weathering stone
- Mountains
- Builds upon simple 3D models to create complex aggregate models

Renderer

- Converts 3D models into a 2D image
- Incorporates models of appearance
- Requires designing the viewpoint and the view
- The Director's job

Animation

- Many levels of abstraction
- Manually specify points
- Manually specify key frames and let the computer interpolate the rest
- Manually specify paths (3D curves)
- Manually specify joint relationships
- Procedural animation - write scripts that automate the above
- AI - write scripts that guide agents and their behavior

Non-photorealistic Rendering

- Forget realism, go for artistic styles
- Has practical application in technical drawings
- Abstracts away from pixels and standard graphics primitives
- Some NPR methods are hyperrealistic imitations of physical artistic forms

2 Graphics Primitives

Graphics primitives are the next level of abstraction above pixels. They are visual representations of geometric objects such as squares, rectangles, circles, ellipses, and so on. It is important to remember that they are representations of such objects; a raster system cannot display geometric primitives exactly

2.1 Pixels

Somehow we need to tell an image how to set the color of a pixel.

- Image_set(?)
  - Needs access to an Image
  - Which pixel? One index or two?
  - What value?

The above raise questions about how we're going to organize our code and give functions the information they need to do their job. One extreme is to make the graphics system **state based**. The other extreme is to make the system **parameter based**.

A state based graphics system maintains a set of internal state variables. State variables might include the following attributes.

- Image
- Color
- Pen width
- Pen style or pattern
- Whether to fill graphics primitives

The graphics system will begin in a default state and the programmer will have access to functions that get or set the state variables. To draw a single red point with a 2x2 size, for example, the programmer would make three function calls.

1. Set the color to red
2. Set the pen width to 2
3. Plot the point using the current graphics state

A parameter based graphics system does not maintain state variables, but relies on the programmer to keep track of all the graphics system information and pass it in as arguments to functions that draw primitives. To draw the same red point in a 2x2 size takes only a single function call in a parameter based system, but the function has to take many more arguments.

There are strengths and weaknesses to both approaches. A state based system can be cumbersome when drawing many unique objects, requiring multiple function calls for each graphics primitive. A parameter based system, however, is more cumbersome when drawing many objects that have the same attributes. Clearly, there are compromise systems that combine aspects of both. A simple improvement on a parameter based system, for example, is to collect all attributes in a single structure and just pass along a pointer to the structure every time a graphics primitive function is called.

There are commercial examples of both kinds of graphics systems. X-windows, for example, uses a parameter based system where the graphics attributes are bundled into a data structure passed along from function to function. OpenGL, on the other hand, is a state-based system where most graphics attributes are set by modifying state variables. For this class we're going to take a more parameter based approach to building a graphics system so that the attributes are more transparent and we avoid the need for global variables (for the most part).

Plotting pixels in our system is going to require that we pass in all the required information necessary to execute the operation. The first design decision we have to make is how to internally represent pixels. While many image formats use unsigned characters (8-bit integers) to represent each color, the calculations we will be making based on colors, lighting and geometry will require floating point operations and make more sense on range of [0, 1]. Therefore, internally, we will want to use a floating point value for each color channel.

In addition, we may want our pixel type to incorporate an alpha channel and a depth channel. The latter will be necessary once we begin to do 3-D hidden surface removal in our rendering system. How you organize your pixel internally does not make much difference, although using an array to represent the colors may make certain operations possible using a loop structure. Something like the following would work.

```
typedef struct {
  float rgb[3];
  float a;
  float z;
} FPixel;
```

An internal image, therefore, will consist of an array of FPixels and basic information like the number of rows and columns in the image.

```
typedef struct {
  long rows;
  long cols;
  FPixel *data;
} Image;
```

With these factors in mind, we might define the following basic functions to access pixels.

- FPixel image_get(Image *src, int r, int c);
- FPixel image_get1D(Image *src, int i);
- void image_set(Image *src, FPixel p, int r, int c);
- void image_set1D(Image *src, FPixel p, int i);

2.2 Graphics Specification Document

Each assignment this semester will be accompanied by a specification of the functions required for the assignment and the names and arguments for those functions. If your code follows the specifications, then you will be able to execute the test function(s) provided for each assignment. It also guarantees that you will be able to continue to build upon your code from week to week. While there will be some situations where you have to go back and add new features to old code, following the specification should make this straightforward.

Note that you have great latitude to go beyond the specification. The specification is a minimalist document and outlines the minimum required capabilities for your system. You are encouraged to add additional functionality throughout the semester. As long as the functions outlined in the specification exist and work properly, the test programs will run.

2.3 Lines

How do we draw lines into an image?

Line equation:

$$y = mx + b \tag{1}$$

Given a starting and end point, we can solve for the slope m and intercept b

$$m = \frac{y_1 - y_0}{x_1 - x_0} \tag{2}$$

$$b = y_0 - mx_0 \tag{3}$$

The differential equation for a line tells us how much to move in y given a unit step in x.

$$\Delta y = m\Delta x \tag{4}$$

To draw a line, we could start at one end (assume the left end, always), draw the first pixel and then take a step, updating x or y appropriately.

$$y_{k+1} = y_k + m \tag{5}$$

$$x_{k+1} = x_k + \frac{1}{m} \tag{6}$$

If the slope is less than one, use x as the loop variable. If we increment x by 1 each time, then we increment y by the slope m. If the slope is greater than 1, use y as the loop variable and increment x by $\frac{1}{m}$ each step.

How do we have to represent x and y? Given fractional values, which pixel do we light up? On older hardware, using floating point values was completely out of the question as it was 100s of times slower than integer math. Even on new hardware, if we do line calculations using integers we save hundreds of thousands of transistors on a graphics card that can be used for other tasks.

2.3.1 Bresenham's line algorithm

Bresenham's algorithm is based on analyzing how an error term changes as you move from one pixel to the next. Figure 1 shows the distances d_1 and d_2, which represent the distances from the line to the upper and lower pixel centers, respectively.

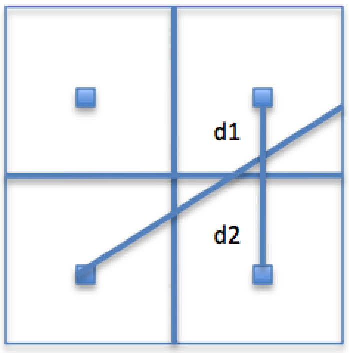


Figure 1: Distance of a line to upper pixel center d_1 and lower pixel center d_2.

If we define an error term $e = d_2 - d_1$, we can look at the sign of e to determine which pixel to light up.

- If the sign is positive, light the upper pixel.
- If the sign is negative, light the lower pixel.

Each iteration of the algorithm we step to the next pixel along the x axis, update the error term $e = e + m$ and look at the new sign of e. We never actually have to calculate $d_2 - d_1$, explicitly.

- If e is positive, then we take a step up and subtract 1 from e to account for the step.
- Initialize $e = -\frac{1}{2}$ so the first decision will be correct (consider the case of a $\frac{1}{2}$ slope).
- The first value of e we test will be $e = -\frac{1}{2} + m$

It's still not an integer algorithm!

Integer Bresenham's algorithm

A few simple modifications to the algorithm can convert it to an integer algorithm.

- Define $e' = 2\Delta x e$, so the initial value of $e' = 2\Delta x(-\frac{1}{2}) = -\Delta x$
- Represent $m = \frac{\Delta y}{\Delta x}$, both Δy and Δx are integer values, and $2\Delta x(m) = 2\Delta x \left(\frac{\Delta y}{\Delta x}\right) = 2\Delta y$
- Consider the first value of e' we test

$$e' = 2\Delta x(-\frac{1}{2} + m) = 2\Delta x(-\frac{1}{2} + \frac{\Delta y}{\Delta x}) = 2\Delta y - \Delta x \tag{7}$$

- The update rule for e' is $e'_{k+1} = e'_k + 2\Delta y$
- When $e' > 0$ we subtract $2\Delta x$.

Pseudo-code algorithm (for lines in first octant going left to right)

```
x = x0
y = y0
dx = x1 - x0
dy = y1 - y0
e = 2 * dy - dx

for(i = 0; i<dx; i++) {
  Image_set(im, color, y, x)
  while( e > 0) {
    y = y + 1
    e = e - 2 * dx
  }
  x = x + 1
  e = e + 2 * dy
}
```

Example: Drawing a line

Let $(x_0, y_0 = (20, 10)$ and $(x_1, y_1) = (30, 18)$, so $(\Delta x, \Delta y) = (10, 8)$.

i	draw pixel	e_{k+1}	x_{k+1}	y_{k+1}
Initial	n/a	$6 = 2 * 8 - 10$	20	10
0	(20, 10)	2	21	11
1	(21, 11)	-2	22	12
2	(22, 12)	14	23	12
3	(23, 12)	10	24	13
4	(24, 13)	6	25	14
5	(25, 14)	2	26	15
6	(26, 15)	-2	27	16
7	(27, 16)	14	28	16
8	(28, 16)	10	29	17
9	(29, 17)	6	30	18

To generalize Bresenham's algorithm always draw the line from bottom to top and consider the cases of the first four octants.

- First octant uses the algorithm as specified above.
- Second octant flips the roles of x and y.
- Third octant is like the second, but steps left in x instead of right.
- Fourth octant is like the first, but steps left in x instead of right.
- Horizontal and vertical lines must be handled as special cases.

Faster line algorithms

- Can take advantage of symmetry and draw in from both ends simultaneously
- Can take multiple steps at a time. There are only four possible arrangements of 2 pixels in a line.

2.4 Screen coordinate systems

Lines, polygons, circles, and ellipses are mathematic objects with definitions in a continuous world. In the absence of a vector display, we have to render these continuous objects on a discrete raster display. When we display graphics primitives, we would like to have attributes like length and area be the same in their rendered state as they are in their theoretical state.

Consider the example above. If we consider the coordinate grid to be located at pixel centers, then we probably want to light up both the first and last pixel on the line: (20, 10) and (30, 18). However, if we do so, the line is no longer the correct length in its visible representation. If we stop one before the last pixel, the line is the correct length, but the choice of whether to light up the first or last pixel seems arbitrary.

What if we consider the coordinate system to correspond to the left left corner of a pixel?

- Makes sense to not color in the last pixel
- Need to be careful how we draw lines
- Need to think about any impact this has on Bresenham's line algorithm

2.4.1 Updating Bresenham's algorithm

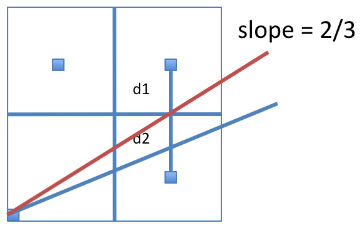


Figure 2: Distance of a line to upper pixel center d_1 and lower pixel center d_2 given coordinates aligned with the lower left corner.

If we move the coordinate system to the lower left corner, then the line slides down and left by half a pixel. The decision to move up or not is still the same: look at whether the line is above or below the pixel boundary at the halfway point.

- Original initial testing point: $e'_0 = 2\Delta x(\frac{\Delta y}{\Delta x} - \frac{1}{2}) = 2\Delta y - \Delta x$
- Modified initial testing point is down by half a pixel $-\frac{1}{2}$ and left by half a pixel $+\frac{1}{2}\frac{\Delta y}{\Delta x}$:

$$e'_0 = 2\Delta x\left(\frac{\Delta y}{\Delta x} - \frac{1}{2} + \left(\frac{\Delta y}{2\Delta x} - \frac{1}{2}\right)\right) = 2\Delta x\left(\frac{3\Delta y}{2\Delta x} - 1\right) = 3\Delta y - 2\Delta x \tag{8}$$

Since the increments to the error are based on unit steps, and we're still doing unit steps, the incremental values do not change when using a lower-left corner origin for the coordinate system.

How does this affect the lines? Consider where the cutoff points are for stepping up or across.

- $[0, \frac{2}{3}]$: step horizontally
- $(\frac{2}{3}, \frac{3}{2})$: step diagonally

- $[\frac{3}{2}, \infty]$: step up

What about horizontal and vertical lines? The concept of line direction becomes important, especially when drawing rectangles. One convention is to make horizontal and vertical lines rotated versions of one another.

- Draw horizontal lines going right in the first quadrant
- Draw vertical lines going up in the second quadrant
- Draw horizontal lines going left in the third quadrant
- Draw vertical lines going down in the fourth quadrant

If you draw closed polygons using horizontal and vertical lines in a counterclockwise order using the convention above, then the area enclosed by the lines is mathematically correct. Drawing the same set of points in the opposite order results in missing corners. If you try to draw all vertical lines in the first quadrant and all horizontal lines in the first quadrant, then there isn't any ordering of the corners that produces a mathematically correct solution: you have to draw to different points.

Programs like Adobe Photoshop take into account where the user clicks in the image when drawing lines. If you are zoomed in, Photoshop actually adapts the starting location of the line to a subpixel location. Clicking near the lower left corner of a pixel produces a line that is identical to one produced by Bresenham's with a lower left starting point.

2.5 Polygons and polylines

A **Polyline** is simply a collection of connected line segments where the end of one line is the beginning of the next. Polylines do not connect the last point to the first. A polyline function will take in a list of points and graphics attributes and draw the line segments.

A **Polygon** is a collection of connected line segments where the last point is connected back to the first point by a line. A polygon function takes the same arguments as the polyline function and automatically adds the last connecting line segment.

Because polygons and polylines can possess any number of line segments, these are dynamic data structures, unlike lines, points, circles, or ellipses, which have fixed size representations.

These is no necessary restriction to polylines or polygons containing lines that cross. As polylines are not filled objects, crossing lines are acceptable and do not create any real problems. However, polygons are filled objects. when a polygon contains crossed lines, the question of which parts of the image are inside the polygon and which are without becomes relevant. In general, this question is answered by determining how many lines you have to cross in order to reach a particular pixel. If it takes an odd number of crossings to reach a pixel, then it is inside the polygon, otherwise it is outside.

2.6 Circles and Ellipses

How can we draw a circle into an image?

Algorithm 1: cartesian circle equation

- Circle equation: $r^2 = (x - x_c)^2 + (y - y_c)^2$
- Solving for y
$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \tag{9}$$
- Step across in x, calculating the next y.
- Switch orientations half-way through.

Problems:

- Non-uniform spacing
- Get points along the circle, not a complete, closed shape
- Square roots are expensive

Algorithm 2: polar circle equation

We can improve the spacing by using an angular formulation of a circle.

$$\begin{aligned} x &= x_c + r\cos(\theta) \\ y &= y_c + r\sin(\theta) \end{aligned} \tag{10}$$

If you walk around the circle by modifying (θ) in steps of $\frac{1}{r}$ you can get unit steps. However, it makes use of trigonometric functions, which makes it costly.

Algorithm 3: Bresenham's Circle Algorithm

The main idea of the integer circle algorithm is to use the cartesian circle function to test whether to step across or down by evaluating the relative position of the two possible next pixels to the circle boundary. We can do this in a single test by evaluating the circle equation on a point that is mid-way between the pixels.

- $f(x, y) = x^2 + y^2 - r^2$
- If (x, y) is our test point, then the sign of $f(x, y)$ tells us if the boundary is inside the circle (< 0) on the circle ($= 0$) or outside the circle (> 0).

We actually know the first point to draw. Relative to the origin (x_c, y_c), the first point is one of the four pixels horizontally or vertically from the center. Now we just need to calculate the next location to test. Ideally, we'd like to just keep track of an error term, update the error term with each step, and choose which pixel to draw next based on the error term. Note that circles are radially symmetric, so we only need to figure out the points in one octant.

Derivation:

- The last point drawn is (x_k, y_k)
- The next test location is $(x_k + 1, y_k - 1/2)$

- The next value to test is
  $p_k = f(x_k + 1, y_k - 1/2) = (x_k + 1)^2 + (y_k - 1/2)^2 - r^2$
  $p_k = x_k^2 + 2x_k + 1 + y_k^2 + y_k + 1/4 - r^2$
  $p_k = x_k^2 + y_k^2 + 2x_k + y_k + 5/4 - r^2$
- The value to test on the next iteration is
  $p_{k+1} = f(x_{k+1} + 1, y_{k+1} - 1/2) = (x_{k+1} + 1)^2 + (y_{k+1} - 1/2)^2 - r^2$
  $p_{k+1} = x_{k+1}^2 + 2x_{k+1} + 1 + y_{k+1}^2 + y_{k+1} + 1/4 - r^2$
  $p_{k+1} = x_{k+1}^2 + y_{k+1}^2 + 2x_{k+1} + y_{k+1} + 5/4 - r^2$
  $p_{k+1} = (x_k + 1)^2 + y_{k+1}^2 + 2(x_k + 1) + y_{k+1} + 5/4 - r^2$
  $p_{k+1} = x_k^2 + y_{k+1}^2 + 2(x_k + 1) + 2x_k + y_{k+1} + 1 + 5/4 - r^2$
- Subtract p_k from p_{k+1} to get the update rule

$$p_{k+1} = p_k + 2[x_k + 1] + [y_{k+1}^2 - y_k^2] - [y_{k+1} - y_k] + 1 \tag{11}$$

- y_{k+1} is either y_k or $y_k - 1$

Consider that the second and third terms involve a difference of y values, which will be either 0 if no step was taken, or a known increment if a step was taken. How we update p depends upon whether we stepped across ($y_{k+1} = y_k$) or stepped down ($y_{k+1} = y_k + 1$). In all cases, we know that $x_{k+1} = x_k + 1$.

- If p_k is negative: $p_{k+1} = p_k + 2x_{k+1} + 1$
- if p_k is non-negative: $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$

Note that all of the updates are integral, so there is no need for floating point values. The only exception is that the initial starting value for p is $5/4 - r$, but rounding the expression to $1 - r$ works just fine.

The algorithm for drawing a circle, then, is to begin at one of the axis extrema and then iterate over one octant, reflecting each drawn pixel to the other eight octants. At each iteration test the value of p_k and step either across or down, updated the value of p_k appropriately given the rules above.

Ellipses

The ellipse algorithm is almost identical to the circle algorithm except that you have to draw a full quadrant of the ellipse and change the step variable at some point in the process. The quadrant divides where the slope is -1, or the first pixel where the change in x is less than the change in y. Otherwise, the process is the same: take a step, update an error term, and reflect the points across to the other four quadrants.

2.6.1 Coordinate System Issues

Like lines, we have to be careful about how to draw ellipses and circles or they end up being too big. The solution for both primitives is to make all calculations in the third quadrant and reflect the points around to the others using the diagram in figure 3. For the circle algorithm, this modifies the initial values of x and y and modifies the update rule because the x values are negated. Note that all the x and y values are relative to the circle's center, which is used only when actually plotting the pixels.

- Initial value of $x, y = -1, -r$
- Initial value of $p = 1 - r$
- Decrement x each iteration
- Increment y each iteration (moving up and left in the third quadrant)
- Iterate until $x > y$
- Update rule if $p < 0$: $p_{k+1} = p_k + 1 - 2x_{k+1}$, where $x_{k+1} = x_k - 1$
- Update rule if $p \geq 0$: $p_{k+1} = p_k + 1 - 2(x_{k+1} - y_{k+1})$ where $y_{k+1} = y_k + 1$ and $x_{k+1} = x_k - 1$
- When reflecting, subtract one from the reflected value any time the x or y coordinate is negated

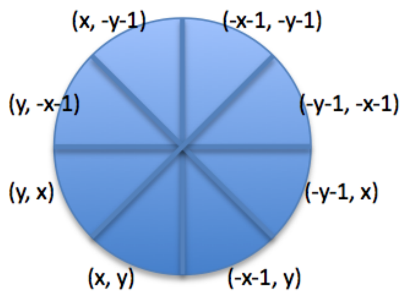


Figure 3: Reflecting points around the octants

Corrected Circle Algorithm

```
Parameters: cy, cx, radius
set x to -1
set y to -radius (rounded)
set p to 1 - radius (rounded)
Plot (x + cx, y + cy) and reflect it around the circle

while x > y
  x--
  if p < 0
    p = p + 1 - 2 * x
  else
    y++
    p = p + 1 - 2 * (x - y)
  plot ( x + cx, y + cy ) and its reflections
```

For ellipses, you get the following modifications, which are meaningful in the context of the example code provided with the lab.

- Initial value of $x = -1$
- Initial value of $y = -R_y$ (y-axis radius)
- Initial value of $p_x = 2R_y^2$
- Adjust the initial value of p in section 1 by adding $R_y^2 + p_x$
- Adjust the initial value of p in section 2 by adding $R_x^2 - p_y$
- As with the circle, when reflecting around, subtract 1 from a coordinate whenever it is negated.

Corrected Ellipse Algorithm

```
Parameters: cx, cy, Rx, Ry
Initial value of x = -1
Initial value of y = -Ry
Initial value of px = 2 * Ry * Ry
Initial value of py = 2 * Rx * Rx * -y

Plot (x + cx, y + cy) and its four reflections
p = Ry * Ry - Rx * Rx * Ry + Rx*Rx/4 + Ry*Ry + px

while px < py
  x--
  px = px + 2 * Ry * Ry
  if p < 0
    p = p + Ry*Ry + px
  else
    y++
    py = py - 2 * Rx * Rx
    p = p + Ry * Ry + px - py
  Plot(x + cx, y + cy) and its four reflections

p = Ry*Ry * (x*x + x) + Rx*Rx * (y*y - 2*y + 1) -Rx*Rx * Ry*Ry+ Rx*Rx - py

while y < 0
  y++
  py = py - 2 * Rx * Rx
  if p > 0
    p = p + Rx * Rx - py
  else
    x--
    px = px + 2 * Ry * Ry
    p = p + Rx * Rx - py + px
  Plot (x + cx, y + cy) and its four reflections
```

2.6.2 Filled Circles and Ellipses

Fill circles and ellipses is trivial

- For each horizontal pair of points, fill in the row from left to right.
- For each point you calculate for a circle, you get four lines
- For each point you calculate for an ellipse, you get two lines

2.6.3 Why Primitives are Important

Trying to draw simple graphics primitives quickly and correctly raises many issues that are essential to graphics.

1. Fast algorithms use differential methods and avoid computing (x, y) from scratch
2. Deriving differential algorithms involves the following process
    - identify the initial point
    - identify a test location
    - calculate the error at iteration i
    - calculate the error at iteration $i + 1$ in terms of the variables at step i
    - subtract the two error terms to identify the update rule for the error value
3. Integer algorithms exist for most basic primitives
4. Picking the right coordinate frame is important
    - impacts the geometric qualities of the displayed object (area, perimeter, length)
5. Trying to render continuous mathematical objects on a discrete screen results in aliasing artifacts

2.7 Generic Fill Algorithms

Often we want to fill arbitrary shapes or polygons. The two major approaches are flood-fill and scan-fill. Flood-fill algorithms start at a seed point inside the shape or area and grow out from the seed until they hit a boundary. Scan-fill algorithms begin at the top of the shape and work down row by row, figuring out which pixels to color on each row. Flood fill algorithms are quick and easy to implement and work well for interacting drawing programs. Scan-fill algorithms are more appropriate for automatic rendering. Scan-fill algorithms also provide a framework for doing many more complex tasks while drawing the shape.

2.7.1 Flood-fill Algorithms

The **flood fill** algorithm is implemented as the paint can tool in most drawing programs. The idea is to fill in an area surrounded by a boundary, starting at a seed pixel and growing outwards. The flood-fill algorithm is straightforward to implement using a stack.

```
Make the boundary using a selected color C
Pick an initial point
Color the initial point
Push the initial point on a stack

While the stack is not empty
  Pop a pixel P off the stack
  For each neighbor N of P
    If N is not already colored
      Color N
      Push N on the stack
```

There are a number of variations to the algorithm.

- If you want the boundary to be a different color than the interior, modify the if-statement so it doesn't color the pixel N if its existing color is the boundary color or the fill color.
- If you want the new color to fill over only one existing color C_{old}, modify the if-statement so that it colors pixel N only if the existing color is C_{old}

A variation on the flood-fill algorithm is the **scanline flood fill** algorithm that is a bit faster and generally uses less memory. The idea behind the scanline flood-fill is to fill in all of the contiguous pixels on a row before moving up or down to the next row.

- Fill in the scanline from left to right
- At each pixel, look at the row above and the row below to see if a seed needs to be pushed onto the stack
- Seeds need to be placed any time the scan hits the end of a contiguous span of pixels to be colored

Figure 4 shows the sequence of seeds for a polygonal shape with some concave boundaries.

Algorithm

```
Make the boundary using a selected color C
Pick an initial point (uppermost, leftmost pixel to be colored)
Color the initial point
Push the initial point on the stack

While the stack is not empty
  Pop a pixel P off the stack
  Find the leftmost pixel in the scanline that should be colored
  For each pixel Q from the leftmost to the rightmost fill location on the scanline
    Color Q
    If the pixel above Q should be colored and
    the pixel up and right from Q is on the boundary
      Push the pixel above Q onto the stack
    If the pixel below Q should be colored and
    the pixel down and right from Q is on the boundary
      Push the pixel below Q onto the stack
    If the pixel below the last pixel on the scanline should be colored
      Push the pixel below onto the stack
    If the pixel above the last pixel on the scanline should be colored
      Push the pixel above onto the stack
```

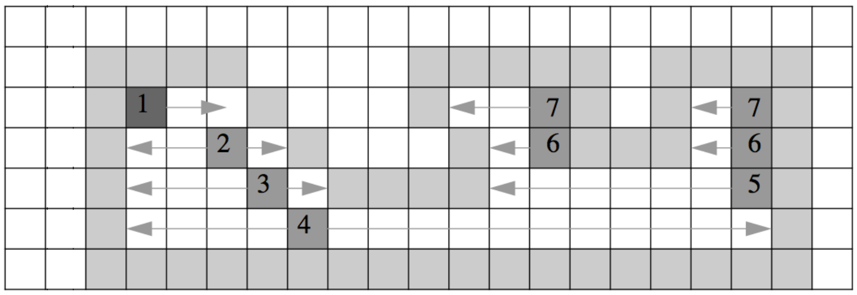


Figure 4: Diagram of scanline flood fill showing all of the seeds generated in the process. The initial seed is #1. Note the two #6 seeds because the scanline is broken by the interior boundary

2.7.2 Connectedness

We want outlines on a raster display to form a connected series of pixels. But the definition of connectedness is not always clear. Are only NSEW neighbors connected, or are diagonal pixels also conected?

4-connected: only the NSEW neighbors of a pixel are considered to be connected

8-connected: all eight neighbors of a pixel are considered to be connected

Note that the connectedness of an outline influences the definition of neighbors while filling. If a boundary is 4-connected, then you want to fill using an 8-connected definition. If a boundary is 8-connected, then you want to fill using a 4-connected definition. If you don't use a fill connectedness that is the opposite from the boundary connectedness, then the fill either will not be complete or will bleed through the boundary.

2.7.3 Beyond Single Colors

There are many ways to draw a line

- Single color
- Gradient
- Dashed line
- Transparent line

Likewise, there are many different ways to fill a shape.

- Single color
- Gradient fill
- Pattern fill
- Transparent fill
- Texture fill

How can we implement these different methods of rendering shapes?

Gradient fill

To implement a gradient fill we need two colors–C_a and C_b–and a method of choosing, for each pixel, what percentage of each color to use. Then we can use alpha blending to create the color to draw onto the screen.

$$C_s = \alpha C_a + (1.0 - \alpha) C_b \tag{12}$$

For drawing a line that blends evenly from one color to the other, we just need to keep track of the change in α as we draw the line. If the initial value of α is 0, then the final value should be 1. In the first octant, the number of steps is given by dx, so the update equation for alpha is given by the following.

$$\alpha_{i+1} = \alpha_i + \frac{1}{dx} \tag{13}$$

It's possible to execute alpha blending in a completely integral fashion if you multiply α by dx and adjust the blending equation accordingly.

For drawing a shape like a circle or ellipse that blends evenly from one color to another, there are many more options. The simplest is to pick a horizontal or vertical orientation for the gradient. Then α begins at 0 on one side of the shape and goes to 1 on the other side. However, there are many other possible blending strategies. For example, you may want the center of the circle to be color C_b and the outside to be color C_a. In that case, α is related to the radial distance of the pixel from the center of the circle.

$$\alpha_{x,y} = \frac{\sqrt{(x - x_c)^2 + (y - y_c)^2}}{r} \tag{14}$$

In fact, you can make any relationship you wish between α and location within the shape, including picking α from a random distribution. The key is to develop a set of parameters for drawing gradient fill that gives the programmer/user flexibility without making the function unwieldy.

Pattern fill

Most graphics programs give you the option of selecting a repeating pattern or texture for drawing filled objects. The patterns are generally modifications of the foreground color (or you can think of them as patterns of alpha-blending. The textures are generally designed so they flow seamlessly from one tile to the next in any direction.

At each pixel, we just need to know what part of the pattern to draw. There is a one-to-one correspondence between the patterns or textures and pixels, so there is no sampling issue, we just need to know which pixel to grab from the pattern.

The simplest method of picking a pattern pixel is to use modulo arithmetic. If the pattern is 16×16, then if you take the coordinates of the pixel you want to draw modulo 16, you get an index into the pattern. The only issue with using the raw coordinate is that if you draw the graphics primitive in a different location, the pattern may be different with respect to the object's origin. As the object moves across the screen, the pattern stays constant with respect to the image coordinate system.

A simple modification solves the problem. Simply subtract off the object's origin (often the upper left-most pixel of the bounding box) from the pixel coordinates of interest before executing the modulo operator.

$$\begin{aligned} x_{\text{pattern}} &= (x_{\text{image}} - x_{\text{origin}}) \mod \text{PatternSize}_x \\ y_{\text{pattern}} &= (y_{\text{image}} - y_{\text{origin}}) \mod \text{PatternSize}_y \end{aligned} \tag{15}$$

Transparent fill

To fill a shape transparently is simply a variation on alpha blending. To have a uniform transparent fill, select α and then blend the foreground fill color C_f with the existing color in the image C_i.

$$C = \alpha C_f + (1.0 - \alpha) C_i \tag{16}$$

You can use the same kinds of functions to determine α for transparent filling as you do for gradient fill. The only difference is that instead of specifying a background color, use the current image values.

Texture fill

Texture fill is a little more complicated because the texture is likely to be some arbitrary image, and the object you want to fill is not likely to have the same shape. The key to texture fill is finding a mapping from the pixel you want to draw to the pixel in the texture you want to use.

For example, if we want to fill a circle with some part of an image, we need to find a mapping from a pixel on the circle to the texture image. A simple way to execute the mapping is to enforce a 1-1 mapping from image to texture and match the bounding box of the circle, defined by $(x_c - r, y_c - r)$ and $(x_c + r, y_c + r)$ to the upper left corner of the texture image. The coordinates in the texture image (i_t, j_t) would correspond to pixel (i, j) in the circle as in (17).

$$\begin{aligned} i_t &= i - (y_c - r) \\ j_t &= j - (x_c - r) \end{aligned} \tag{17}$$

If we let the user pick an origin for the box in the texture image, then we need to add the offset of the box (t_x, t_y) in the texture image relative to the upper left corner.

$$\begin{aligned} i_t &= i - (y_c - r) + t_y \\ j_t &= j - (x_c - r) + t_x \end{aligned} \tag{18}$$

What if we let the user pick any axis-oriented non-zero box to map onto the circle? The process of mapping from circle to texture image then becomes: align the coordinate systems, scale the coordinate systems, add the offset. If we let the box in the texture image be defined by the point pair (p_{0x}, p_{0y}) and (p_{1x}, p_{1y}), then the scale factors are the texture box width and height divided by circle bounding box size $2r$.

$$\begin{aligned} s_x &= \frac{p_{1x} - p_{0x}}{2r} \\ s_y &= \frac{p_{1y} - p_{0y}}{2r} \\ \\ i_t &= [i - (y_c - r)] * s_y + t_y \\ j_t &= [j - (x_c - r)] * s_x + t_x \end{aligned} \tag{19}$$

The relationship in (19) provides a method for putting any part of an image into the circle at a reasonable scale.

Now what if we let the user arbitrarily pick four points to define a mapping from the texture images to the circle? It becomes much more difficult to map a square onto an arbitrary set of four unique points if we try to do it geometrically. However, we can cheat.

2.7.4 Interpolation

Given a circle, its bounding box is a square in the image.

- Pick four points that define a quadrilateral in the texture image
- Assign the four points to the four points of the square
- To discover the texture coordinate of a pixel in the circle, use bilinear interpolation.

Interpolation is a method of smoothly varying a value from one state to another. Linear interpolation makes the rate of change constant. Bilinear interpolation is useful on 2D surfaces like images, because it makes use of four endpoints instead of two.

The process for bilinear interpolation is as follows.

- Given four corners A, B, C, D and a location (i, j) in between them
- Based on the column j, mix the values of corners A and B, put the result in E
- Based on the column j, mix the values of corners C and D, put the result in F
- Based on the row i, mix the values of E and F

The end result is a texture coordinate for the pixel (i, j).

What if the pixel within the circle covers many pixels in the texture image?

- Identify the texture coordinates of the four corners of the pixel: $(i, j), (i+1, j), (i, j+1), (i+1, j+1)$
- Calculate the enclosing bounding box of the four texture coordinates
- Average the pixel values within the bounding box

2.7.5 Scanline Fill Algorithm

The scanline fill algorithm is one of the most commonly used algorithms in computer graphics. Its use goes far beyond simply filling polygons with a single color. The basic concept is the underlying algorithm for many rendering systems.

The basic concept is simple:

- For each scanline, intersect the scanline with all the edges of the polygon
- Sort the edges left to right and fill between the edge pairs

Problems can arise, however, because some vertices require special handling

- Some vertices need to be counted twice
- Some vertices need to be counted once

It turns out the the property differentiating the two cases is whether the vertex is an extremum or not. Vertices that are also vertical extrema need to be counted twice. Vertices that are not extrema should only be counted once. Note that, in the case of non-extrema vertices, we could also just shorten one of the edges by one pixel.

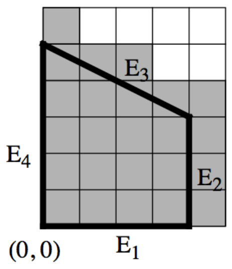
Rather than calculating the intersections of the polygon edges with the scanlines explicitly, we can take advantage of **coherence** since the edges of the polygon are lines. The slopes don't change. Therefore, we can just keep track of the current boundary point for each edge that intersects the current scanline and update those in fixed increments as we move to the next scanline. Then the major issue is just handling which edges intersect the current scanline.

We could use Bresenham's algorithm to calculate the boundary pixels, but it turns out there is no advantage to not using floating point math for polygons: most of the time we're trying to render polygons that are mappings of 3D objects onto a 3D plane and we want to be as accurate as possibly with our placement.

Ordered Edge List Algorithm

1. Process a polygon by going through the vertices in order
2. For each non-horizontal edge
    - Calculate edge properties (e.g. inverse slope, start point, end point)
    - Insert the edge into an ordered edge list according to starting scanline
    - Set the endpoint properly, shortening it by one if it is not an extremum endpoint.
    - Keep track of the starting scanline
3. For each scanline until the edge list is empty
    - Put any edges that start on this scanline into the active edge list (keep it sorted)
    - Fill the scanline from left to right, pairing up edges
    - Remove edges that terminate on this scanline
    - Update the active edges by incrementing their intersections
    - Sort the active edge list using the new intersection values

Polygon Example



Edges:

- E_1: (0, 0) (4, 0)
- E_2: (4, 0) (4, 3) [modified to (4, 0) (4, 2) in preprocessing step]
- E_3: (4, 3) (0, 5)
- E_4: (0, 5) (0, 0)

Ordered edge list: HEAD $\rightarrow E_4(0,4) \rightarrow E_2(0,2) \rightarrow E_3(3,4) \rightarrow$ NULL

Active Edge List:

- Scanlines $\{0, 1, 2\}$: HEAD $\rightarrow E_4(0,4) \rightarrow E_2(0,2) \rightarrow$ NULL
- Scanlines $\{3, 4\}$: HEAD $\rightarrow E_4(0,4) \rightarrow E_3(3,4) \rightarrow$ NULL

Going through the process from scanline 0 to 5 draws the shape above. Shortening E_2 solves the double-counting problem, but the polygon is way too big.

2.7.6 Matching polygons to the grid

The following modifications repair the algorithm so it creates polygons with the correct area, if they are integer coordinates.

- Shorten the top edge of all edges by 1 (not just non-extrema)
- Fill up to the right pixel, but don't color the last one
- Add $\frac{1}{2}\frac{dx}{dy}$ to the initial intersection for integer coordinates.

For non-integer coordinates, it is important to account for the exact location of the vertex within the pixel. The general rule is that the x-intersection needs to be initialized to the intersection of the edge with the first scanline it crosses. All edges start at their lower y-coordinate and move up (horizontal lines are ignored). If V_y is the smaller Y coordinate, the following cases determine the modifier to the initial x-intersect.

- Case $V_y - floor(V_y) \leq 0.5$: add $(0.5 - (floor(V_y) - V_y))\frac{dx}{dy}$
- Case$V_y - floor(V_y) > 0.5$: add $((1.0 - (floor(V_y) - V_y)) + 0.5)\frac{dx}{dy}$

2.8 Interpolation and Barycentric Coordinates

The scanline fill algorithms provides a useful framework for not only drawing a polygon, but also shading it, texturing it, or otherwise modifying it based on the properties of its vertices. If, for example, we know the color of the polygon at each vertex, then we can interpolate the color along each edge and then interpolate the color along each scanline.

Interpolation is the process of smoothly blending from one value to another across some distance. In this case, the distance is across pixels. In general, if you have a value C_A at location x_0 and a value C_B at location x_1, then you can use a variation of the alpha blending equation to implement linear interpolation from C_A to C_B across the distance from x_0 to x_1. At each location x, the new color $C(x)$ is defined as in (20).

$$C(x) = C_A \left(1 - \frac{x - x_0}{x_1 - x_0}\right) + C_B \left(\frac{x - x_0}{x_1 - x_0}\right) \tag{20}$$

If we are using linear interpolation, then the change in C for a fixed size step in x is constant.

It is also possible to do interpolation across a plane as a weighted combination of three vertices A, B, and C. The weights for the vertices A, B, C are given by the barycentric coordinates (α, β, γ), which we can compute for any (x, y). The barycentric coordinates must obey the relationships in (21), where $P_{x,y}$ is some point on the plane.

$$\begin{gathered} \alpha A + \beta B + \gamma C = P_{x,y} \\ \alpha + \beta + \gamma = 1 \end{gathered} \tag{21}$$

Conceptually, barycentric coordinates are defined by a possibly non-orthogonal coordinate system. In practice, we tend to use three points–the vertices of a triangle–and are mostly concerned about the coordinates of pixels inside the triangle. Given three vertices, (A, B, C), the barycentric coordinates are defined by the perpendicular distance from the line opposite the vertex. The distance from line CA is the coordinate for point B; the distance from line AB is the coordinate for point C, and the distance from line BC is the coordinate for point A. Note the orientation of the line segments: the traversal of the vertices is in a counter-clockwise direction.

We can use the implicit algebraic equation for a line to calculate the barycentric coordinates. The implicit equation for a line generated by two points (x_0, y_0), (x_1, y_1), is given by (22).

$$f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 \tag{22}$$

Using the implicit equation, we can calculate β and γ and then use (21) to compute α.

$$\begin{aligned} \beta &= \frac{f_{ca}(x, y)}{f_{ca}(x_b, y_b)} = \frac{(y_c - y_a)x + (x_a - x_c)y + x_c y_a - x_a y_c}{(y_c - y_a)x_b + (x_a - x_c)y_b + x_c y_a - x_a y_c} \\ \gamma &= \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)} = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a} \\ \alpha &= 1 - \beta - \gamma \end{aligned} \tag{23}$$

2.9 Anti-aliasing

Aliasing is caused by sampling a continuous world at too low a frequency. Aliasing can occur in many different forms.

- Spatial aliasing: trying to represent continuous objects using a discrete grid
- Time aliasing: trying to represent continuous motion using a discrete time step
- Spectral aliasing: trying to represent continuous spectra using discrete color bands

A good graphics system must deal with all three (the third we don't tend to notice so much because the sampling frequency is high enough for most situations).

Spatial aliasing is most relevant for object primitives. There are a number of methods for improving the appearance of objects.

2.9.1 Supersampling

The basic concept of supersampling is to render the object onto a much larger temporary image and use the higher resolution rendering to calculate how to draw the object into the original image.

- Use 9 or 16 subpixels per final pixel
- Draw the object scaled up appropriately
- Count how many subpixels are on within each final pixel to draw into the final image
- Alpha blend when putting the object into the final image

You can also draw the entire scene in an image that is large and then scale it down. When working with polygons, this is fine and works well. For lines, however, unless you draw the lines several pixels thick in the high resolution image, they will almost disappear in the final image.

One problem with supersampling single lines is that a line can intersect at most 3 subpixels within a superpixel. Therefore, you have only four levels of coloring $[0, \frac{1}{3}, \frac{2}{3}, 1]$.

2.9.2 Area Sampling

The concept of area sampling is to use an expanded canvas, but draw the line with appropriate thickness. If you're using a 3x3 grid, for example, then draw a 3-pixel wide line.

- Keep track of the upper and lower edges of the line using Bresenham's algorithm (left and right for 2nd and 3rd octants)
- Fill between the lines
- Add a little bit for the ends

The distance between the upper and lower lines depends upon the slope, as shown in figure 5.

$$y_{\text{upper}} = mx + b + \tfrac{1}{2}\sqrt{m^2+1}$$
$$y_{\text{lower}} = mx + b - \tfrac{1}{2}\sqrt{m^2+1} \qquad (24)$$

The benefit of using thick lines and then supersampling is that the lines don't disappear. Because all of the subpixels in a pixel could be filled, you get the full benefit of increasing the size of the image. When you supersample at 3x3, you get 10 levels of brightness.

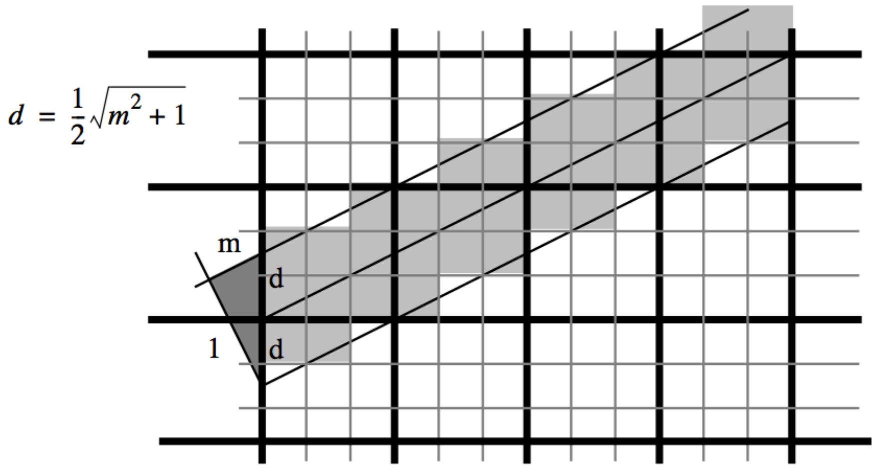


Figure 5: A solid line in an upsampled image

2.9.3 Error Terms for Antialiasing

How about using the error term in Bresenham's algorithm to antialias a line?

- The error term is $2dx$ times the difference in the distances between pixel centers
- If the error term is in the range [0, 1.0] then the fraction to color the top pixel is $\alpha = 1 - \frac{e}{2}$
- If the error term is in the range [-1.0, 0) then color the top pixel $\alpha = \frac{1}{2} + \frac{e}{2}$
- If the error term is in the range [-2.0, -1.0) then color the pixel below $\alpha = -\frac{e}{2} - \frac{1}{2}$
- The fraction for the current pixel is always $f = 1 - \alpha$.

Since the scanline-fill algorithm maintains the x value of the intersection with the current scanline and the slope, you have all the information you need to properly color the current pixel of a polygon–use the fraction of the pixel covered by the polygon. Note that, to properly anti-alias, you must make the calculation for each pixel, and only pixels completely below the starting or ending edge should be fully lit.

When rendering a scene with a polygonal mesh, you probably don't want to anti-alias individual polygons. Supersample instead.

2.9.4 Post-filtering

If you draw your scene in an image that is much larger than the final one (using appropriate width lines), then you can apply a post-filtering process to reduce the size of the image. A post-filter is generally a form of **convolution**. Convolution is simply the set of rules for how to take an image A and filter it with another image B. Generally, B is a small image that represents some operation we want to execute on A.

The algorithm for convolution of image A by image B is as follows.

```
Create an output image Z the size of A.

For each pixel (i, j) in A
  Place B so its center is over pixel (i, j)
  Initialize a sum variable S to 0

  For each pixel (x, y) in B
    Multiply pixel (x, y) and its corresponding pixel in A
    Add the product to S
  Assign S to pixel (i, j) of Z
```

It is important to note that, technically, B should be flipped horizontally and vertically before running the algorithm above. For any filter with horizontal and vertical symmetry, however, it makes no difference.

Common filters for anti-aliasing are box filters and Gaussian filters.

- Box filter - the values of the filter image B are all the same and sum to 1 (simple averaging)
- Gaussian filter - the values of the filter image B are calculated using a 2D Gaussian distribution (normal distribution)

Real applications (including operating systems) draw anti-aliased lines. The jaggies are bad.

2.10 Cells, Icons, Sprites, and Characters

Cell arrays are small 2-D arrays of pixels. In addition to colors, they can hold an alpha map.

- Icons: generally square 8x8, 16x16, or 32x32
- Sprites: arbitrarily scaled rectangles, generally with an alpha map, often used for 2D characters in games
- Characters: fonts used to be represented by cell arrays containing a binary mask

Most graphics systems give you lots of options for plotting cell arrays on a screen.

- Source copy: paste it over the current image content
- Source mix: use a single alpha value (in addition to any inherent alpha map)
- Source invert: use the map to invert the current image content
- Logical operations: can do bitwise AND, OR, NOT, and XOR operations

3 2-D Transformations and Viewing

When describing a scene or animation, we want to be able to describe transformations to an object that cause it to appear a certain way. Common transformations include translation, rotation, and scaling.

If we want to translate an object around the screen, we have three options.

1. Change the coordinates by hand
2. Add an offset to the current coordinates $\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$
3. Develop a general system of representing and applying transformations

If we want to rotate an object around the screen, we have three options.

1. Change the coordinates by hand
2. Transform the points along the path of a circle $\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} x_0 \cos\theta - y_0 \sin\theta \\ x_0 \sin\theta + y_0 \cos\theta \end{bmatrix}$
3. Develop a general system of representing and applying transformations

If we want to scale an object, we have three options.

1. Change the coordinates by hand
2. Multiply each point by a scalar value $\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} x_0 s_x \\ y_0 s_y \end{bmatrix}$
3. Develop a general system of representing and applying transformations

While option B works fine, option C is much easier and permits easy manipulation and mixing of transformations.

3.1 Matrix Representations

Homogeneous coordinates are the basis for building a generic manipulation system. The homogeneous coordinates for a 2-D Cartesian point (x_c, y_c) are (x, y, h). The relationship between the homogeneous and standard coordinates is given in 25.

$$\begin{aligned} x_c &= \tfrac{x}{h} \\ y_c &= \tfrac{y}{h} \end{aligned} \tag{25}$$

Note that this permits many representations of a single 2-D Cartesian point. In the standard, or normalized form of homogeneous coordinates $h = 1$. Using homogeneous coordinates lets us implement transformations such as translation, rotation, and scaling using a single matrix representation.

Translation

$$\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 + t_x \\ y_0 + t_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \tag{26}$$

Rotation (about Z-axis)

$$\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \cos\theta - y_0 \sin\theta \\ x_0 \sin\theta + y_0 \cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \tag{27}$$

Scaling

$$\begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 s_x \\ y_0 s_y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \tag{28}$$

Because each 2-D transformation is represented by a 3x3 matrix, we can use matrix multiplication and manipulation to concatenate transformations. Since we use column vectors for our coordinates, the order of operation moves right to left. The right-most transformation executes first.

- Order doesn't matter when concatenating only translations, only rotations, or only scales.
- Order matters when you mix different transformation type.

By concatenating a series of transformations, we can do things like rotate or scale an object around its center.

Rotating around a pivot

1. Translate from the pivot point (x_p, y_p) to the origin
2. Rotate by θ around the origin
3. Translate back to the pivot point.

$$\begin{aligned} R_{x_p,y_p}(\theta) &= T(x_p, y_p) R_z(\theta) T(-x_p, -y_p) \\ &= \begin{bmatrix} 1 & 0 & x_p \\ 0 & 1 & y_p \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_p \\ 0 & 1 & -y_p \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & x_p(1-\cos\theta) + y_p \sin\theta \\ \sin\theta & \cos\theta & y_p(1-\cos\theta) + x_p \sin\theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{29}$$

Scaling around a pivot

1. Translate from the pivot point $(x_p, y_p$ to the origin
2. Scale around the origin by $(s_x, s_y$
3. Translate back to the pivot point

$$
\begin{aligned}
R_{x_p,y_p}(\theta) &= T(x_p, y_p)R_z(\theta)T(-x_p, -y_p) \\
&= \begin{bmatrix} 1 & 0 & x_p \\ 0 & 1 & y_p \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_p \\ 0 & 1 & -y_p \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} s_x & 0 & x_p(1-s_x) \\ 0 & s_y & y_p(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{30}
$$

Scaling around a pivot and orientation

1. Translate from the pivot point (x_p, y_p) to the origin
2. Rotate to align the scale axis $\hat{n} = (n_x, n_y)$ with the x-axis $R(-\theta) : \theta = \arctan\frac{n_y}{n_x}$
3. Scale about the origin by (s_x, s_y)
4. Rotate back by θ
5. Translate back to the pivot point

$$
S_{(x_p,y_p,\hat{n})}(s_x, s_y) = T(x_p, y_p)R(\theta)S(x_p, y_p)R(-\theta)T(-x_p, -y_p) \tag{31}
$$

You can pre-compute the complete matrix or just let the computer multiply the matrices.

Shears

A shear moves different parts of the coordinate system differently. Shears can simulate tilting of the 2-D coordinate plane away from the viewer.

$$
Sh_x(k_x) = \begin{bmatrix} 1 & k_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{32}
$$

$$
Sh_y(k_y) = \begin{bmatrix} 1 & 0 & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{33}
$$

$$
Sh_{xy}(k_x, k_y) = \begin{bmatrix} 1 & k_x & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{34}
$$

3.2 2-D Viewing Pipeline

Consider the following situation. You have designed a nice little 2D logo. You want to make the logo appear as though it were on the label of a vinyl record. Furthermore, the record is on a turntable that is spinning. You want the camera view to begin with the turntable fully visible, then zoom in on the label, moving with it as the view gets closer until the label appears to be rotating in place in the middle of the image as the rest of the scene rotates around it. All of this should be rendered on a 4500px by 8000px image.

For any given frame of the sequence, the transformation from model point to screen coordinate is describable by a single matrix transformation. However, building the transformation is a non-trivial process, and there are many parameters. We want to control the location of the label on the record, the speed of the turntable, what part and what amount of the world is visible, and the size of our final image.

Breaking apart the scenario in different coordinate systems is an extremely useful way to subdivide the problem into manageable pieces. In fact, it is so useful, that there is a standard pipeline of coordinate systems that almost all CG scenes use to move from model coordinates to screen coordinates.

1. Model Coordinates - Model coordinates are where you do your work. The model coordinate frame is attached to the object in a convenient way.

   - Each model can have its own coordinate system
   - A collection of models can have its own coordinate system
   - Model coordinates should make it easy to describe the model or collection

   In the example above, there are several useful model coordinate systems.

   - The logo should have its own origin and orientation that make it easy to describe. The central rotation point of the logo, for example, would be a useful model origin.
   - The record should have its own origin. The center of the record makes the most sense, in this case. The orientation of the axes are arbitrary since the object is radially symmetric.
   - The turntable should have its own model coordinates; a corner origin makes it simple to define.

2. World Coordinates - The world coordinate frame allows you to place models in relation to one another, providing a single frame of reference. The world coordinate system is the frame in which the viewing system is defined.

   - Pick a convenient origin
   - A transformation defines the location of each instance of each model in the scene
     - We can create multiple instances of each model or object
     - Creating a new transformation matrix for an object puts it in a different place
     - Duplicating the polygons in a model and adjusting the transformation makes a new copy

   In the example above, aligning the world coordinates with the turntable coordinates is not a bad choice. It is the largest object in the scene, and locating it at $(0, 0)$ makes sense. If the turntable were itself sitting on a table inside a house, then choosing a world coordinate system aligned with the house would be more appropriate. Given the world coordinate system, we can specify the location of each object. Note that, in some situations, it's better to define the position of objects relative to one another–e.g. the record relative to the turntable–rather than using world coordinates.

3. Viewing Coordinates - The viewing coordinate frame specifies how the camera views the world. The view coordinates are defined relative to world coordinates through an imaging model.

   - In 2D, the imaging model is generally a box, but other shapes are possible
   - The box can be located anywhere, possibly in any orientation
   - Need to specify the size, location, and orientation of the box
   - Viewing coordinates are defined relative to world coordinates so you know where the camera is located relative to objects in the scene.

   In the example above, the 2D view window could be defined by an origin in the center of the window, the length of its sides (in world coordinate units) and the orientation of the view window's horizontal axis relative to the world coordinate x-axis. It is important to think about the view window in world coordinate units, not pixels. The number of pixels in the final image is arbitrary and many different size images could be generated from a single view of a scene.

   To move from world coordinates to viewing coordinates, we need to translate the origin of the view window (v_x, v_y) to the world coordinate origin and then rotate to orient the view window x-axis (n_x, n_y) with the world coordinate x-axis. If (n_x, n_y) represent a normalized vector, then the transformation is given below.

$$X_{\text{world}\rightarrow\text{view}}(v_x, v_y, n_x, n_y) = R_z(n_x, -n_y)T(-v_x, -v_y) \tag{35}$$

   In general, if you have a coordinate system defined by two orthonormal vectors (u_x, u_y) and (v_x, v_y), then the matrix to align the coordinate system with the world axes is given below.

$$X_{\text{align}}(\vec{u}, \vec{v}) = \begin{bmatrix} u_x & u_y & 0 & 0 \\ v_x & v_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{36}$$

4. Normalized Viewing Coordinates - Normalized viewing coordinates are independent of the input or output viewing system. The points are scaled to be in the range $[0, 1]$

   - Clipping is possible (and fast) with normalized viewing coordinates
   - Normalized coordinates are ready for multiple display systems

   To move from viewing coordinates to normalized viewing coordinates we scale by the width du and height dv of the view window.

$$X_{\text{view}\rightarrow\text{norm}}(du, dv) = S(\frac{1}{du}, \frac{1}{dv}) \tag{37}$$

5. Screen Coordinates - Screen coordinates specify which pixels to light up. The coordinate system is defined by the output display device and each output display device will have its own. Using normalized viewing coordinates makes it easy to send the output to numerous different output devices.

   To move to screen coordinates we need to invert the y-axis, scale to the number of pixels in each direction, then translate the view window so all y-coordinates are positive.

$$X_{\text{norm}\rightarrow\text{screen}}(\text{rows}, \text{cols}) = T(\text{cols}/2, \text{rows}/2)S(\text{cols}, -\text{rows}) \tag{38}$$

4 3-D Transformations

Move from 2-D to 3-D adds a z-coordinate to the points and matrices. Otherwise, the concepts are identical. The number of rotations increases, as we can rotate around any of the three axes, not just the z-axis.

Translation

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 + t_x \\ y_0 + t_y \\ z_0 + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{39}$$

Rotation (about X-axis)

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \cos\theta - z_0 \sin\theta \\ y_0 \sin\theta + z_0 \cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{40}$$

Rotation (about Y-axis)

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \cos\theta + z_0 \sin\theta \\ y_0 \\ -x_0 \sin\theta + z_0 \cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{41}$$

Rotation (about Z-axis)

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \cos\theta - y_0 \sin\theta \\ x_0 \sin\theta + y_0 \cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{42}$$

Scaling

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 s_x \\ y_0 s_y \\ z_0 s_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{43}$$

Rotations about a single axis commute, but not rotations about different axes. One other interesting fact about rotations is that to reverse the effect of a rotation you just transpose the matrix.

$$R_a(-\theta) = R_a^t(\theta) \tag{44}$$

The transpose of a rotation matrix is also its inverse, which makes sense, as $R_a(\theta)R_a(-\theta) = I$.

Orienting Coordinate Systems

If you have an orthonormal basis (three orthogonal, normalized vectors that follow the right hand rule), then rotating the basis to orient with the primary axes is a one-step process. Given the three basis vectors $\vec{u}, \vec{v}, \vec{w}$, the matrix that orients the coordinate system with the primary axes is given by $R_{xyz}(\vec{u}, \vec{v}, \vec{w})$. The transpose of the matrix will execute the inverse rotation.

$$R_{xyz}(\vec{u}, \vec{v}, \vec{w}) = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{45}$$

The orienting matrix makes it possible to easily do things like scale in an arbitrary coordinate system $(\vec{u}, \vec{v}, \vec{w})$ around a pivot point (p_x, p_y, p_z).

$$S(\vec{u}, \vec{v}, \vec{w}; p_x, p_y, p_z; s_x, s_y, s_z) = T(p_x, p_y, p_z) R^t(\vec{u}, \vec{v}, \vec{w}) S(s_x, s_y, s_z) R(\vec{u}, \vec{v}, \vec{w}) T(-p_x, -p_y, -p_z) \tag{46}$$

Shears, mirroring, and other transformations are also possible in 3D, just as in 2D. More complex shears are possible because any two axes can affect the third.

5 Viewing

Generic 2D viewing requires only rotations, translations, and scales in order to implement a view rectangle at an arbitrary location and orientation. Since we are converting coordinates from a plane to a plane, we don't need anything special in order to view the scene.

In order to view a 3D scene, however, we have to project 3D points onto a plane. Mappings onto a plane are the domain of plane geometric projections. Generic geometric projects are, in general, between equivalent dimensional spaces. 3D rotations, translations, and scales map from one 3D space to another. Plane geometric projects, however, drop one dimension. Therefore, the rank of a 4x4 plane geometric projection matrix can be no more than three.

Geometric projections fall into one of two categories.

- Affine: affine transforms maintain parallel lines
- Perspective: perspective transforms do not necessarily maintain parallel lines (e.g. railroad tracks going off into the distance)

When defining viewing pipelines, we can view the process in one of two ways.

- Fixed viewing geometry: the viewer remains stationary and the object moves
- Fixed object geometry: the scene remains stationary and the viewer moves

Both approaches produce identical matrices, but the two processes make sense in different situations. A fixed viewing geometry is useful when working with CAD models or 3D modeling programs. Then the situation is similar to a carver holding a statue and moving it around in front of their eyes. Fixed object geometry is useful when rendering a scene from the point of view of someone within it. We don't tend to think of a city flowing by us or rotating around as we walk down a sidewalk.

Planar geometric projections divide into two large groups and then several subcategories within them. Most of the subdivisions are specialty viewing situations designed to be useful in engineering drawings.

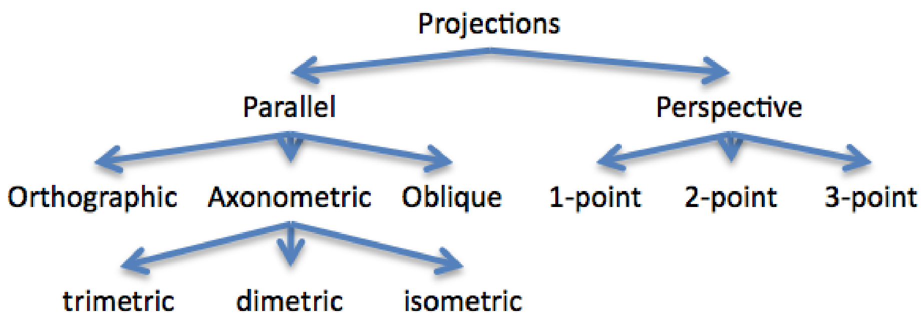


Figure 6: Categories of planar projections

All projection methods can be defined by a plane and a center of projects [COP]. To find the location of a 3D world point on the view plane, create a line from the center of projection through the world point and find where it intersects with the view plane. That defines its projection. The difference between parallel and perspective projections is whether the COP is at infinity (parallel) or at a finite distance (perspective).

5.1 Orthographic Projection

Orthographic projection is the simplest of the projection methods: drop the z-coordinate. Conceptually, the view plane is a rectangle parallel to the x-y plane. The COP is at infinity in the Z-direction, so each line perpendicular to the x-y plane is a line of projection. The orthographic projection matrix is an identity matrix with the z-row set to all zeros.

$$P_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{47}$$

Engineering drawings are normally created using orthographic projection. By using the six possible combinations of axes and COP orientations an object can be rendered so all faces are visible in at least one drawing. Furthermore, distances are preserved for any face parallel to the projection plane.

5.2 Axonometric Projections

Axonometric projection is a variation of orthographic projection. The various axonometric projections occur when multiple faces of an object are visible. The concept is only meaningful for fixed viewing geometry. The subdivisions of axonometric viewing indicate how many rulers are required to measure the different faces of the object.

1. Trimetric projection: three faces of an object are visible, but the three faces require different rulers. When a line segment is angled relative to the view plane, its projected length is dependent upon the angle. A larger angle means the same unit step in the image corresponds to a longer step in the 3D scene. In trimetric projection, the three sides of the object have unique angles relative to the view plane.
2. Dimetric projection: two faces of an object share the same ruler. Two of the a sides of the object have identical angles relative to the view plane.
3. Isometric projection: all faces of the object share the same ruler. There are only four possible isometric projections.
   - The foreshortening factor is 0.8165 (a step of 0.8165 in the world corresponds to a step of 1 in the image).
   - The angle with the horizontal axes and the projected axis is $\pm 30°$.
   - Isometric projections are often used in engineering drawings because it is easy to measure distances.

5.3 Oblique Projections

Orthographic and axonometric projections both use lines of projection that are perpendicular to the view plane. Oblique projections occur when the line from the view plane origin to the COP is not perpendicular to the view plane. Conceptually, it's like staring straight ahead and seeing stuff to the side (but not stuff straight ahead). With oblique projections, faces parallel to the view plane are rendered without distortion, but non-parallel faces and lines are distorted (basically with a shear).

Oblique projection lets you see the side of an object when that side is perpendicular to the view plane.

Cavalier projection: the angle between the projective lines and the view plane is $\alpha = 45°$.

- The foreshortening factor for lines perpendicular to the view plane is $\sqrt{2} = 0.7071$

Cabinet projection: the angle between the projective lines and the view plane is $\alpha = 63.43°$.

- The foreshortening factor for lines perpendicular to the view plane is 0.5

In general, f is the desired foreshortening factor for the z-axis and α is the angle the projected z-axis makes with the x-axis. The horizontal orientation of the projection vector is determined by f. The vertical orientation of the projection vector is determined by α.

$$P_{\text{oblq}} = \begin{bmatrix} 1 & 0 & -f\cos\alpha & 0 \\ 0 & 1 & -f\sin\alpha & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{48}$$

5.4 Arbitrary Parallel Projections

All of the above projections are suitable for a fixed viewing geometry situation. However, we can also set up parallel projects for fixed scene geometry in a way that includes all of the above projection methods. The basic concept is to define a camera in the world and then develop a viewing pipeline that transforms world coordinates into image coordinates.

The camera:

- View Reference Point [VRP] is the origin (center) of the view plane VRP $= (P_x, P_y, P_z)$
- View Plane Normal [VPN] is the normal to the view plane VPN $= (n_x, n_y, n_z)$
- View Up Vector [VUP] is the direction of the y-axis on the view plane VUP $= (Y_x, Y_y, Y_z)$
- View plane extent in coordinates relative to the VRP on the view plane $(u_0, v_0), (u_1, v_1)$
- Direction of Projection [DOP] is the direction of the lines of projection in the coordinate system defined by the VPN and VUP vectors DOP $= (D_x, D_y, D_z)$
- Front clip plane and back clip plane, specified as distances along the VPN, F, and B.
- Screen size in pixels R and C.

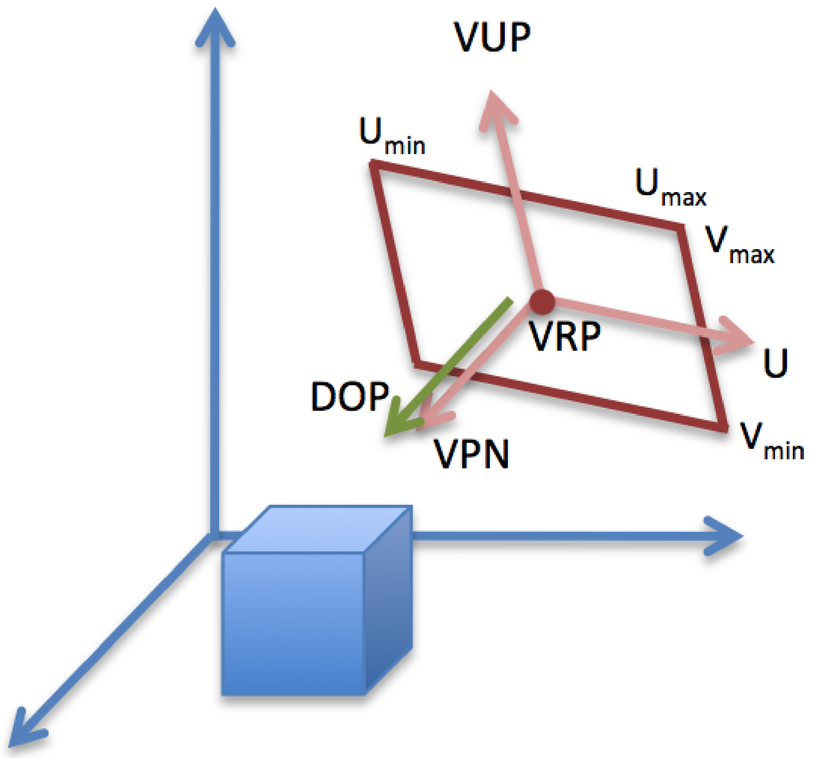


Figure 7: View geometry showing the view plane, its origin (VPN), the view reference coordinates (VPN, VUP, and $\hat{U}$), the direction of project (DOP) and the view window boundaries $(U_{\text{min}}, U_{\text{max}}), (V_{\text{min}}, V_{\text{max}})$

The basic idea of the parallel projection process is to align the view reference coordinate system with the world coordinate axes, shear so the view volume is aligned with the axes, scale the volume to an appropriate size, and then project the world onto a plane using orthographic projection.

The process:

1. Calculate the U vector for the view reference coordinate system

$$U = \text{VUP} \times \text{VPN} \tag{49}$$

2. Recalculate the VUP so that the view reference coordinates are orthogonal

$$\text{VUP} = \text{VPN} \times U \tag{50}$$

3. Translate the VRP to the origin

$$\text{VTM}_1 = T(-\text{VRP}) \tag{51}$$

4. Normalize U, VUP and VPN and use the R_{xyz} rotation method to align the axes.

$$\text{VTM}_2 = R_{xyz}(\hat{U}, \hat{\text{VUP}}, \hat{\text{VPN}})\text{VTM}_1 \tag{52}$$

5. Shear so the DOP is aligned with the positive z-axis.

$$\text{VTM}_3 = Sh_z(\frac{D_x}{D_z}, \frac{D_y}{D_z})\text{VTM}_2 \tag{53}$$

$$Sh_z(k_x, k_y) = \begin{bmatrix} 1 & 0 & k_x & 0 \\ 0 & 1 & k_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{54}$$

6. Translate so that the center of the view window and the front clip plane are at the origin.

$$\text{VTM}_4 = T(-\frac{u_0 + u_1}{2}, -\frac{v_0 + v_1}{2}, -F)\text{VTM}_3 \tag{55}$$

7. Scale to the canonical view volume, which is the box bounding by $[-1, 1]$ in x and y and $[0, 1]$ in z.
   - The distance between the front and back clip planes is $B' = B - F$
   - The width of the view window is $du = u_1 - u_0$
   - The height of the view window is $dv = v_1 - v_0$

$$\begin{aligned}&\text{VTM}_{\text{World}\rightarrow\text{CVV}} = \\ &S(\frac{2}{du}, \frac{2}{dv}, \frac{1}{B'})T(-\frac{u_0 + u_1}{2}, -\frac{v_0 + v_1}{2}, -F)Sh_z(\frac{D_x}{D_z}, \frac{D_y}{D_z})R_{xyz}(\hat{U}, \hat{\text{VUP}}, \hat{\text{VPN}})T(-\text{VRP})\end{aligned} \tag{56}$$

At this point in the process, all vertices that are visible are within the canonical view volume. Clipping is normally executed here, since tests within the bounding volume are simple to calculate and evaluate. Clipping to the front and back of the view volume are especially important, as they cannot be handled in a 2D clipping process.

8. The next step in the process is to use orthographic projection to convert the 3D points to 2D points on the plane. Since the canonical view volume is aligned with the major axes and the DOP is aligned with the z-axis, orthographic projection just drops the z-coordinate.
9. The final step in the process is to scale and translate the scene into screen coordinates. Note that because we used a right-handed coordinate system for the view reference coordinates and made positive Z our viewing axis, that the X-axis points to the left. So we need to invert both the x- and y-axes to get screen coordinates.

$$\text{VTM}_{\text{CVV}\rightarrow\text{Screen}} = T(\frac{C}{2}, \frac{R}{2})S(-\frac{C}{2}, -\frac{R}{2})P_{\text{orth}} \tag{57}$$

5.5 Perspective Projection

Parallel projection is often used in 3D modeling systems (e.g. CAD Design) and some video games because of the need to map from 2D back into 3D (mouse clicks to object creation) and because of the need to compare relative distances accurately. Our eyes, however, do not implement parallel projection. Instead, things that are farther away get smaller.

Perspective projection was one of the concepts that came out of the Renaissance. Artists often think about perspective viewing in a fixed-view sense, orientating objects in the scene according to a 1-point, 2-point, or 3-point perspective view.

- A 1-point perspective occurs when the object's front face is parallel to the view plane, which means parallel lines on the front face remain parallel.
- A 2-point perspective occurs when the object is oriented so that lines in only one axis (usually the y-axis) are parallel to the view plane. For example, when a cube is oriented so an edge is towards the viewer and the line along the edge is parallel to the view plane.
- 3-point perspective occurs when none of the major object edges are parallel to the view plane.

In graphics, however, the fixed scene geometry view makes more sense with perspective projection. A useful paradigm is to think of the viewer as a camera that can move around and through the scene. Similar to a camera, we can parameterize the view with values that correspond to the size of the lens and the zoom setting.

The camera:

- View Reference Point [VRP] is the origin (center) of the view plane VRP $= (P_x, P_y, P_z)$
- View Plane Normal [VPN] is the normal to the view plane VPN $= (n_x, n_y, n_z)$
- View Up Vector [VUP] is the direction of the y-axis on the view plane VUP $= (Y_x, Y_y, Y_z)$
- View plane extent in coordinates relative to the VRP on the view plane $(u_0, v_0), (u_1, v_1)$
- Center of Projection [COP] is the location of the focal point of the projection, given in View Reference Coordinates $(\text{COP}_x, \text{COP}_y, \text{COP}_z)$. In many cases, the COP is reduced to an offset d along the negative VPN axis.
- Front clip plane and back clip plane, specified as distances along the VPN, F, and B.
- Screen size in pixels R and C.

As with parallel projection, the basic process is to align the view reference coordinate system with the world coordinate axes, move the COP to the origin and shear to center the view volume, scale the volume to an appropriate size, and then project the world onto a plane using a perspective projection.

The process:

1. Calculate the U vector for the view reference coordinate system

$$U = \text{VUP} \times \text{VPN} \tag{58}$$

2. Recalculate the VUP so that the view reference coordinates are orthogonal

$$\text{VUP} = \text{VPN} \times U \tag{59}$$

3. Translate the VRP to the origin

$$\text{VTM}_1 = T(-\text{VRP}) \tag{60}$$

4. Normalize U, VUP and VPN and use the R_{xyz} rotation method to align the axes.

$$\text{VTM}_2 = R_{xyz}(\hat{U}, \hat{\text{VUP}}, \hat{\text{VPN}})\text{VTM}_1 \tag{61}$$

5. Translate the COP to the origin. In many cases, the COP is just defined by the projection distance d. In that case, the translation becomes $T(0, 0, d)$.

$$\text{VTM}_3 = T(-\text{COP}_x, -\text{COP}_y, -\text{COP}_z)\text{VTM}_2 \tag{62}$$

6. Calculate the direction of projection (in case the COP or window center are not along the VPN).

$$\text{DOP} = \begin{bmatrix} \frac{umin+umax}{2} - \text{COP}_x \\ \frac{vmin+vmax}{2} - \text{COP}_y \\ -\text{COP}_z \end{bmatrix} \tag{63}$$

7. Shear by the DOP. Note that if the COP is located long the -VPN, then no shearing is required.

$$\text{VTM}_4 = Sh_z(\frac{D_x}{D_z}, \frac{D_y}{D_z})\text{VTM}_3 \tag{64}$$

8. Scale to the canonical view volume, which is a pyramid.
   - Update the location of the VRP
   $$\text{VRP}' = \text{VTM}_4\text{VRP} \tag{65}$$
   - Update the distance from the origin to the back clip plane.
   $$B' = \text{VRP}'_z + B \tag{66}$$
   - The width of the view window is $du = u_1 - u_0$
   - The height of the view window is $dv = v_1 - v_0$

$$\text{VTM}_{\text{World}\rightarrow\text{CVV}} = S(\frac{2\text{VRP}'_z}{B'du}, \frac{2\text{VRP}'_z}{B'dv}, \frac{1}{B'})\text{VTM}_4 \tag{67}$$

The canonical view volume for perspective projection is a pyramid with its apex at the origin. The back clip plane is at $Z = 1$, and the base of the pyramid is a 2x2 square from (-1, -1) to (1, 1) in X and Y. As with parallel projection, clipping needs to occur here, particularly front and back clipping. The front clip plane is located at F'.

$$F' = \frac{\text{VRP}'_z + F}{B'} \tag{68}$$

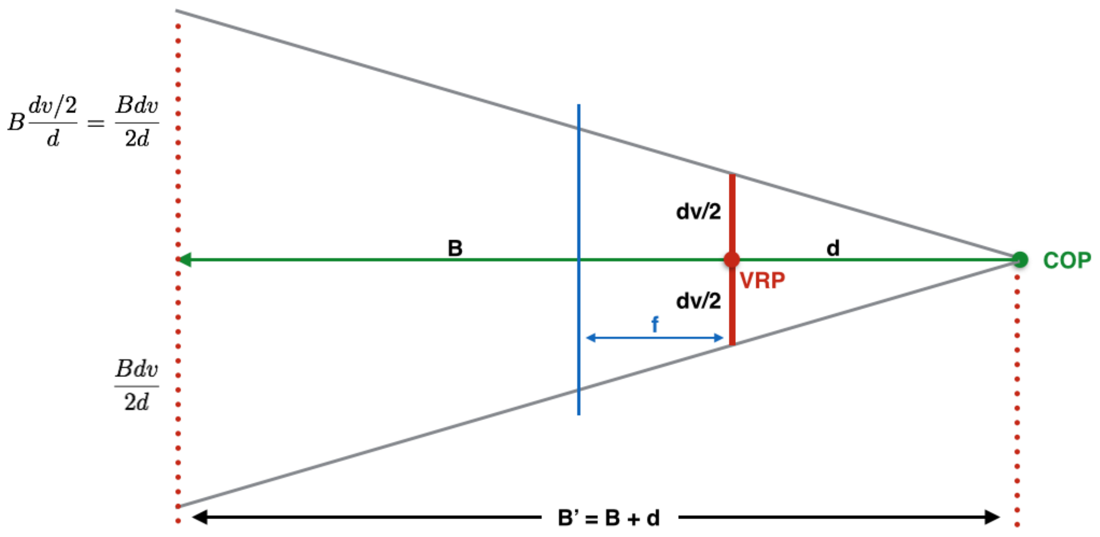


Figure 8: The scale parameters necessary to convert the view volume to the canonical pyramid. When the COP is not on the negative VPN, then VRP'_z is used as the distance d.

9. Project the scene onto the view plane, located a distance d along the VPN.

$$d = \frac{\text{VRP}'_z}{B'} \tag{69}$$

$$P_{\text{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \tag{70}$$

10. Scale to the image size, negate the x- and y-axis, and translate the origin to the center of the image. Note that the view plane is $2d \times 2d$, where d is the updated projection distance.

$$VTM_{\text{CVV}\rightarrow\text{Screen}} = T(\frac{C}{2}, \frac{R}{2})S(-\frac{C}{2d}, -\frac{R}{2d})P_{\text{per}} \tag{71}$$

5.5.1 Perspective Projection Matrix

Why does the perspective projection matrix work? Consider figure 9. The projection of the point y onto the image plane produces $y' = \frac{yd}{z}$.

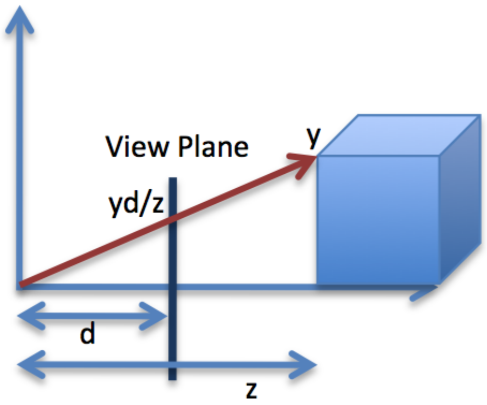


Figure 9: Similar triangles show how the scene location is converted to a point on the view plane.

Now consider the effect of the perspective matrix on the point $[xyz1]'$. Re-normalizing the homogeneous point after the transformation produces the perspective projection onto the view plane.

$$\begin{bmatrix} \frac{xd}{z} \\ \frac{yd}{z} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{d}{z} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{72}$$

Therefore, the perspective transformation requires us to re-normalize the homogeneous coordinates after multiplying by the perspective projection matrix. One caveat to the process, however, is that it can be important to leave the z-coordinate alone rather than also dividing it by h. The perspective transformation converts all z-coordinates to the value d, but we may still have a use for the z-coordinate later on when determining visibility. In practice, the following pipeline is useful to implement.

1. Multiply by $\text{VTM}_{\text{World}\rightarrow\text{CVV}}$ to obtain CVV coordinates.
2. Clip graphics objects to the front and back clip planes.
3. Multiply by $\text{VTM}_{\text{CVV}\rightarrow\text{Screen}}$.
4. Divide x and y by the homogeneous coordinate to obtain the true screen coordinates.
5. Draw the graphics object to the screen.

5.5.2 Simple Perspective View

There are two assumptions we can make that simplify the view transformation process.

- The VRP is the center of the view window, and the window is defined by its width du and height dv.
- The COP is on the negative VPN and is defined by a distance d.

With the simplifying assumptions, the new process is as follows.

1. Calculate the U vector for the view reference coordinate system
2. Recalculate the VUP so that the view reference coordinates are orthogonal
3. Translate the VRP to the origin
4. Normalize U, VUP and VPN and use the R_{xyz} rotation method to align the axes.
5. Translate the COP to the origin using the transformation $T(0, 0, d)$.

   Update the back clip plane distance $B' = d + B$
6. Scale to the canonical view volume using $S(\frac{2d}{B'du}, \frac{2d}{B'dv}, \frac{1}{B'})$
7. Project the scene onto the view plane, located a distance d' along the VPN.

$$d' = \frac{d}{B'} \tag{73}$$

8. Scale to the image size, negate the x- and y-axis, and translate the origin to the center of the image.

$$VTM = T(\frac{C}{2}, \frac{R}{2})S(-\frac{C}{2d'}, -\frac{R}{2d'})VTM \tag{74}$$

Figure 10 shows all of the parameters of the simple perspective view in world coordinate space.

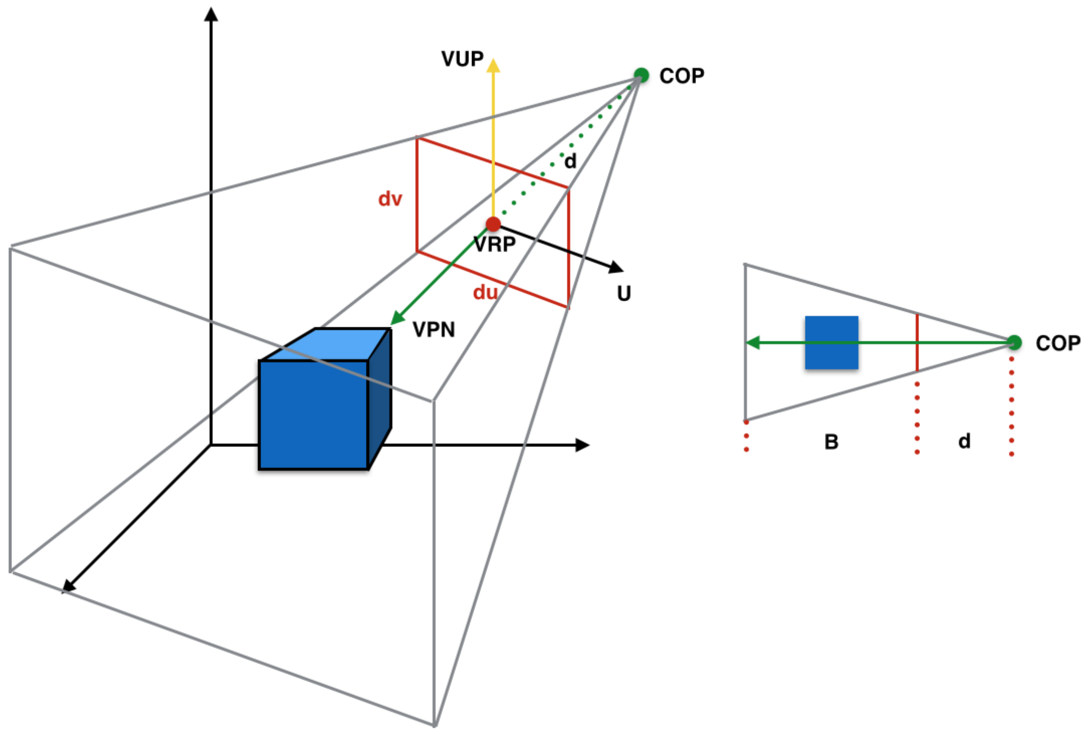


Figure 10: The complete specification of the perspective view.

5.6 Viewing Pipeline

Now that we know the steps required to move from 3D models to a 2D image, we can build a rough model of the viewing pipeline. Modern graphics cards implement most of the steps in the pipeline, and do so using massive parallelism since the same process is applied to many, many data points. The pipeline assumes objects are in world coordinates and that the viewing geometry is defined.

1. Transform the graphics primitives–usually points, lines, or triangles–to the canonical view volume
2. Clip the graphics primitives to the CVV
3. Transform the graphics primitives into screen space
4. Draw the visible primitives into the image

We have developed techniques for doing everything except the last step. Hidden surface removal (which means the same thing as visible surface drawing) is a topic we'll cover in section 8.

There are some basic CS concepts tucked into the graphics pipeline. The first is whether the pipeline should be applied serial to each graphics primitive or whether all of the primitives should move along the pipeline simultaneously. For transformations and clipping, it turns out there isn't much difference between the two approaches. Moving the data along each step simultaneously offers some potential caching advantages, but the number of operations is similar.

When drawing the visible primitives onto the screen, however, there are definite benefits to handling all of the graphics primitives simultaneously. The primary benefit is that we can avoid drawing over a pixel multiple times. Instead, if we have access to all of the polygons that cover a particular pixel, we can identify which polygon should be drawn and color the pixel once. It also permits us to more effectively implement concepts like anti-aliasing and transparency. We'll go over these algorithms and concepts when we cover hidden surface removal.

6 Hierarchical Modeling

A scene is a complex combination of basic primitive shapes. Most real-time rendered graphics, for example, consist purely of triangles. However, building a complex scene from triangles is similar to building a complex image one pixel at a time. Scenes are not, in general, abstract combinations of shapes, but are constructed from objects that have coherence. When we want a table in a scene, we do not think of it as 1000 triangles, but as a single object. We want to be able to place, orient, and scale the table as a unit.

Hierarchical modeling allows us to construct a scene out of parts. Each part may be further subdivided into simpler parts: a table might be constructed from four legs and a top. Having a hierarchical model of the scene permits us to separately manipulate parts of the scene at different levels. We can move the table as a unit, or we could expand the size of the legs, or we could add a small mark to one polygon.

At the root of hierarchical modeling is the concept of a structure or module. A module is one or more primitive graphic elements, attributes, and local transformations, organized as a directed acyclic graph. In addition to graphics primitives and transformations, a module can include other modules, enabling a module to become a node in a larger graph.

- The graph defines a sequence of operations that generate drawing commands
- Modules can contain modules
- Traversal of the graph is depth first
- A module can contain another module many different times (multiple instances)

It's important to understand that a module is a recipe for drawing something. When a module is inserted into a scene graph then the scene contains an **instance** of the module. A scene can contain many instances of a module, all of which have identical recipes.

One method of building the scene graph is, in fact, to have only a single physical copy of each module, storing a reference to the module in the overall scene graph for each instance. Making a copy of a module is more costly, but enables modifications to a single instance that do not propagate to the rest of the scene. However, one of the strengths of using a reference instead of a copy is that changes to a module do propagate across a scene with no further work required. If you find a missing polygon in your chair model, for example, changing your code in one place fixes it for the entire scene.

Modules allow us to build up a scene in layers that correspond to logical partitions. They reduce duplication and enable fast development of complex scenes with a small number of actual 3D polygons. A unit box module, for example, can be used to build an arbitrary box of any dimension by preceding the box with a set of transformations. More generally, the combination of a set of transformations and simple geometric primitives–e.g. spheres, cylinders, boxes, or pyramids–provides the basic building blocks for many objects and scenes.

Within a module, the elements form an ordered list. When traversing the list, the interpreter must maintain a current transformation state. Any graphics primitive is drawn using the current transformation state, generally called the **local transformation matrix** or **[LTM]**. The local transformation matrix collects transformations by pre-multiplying the most recent transformation by itself.

$$LTM_i = T_i LTM_{i-1} \tag{75}$$

A module must also take into account the **global transformation matrix** or **[GTM]** from its parent module. When the interpreter finds a module in the list of elements, it right multiplies the current LTM with the GTM and passes the product to the child module as its GTM.

$$GTM_{\text{child}} = GTM_{\text{parent}} LTM_{\text{parent}} \tag{76}$$

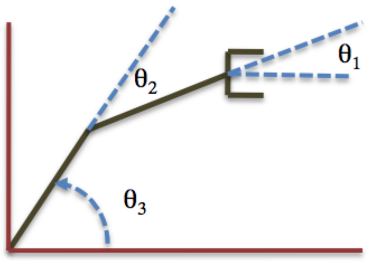
Example: A Robot Arm

A robot arm (or human arm) is an example of an object with a hierarchical relationship. If the elbow joint moves, then the hand moves as well. If the shoulder moves, the whole arm moves.

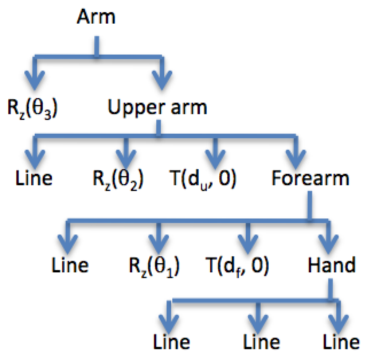
We can build up a robot arm using an end effector (a triangle) and a lines.

1. Create the hand module
   - Draw line 1
   - Draw line 2
   - Draw line 3
2. Create the forearem
   - Draw a line for the forearm
   - Rotate by the end-effector angle $R_z(\theta_1)$
   - Translate to the end of the forearm $T(d_f, 0)$
   - Draw the hand module
3. Create the upper arm
   - Draw a line for the upper arm
   - Rotate by the forearm angle $R_z(\theta_2)$
   - Translate to the end of the upper arm $T(d_f, 0)$
   - Draw the forearm module
4. Create the base
   - Rotate by the upper arm angle $R_z(\theta_3)$
   - Draw the upper arm

Each of the joints is encoded as a rotation parameter. To figure out what the transformation is for any given line, we simply concatenate the transformations from the node to the root. By adjusting the rotation angles, we can make the robot move in what appears to be a realistic manner. The entire arm is parameterized by the joint angles and the length of the translations.



(a)



(b)

Figure 11: (a) Robot arm with three joints and (b) its associated scene graph.

6.1 Building a Scene Graph

How do we implement the mechanics of building a scene graph? What capabilities do we need?

- We need to be able to begin and end modules.
- We need to be able to add graphics primitives to modules (points, lines, polygons, circles, ellipses, polylines, etc)
- We need to be able to add transformations (scales, rotations, translations)
- We could also use the module structure to manage drawing attributes (OpenGL does this)

In a very real sense we are designing a programming language. The keywords for our programming language could be:

- begin - opens a new module for collecting commands
- end - terminates collecting commands to the currently open module
- translate - adds a translation to the open module
- rotate - adds a rotation to the open module
- scale - adds a scale to the open module
- identity - sets the transformation to identity in the open module
- point - adds a point to the open module
- line - adds a line to the open module
- polygon - adds a polygon to the open module
- polyline - adds a polyline to the open module
- circle - adds a circle to the open module
- ellipse - adds an ellipse to the open module

A simple way to organize a module is using a linked list. In general, we will want to add things to the list quickly–so keep a link to the end of the list–and traverse the list. We will rarely do arbitrary insertion or deletions. Operations are always inserted at the end of the list.

There are many design decisions associated with building modules.

- Does the programmer get access to the modules?
  - OpenGL - programmer gets an ID number but no access to the underlying module list
  - Giving programmer access/control of modules permits more flexible programming
- How do you communicate the open module information to the routines that add primitives or transformations?
  - Could use a global variable that holds a pointer to the currently open module (or NULL if none exists)
  - Could pass in the module to which you want to add the item

- How do you represent the elements of a module (primitives, transformations, and modules)?
  - Need to encode what the primitive or transformation is
  - Need to encode the parameters, often different numbers of parameters
  - All need to be part of a linked list
  - Inheritance (C++) or union structures (C) are a reasonable way to represent elements.

A **union** in C is a polymorphic data structure that can hold one of a set of types. The compiler allocates enough space to hold the largest type, but the programmer controls which type is used to access it.

Example: Unions

We can use an Element as the node structures for our linked list, with a union to hold the object data.

```
// defines a union that can hold one instance of any of the constituent types
typedef union {
  Point point;
  Line  line;
  Polygon polygon;
  Matrix matrix;
  void *module;
} Object;

// define an Element. The type field specifies what type the union holds
typedef struct {
  ObjectType type;
  Object obj;
  void *next;
} Element;

// define a Module as holding a linked list
typedef struct {
  Element *head;
  Element *tail;
} Module;
```

To access the fields of a union, use the field name in the union as a prefix to the structure's fields. Note that each union variable can hold only one of the given data types; it does not hold the different elements in different memory locations.

```
Object obj; // declare a union object

// treat the union as a point and access the x coordinate
obj.point.val[0] = 0.0;

// treat the union as a polygon and access the numVertex field
obj.polygon.numVertex = 4;
```

In the above code snippet, for example, setting numVertex means that trying to access the union as a point will no longer make sense.

6.2 Traversing a Scene Graph

We traverse a scene graph in order from the top level down in a depth-first fashion. During the traversal, we have to keep track of the current transformation matrix. It is useful to divide the transformations into three parts.

- VTM - View Transformation Matrix, affects all drawn elements
- GTM - Global Transformation Matrix, contains all of the transformations from the higher levels of the scene graph
- LTM - Local Transformation Matrix, contains all of the transformations seen so far in the current module

When traversing the graph, to draw a graphic primitive we always multiply the points by the current transformation matrix before drawing. The CTM is the matrix product of the VTM, GTM, and LTM.

$$[\text{CTM}] = [\text{VTM}][\text{GTM}][\text{LTM}] \tag{77}$$

When we encounter a transformation matrix, it always premultiplies the LTM. Conceptually, you can think of it as squeezing in between the GTM and LTM, which is why we need to keep them separate.

When we encounter a module, we need to create a new transformation matrix for the module that is the product of the GTM and LTM. Then we recursively call the scene traversal, passing in the new matrix as its GTM. When the module completes, the old GTM and LTM get restored. In most cases it makes sense to keep the VTM as a separate matrix throughout the graph traversal, passing it around as necessary.

The basic idea of the scene traversal algorithm is to loop over the elements of the current module's list and take the appropriate action depending upon the element's type.

- If the element is a graphics primitive, the algorithm calculates the current transformation and uses it to transform all of the points into the correct location. Then it draws the primitive using the transformed points.
- If the element is a matrix, the matrix pre-multiplies the LTM.
- If the element is a module, the algorithm calculates a new GTM, which is the old GTM multiplied by the LTM, and passes the module and the new GTM to the recursive call.

One potential addition to the list of operations above are **push matrix** and **pop matrix** operations for the local transformation matrix. OpenGL, for example, implements push and pop operations. This can be convenient when working with a module that has many pieces to it, all relative to the same location. The push operation sticks a copy of the current local transformation matrix (LTM) onto a stack. The pop operation restores the LTM to the top transformation matrix off the stack.

Scene Traversal Algorithm

The following is pseudo-code for for the Module_draw function that shows the cases for a subset of the different graphics types.

```
Module_draw
   Parameters:
      Module md
      Matrix VTM
      Matrix GTM
      DrawState DS
      Lighting light
      Image src

set the matrix LTM to identity

for each element E in the module md
  switch on the type of E
    Color
      DS->color = color data in E
    Point
      copy the line data in E to X
      transform X by the LTM
      transform X by the GTM
      transform X by the VTM
      normalize X by the homogeneous coord
      draw X  using DS->color (if X is in the image)
    Line
      copy the line data in E to L
      transform L by the LTM
      transform L by the GTM
      transform L by the VTM
      normalize L by the homogeneous coord
      draw L using DS->color
    Polygon
      copy the polygon data in E to P
      transform P by the LTM
      transform P by the GTM
      transform P by the VTM
      normalize P by the homogeneous coord
      if DS->shade is ShadeFrame
        draw the boundary of P using DS->color
      else if DS->shade is ShadeConstant
        draw P using DS->color
    Matrix
      LTM = (Matrix field of E) * LTM
    Identity
      LTM = I
    Module
      TM = GTM * LTM
      tempDS = DS
      Module_draw( (Module field of E), VTM, TM, tempDS, Light, src )
```

7 3D Models

Given that we know where to draw things, what should we draw?

We want to draw 3D scenes, which means we need to model stuff. There are many different ways to model stuff, and they balance a number of different factors.

- Ease of use by graphics designers
- Accuracy in modeling
- Speed of computation

These factors do not always coincide. There are many models that are easy to use, but that are not necessarily accurate or fast. There are very accurate models that are not particularly easy to use for design work.

Almost all modeling systems end up using the same end-stage pipeline in order to meet the needs of speed. The end stage pipeline is defined by the system we've examined so far, which is implemented on most graphics cards: points, lines, or triangles transformed from 3D to 2D and drawn into an image.

All modeling systems, in the end, somehow convert the representations into points, lines, or triangles and feed them through the standard pipeline.

7.1 Lines

The regular 2D line equation is $y = mx + b$, but that form of the equation creates challenges, especially in 3D. The alternative is to use a parametric representation of lines. Parametric representations represent the degrees of freedom of a model. In the case of a line, there is only one degree of freedom: distance along the line. If we have an anchor point A and a direction $\vec{V}$ then we can describe any point on the line using (78)

$$X = A + t\vec{V} \tag{78}$$

In (78) the value of t represents distance long the line. Two values of t define a line segment. Often, A and $\vec{V}$ are set up so that the line segment is defined by $t \in [0, 1]$.

Parametric representations are common in graphics. It turns out that the parametric representation of a line leads to a simple clipping algorithm.

7.1.1 Line Clipping: Liang-Barsky / Cyrus-Beck

Given: a parametric representation of a line with $t \in [0, 1]$

Goal: to clip the line to the visible window. Possible outcomes include drawing the entire line, part of the line, or none of the line. The line may need to be clipped to more than one side of the window.

In the end, we want to know the range of values of t that make the following inequalities true.

$$\begin{aligned} x_{\text{min}} \leq A_x + tV_x \leq x_{\text{max}} \\ y_{\text{min}} \leq A_y + tV_y \leq y_{\text{max}} \\ z_{\text{min}} \leq A_z + tV_z \leq z_{\text{max}} \end{aligned} \tag{79}$$

The above inequalities can be expressed as four inequalities of the form $tp_k \leq q_k$. The expressions for p_k and q_k are given in (80). Note that the algorithm scales easily into 3D or higher dimensions, if necessary.

$$\begin{aligned} p_1 &= -V_x & p_2 &= V_x \\ p_3 &= -V_y & p_4 &= V_y \\ p_5 &= -V_z & p_6 &= V_z \\ q_1 &= A_x - x_{\text{min}} & q_2 &= x_{\text{max}} - A_x \\ q_3 &= A_y - y_{\text{min}} & q_4 &= y_{\text{max}} - A_y \\ q_5 &= A_z - z_{\text{min}} & q_4 &= z_{\text{max}} - A_z \end{aligned} \tag{80}$$

The various p and q values tell us about the line.

- If a line is parallel to a view window boundary, the p value for that boundary is zero. If the line is parallel to the x-axis, for example, then p_1 and p_2 must be zero.
  - Given $p_k = 0$, if $q_k < 0$ then the line is trivially invisible because it is outside view window.
  - Given $p_k = 0$, if $q_k \geq 0$ then the line is inside the corresponding window boundary. It is not necessarily visible, however.
- Given $p_k < 0$, the infinite extension of the line proceeds from outside the infinite extension of the view window boundary to the inside.
- Given $p_k > 0$, the infinite extension of the line proceeds from inside the infinite extension of the view window boundary to the outside.

For any non-zero value of p_k, we can calculate the value of t that corresponds to the point on the line where it intersects the view window boundary.

$$t_k = \frac{q_k}{p_k} \tag{81}$$

To clip a line to the view window, we want to calculate the t_0 and t_f that define the visible segment.

- t_0 will be either 0, if the start of the line is within the view window, or the largest t_k for all $p_k < 0$.
- t_f will be either 1, if the end of the line is within the view window, or the smallest t_k for all $p_k > 0$.
- If the calculations result in $t_f < t_0$ then the line is outside the view window.
- Otherwise, draw the line from the point $A + t_0\vec{V}$ to $A + t_f\vec{V}$.

The clipping algorithm works exactly the same in 3D, and can easily be implemented for a rectangular canonical view volume.

Algorithm: Liang-Barsky/Cyrus-Beck Clipping

Given: two lines and their parametric representations.

```
set t0 = 0 and tf = 1
for each clip boundary {left, right, top, bottom, front, back}
  calculate the corresponding pk and qk values
  if pk = 0
    if qk < 0 discard the line
    else continue to the next boundary

  set tk = qk / pk
  if pk < 0 then
    t0 = max( t0, tk )
  else
    tf = min( tf, tk )

  if t0 >= tf discard the line

draw the line defined by [t0, tf]
```

Example

Given: View window defined by $(0, 0)$ and $(100, 100)$, line from $(-20, 20)$ to $(80, 120)$.

1. Set $t_0 = 0$ and $t_f = 1$
2. Loop 0:
   - $p_1 = -V_x = -100$, $q_1 = A_x - x_{\text{min}} = -20$, $t_1 = 0.2$
   - $p_1 < 0$ so update $t_0 = \max(t_0, 0.2) = 0.2$
3. Loop 1:
   - $p_2 = V_x = 100$, $q_2 = x_{\text{max}} - A_x = 120$, $t_2 = 1.2$
   - $p_2 > 0$ so update $t_f = \min(t_f, 1.2) = 1.0$
4. Loop 3:
   - $p_3 = -V_y = -100$, $q_3 = y_{\text{min}} - A_y = 20$, $t_2 = -0.2$
   - $p_3 < 0$ so update $t_0 = \max(t_0, -0.2) = 0.2$
5. Loop 4:
   - $p_4 = V_y = 100$, $q_4 = y_{\text{max}} - A_y = 80$, $t_2 = 0.8$
   - $p_4 > 0$ so update $t_f = \min(t_f, 0.8) = 0.8$
6. Calculate the new endpoints of the line and draw it
   - $P_0 = A + 0.2\vec{V} = (-20, 20) + 0.2(100, 100) = (0, 40)$
   - $P_f = A + 0.8\vec{V} = (-20, 20) + 0.8(100, 100) = (60, 100)$

7.2 Polygons

A polygon is a piece of a planar surface, or at least it should be. Any planar surface can be represented using the plane equation.

$$Ax + By + Cz + D = 0 \tag{82}$$

The set of points (x, y, z) that satisfy this equation form a plane defined by the coefficients (A, B, C, D). We can derive the coefficients from any three points on the plane that are not collinear, as in (83). For example, the vertices of a proper triangle (the points are not collinear) are sufficient to calculate the coefficients of the plane equation. One benefit of using a triangle is that the points are guaranteed to lie on a plane. This is not not necessarily the case for polygons with more than three vertices.

$$\begin{aligned}
A &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\
B &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\
C &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\
D &= -x_1(y_2 z_3) - x_2(y_3 z_1 - y_1 z_3) - x_3(y_1 z_2 - y_2 z_1)
\end{aligned} \tag{83}$$

The surface normal to the plane is given by the vector $\vec{N} = (A, B, C)$. You can verify this by calculating the cross-product of the two vectors $\vec{v}_{21} = p_2 - p_1$ and $\vec{v}_{31} = p_3 - p_1$, which will produce the vector (A, B, C) as defined in (83). The surface normal at a point is the direction perpendicular to the surface. In general, we want the surface normal to point away from the surface, or to the outside of the polyhedron, if the surface is part of a larger shape.

7.2.1 Dot Products

The dot product, or inner product of two vectors is a useful computation in graphics. The computational definition of the dot product is the sum of the product of corresponding elements in the vector. The dot product also has a geometric interpretation.

$$\vec{A} \cdot \vec{B} = ||A||||B||cos(\theta) \tag{84}$$

Given two vectors $\vec{A}$ and $\vec{B}$, the dot product can be interpreted as the product of their lengths and the cosine of the angle between them. Therefore, the dot product of two orthogonal vectors is always zero. If both $\vec{A}$ and $\vec{B}$ are also unit length, then the dot product is just the cosine of the angle between them.

Whether the vectors are normalized or not, the sign of the dot product can tell us about the relationship of the two vectors. For example, consider a convex polyhedron where all of the surface normals point outwards from the middle of the shape. If we know the location of the viewer, then for each face we can calculate a vector from the center of that face to the center of projection. Call this vector the view vector $\vec{V}$. We can use the dot product of the surface normal $\vec{N}$ and the view vector $\vec{V}$ to calculate the visibility of each face of the polyhedron. If $\vec{V} \cdot \vec{N} \leq 0$ then the angle between the viewer and the surface normal is at least $90°$ and the face is not visible.

Visibility culling is a quick and easy way to reduce the number of polygons in the pipeline by half (on average), and is commonly executed in the canonical view volume along with clipping.

For concave polyhedra, visibility testing is insufficient to determine visibility if the dot product is positive. Other faces might be blocking the visibility of a convex face. Generally, graphics systems will break concave polyhedra into convex polyhedra in order to execute clipping or visibility testing.

7.2.2 Polygon Lists

Polygon lists are efficient ways to store lots of polygons. A polygon list is not simply an array of polygons, however. In most cases, polygons share vertices and edges, such as in polyhedra or triangular meshes. Rather than storing multiple copies of vertices–and their attendant data–it is more efficient to separate polygon definitions into separate lists of vertices, edges, and polygons.

- Vertex list: stores all of the vertices in a single array
  - 3D location
  - Surface normal information
  - Texture coordinates
  - Color/material property information
  - Transparency
- Edge list: stores all of the vertex pairs that make up edges
  - Indices of the two vertices linked by the edge
  - Interpolation data necessary for the scanfill algorithm
  - Links to all of the polygons that share the edge
- Polygon list: stores all of the edges sets that make up polygons
  - Links or indices for the constituent edges
  - Surface normal information for the polygon
  - Link to the texture map for the polygon
  - Might store color/material property information here
  - Bounding box information

If all of the polygons in a scene are stored in a single edge list, then the scanfill algorithm can work with the entire polygon list at once. Rather than rendering a single polygon at a time, it can handle all of the edges in the scene simultaneously. If we have depth information for each surface, it can always draw the polygon nearest to the viewer, enabling hidden surface removal with no extra work.

Many object models correspond to polygon meshes. The majority are triangular meshes, although quadrilateral meshes are sometimes used. A common arrangements of triangles and quadrilaterals is a **strip**.

- Triangle strips have N vertices and N-2 triangles.
- Quadrilateral strips have N vertices and $\frac{N}{2} - 1$ quadrilaterals

Another common arrangement of triangles is a **fan**, which is formed by a central point and a series of points in a radial arrangement. Circles and ellipses are commonly created using fans. A fan generates $N - 2$ triangles from N vertices.

Polygons other than triangles (and sometimes quadrilaterals) are rarely used to model objects. The problem is that polygons with more than three vertices are not guaranteed to be flat. A polygon that is not flat causes all sorts of problems with a rendering pipeline that is built upon the assumption of flat polygons.

Example: Building a cylinder from triangles

A cylinder is a commonly used shape in 3D modeling. We can build a cylinder out of one quadrilateral strip and two triangle fans.

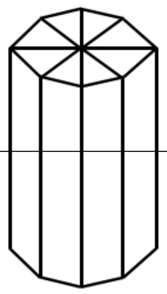


Figure 12: A cylinder created from a quadrilateral strip and two triangle fans.

7.3 Algebraic Surfaces

For many situations, approximate representations of surfaces are just fine. Polygons, for example, can adequately represent a sphere, but are not an exact representation. When modeling a physical object, it is often useful to have an explicit representation of the surface.

Implicit algebraic surfaces have the form $f(x, y, z) = 0$.

- Implicit algebraic surfaces can be visually complex, but have simple representations
- Implicit surfaces can be multi-valued functions–functions that have multiple solutions for a fixed set of parameters.

A general 2nd-degree implicit curve is given by (85).

$$ax^2 + 2bxy + cy^2 + 2dx + 2ey + f = 0 \tag{85}$$

Solutions to (85) form conic sections.

- If the conic section passes through the origin, then $f = 0$.
- If $c = 1$ then to define a curve segment you need five more constraints
  - Start and end points
  - Slope at the start and end points
  - A single point in the middle of the curve
- If $c = 1$ and $b = 0$ then a curve segment is defined by four more constraints
  - Start and end points
  - Slope at the start and end points

Implicit surfaces can be 2D curves or 3D surfaces. Some special cases of general 3D 2nd order implicit surfaces include the following.

Sphere

$$x^2 + y^2 + z^2 = r^2 \tag{86}$$

Ellipsoid

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{87}$$

Torus

$$\left[r - \sqrt{\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2}\right]^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{88}$$

Super-ellipsoid

$$\left[\left(\frac{x}{r_x}\right)^{\frac{2}{s_2}} + \left(\frac{y}{r_y}\right)^{\frac{2}{s_2}}\right]^{\frac{s_2}{s_1}} + \left(\frac{z}{r_z}\right)^{\frac{2}{s_1}} = 1 \tag{89}$$

With the super-ellipsoid, you can make a wide variety of shapes including octahedrons, cubes, cylinders, and ellipsoids.

The difficulty with implicit surfaces, however, is that they are difficult to render in a system geared towards polygons. For rendering systems based on ray casting–e.g. ray tracing–implicit surfaces are great because it is simple to intersect an implicit surface with a line. For standard graphics pipelines, however, they present difficulties. In general, the process of rendering an implicit surfaces is:

- Define a set of polygons on the surface, generally a triangular mesh
- Render the polygons

For many implicit surfaces–in particular convex surfaces–defining a triangular mesh is fairly simple. The nice thing about implicit surfaces is that controlling them is just a matter of modifying a few parameters in the defining equation. The whole rest of the process stays the same and the polygons move where they need to go to represent the surface.

Implicit surfaces can also be used as the model for a subdivision surface. For example, to model a super-ellipsoid, start by approximating the surface with an octahedron whose vertices sit on the implicit surface. Then recursively subdivide each triangular surface of the octahedron into four new triangles by creating a new vertex in the middle of each edge. Move each new vertex along a ray from the origin through the vertex–assuming the implicit surface is centered on $(0, 0)$–until it is on the implicit surface. The new set of polygons is a better representation of the super-ellipsoid. By recursively subdividing each polygon, the polygonal representation continues to improve.

7.4 Parametric Models

Parametric curves and surfaces are an alternative to implicit curves that also enable the creation of arbitrary shapes and surfaces, including multi-valued surfaces. Parametric curves have a single degree of freedom; parametric surfaces have two degrees of freedom. The general form of a parametric curve is given in (90).

$$P(t) = \begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix}^t \tag{90}$$

The tangent vector (analogous to the slope, but represented as a 3D vector) of the curve at any given point is given by (91).

$$P'(t) = \begin{bmatrix} x'(t) & y'(t) & z'(t) \end{bmatrix}^t \tag{91}$$

Since the parametric curve definition is a vector, you can transform the equations by a the standard matrix transforms to obtain transformed versions of the curve in closed form. Normally, however, transformations are executed after the curve is converted into line segments.

One method of converting a curve into line segments is to recursively subdivide it, subdividing any line segment where the deviation from the curve from the line is larger than a specified threshold.

Example: Circles

Circles are an example of a curve that can be represented parametrically. Two possible parametric representations are:

$$P(\theta) = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \qquad P(t) = \begin{bmatrix} \frac{1-t^2}{1+t^2} \\ \\ \frac{2t}{1+t^2} \end{bmatrix} \tag{92}$$

Using the parameterization by θ, equal spacing of θ generates equal arc-length segments around the circle. Using the second parameterization does not produce equal arc-length segments for equally spaced values of t, but the result is close and the function is computationally simpler.

7.5 Splines

Splines were created to allow designers to create natural looking shapes by specifying just a few control points. The inspiration for splines was the woodworking technique of bending a thin piece of wood to make a curve, using a small number of metal or stone ducks connected to the wood to control the shape. On computers, control points take the place of the ducks, and a polynomial represents the continuous function of the thin piece of wood.

The properties of a spline are all defined by its complexity and its control points. In some cases, other parameters may be added to affect certain properties of the curve. A single spline defines a curve in 2D or 3D space. Using splines in orthogonal directions, it is possible to generate 3D surfaces.

The relationship of the control points to the spline is an important attribute. Some spline definitions require the curve to go through the control points, in which case the spline is said to **interpolate** the control points. Other spline definitions do not require the curve to go through some or all of the control points, which means the curve **approximates** the control points.

Interpolating splines are good for situations where the curve has to go through certain locations in the scene. Animation, for example, may require an object to be in exact locations at certain times. Approximating splines, on the other hand, can be better for fitting noisy data, or for free-form drawing where the designer is suggesting a shape rather than enforcing one.

Approximating splines may also have the property that the curve is guaranteed to fit within the **convex hull** of the control points. The convex hull of a set of points is the minimal convex polygon that contains all of the points in the set. The vertices of the convex hull will be points within the set. A convex hull can have many uses. One use, for example, is intersection detection in games. It is possible to quickly ascertain whether a point is inside a convex polygon (but not a concave polygon) by testing whether the point is on the interior side of each edge. Convex hulls are often used as bounding polygons in 2D games to calculate intersections with more complex, concave objects.

To approximate a complex curve, graphics systems often use multiple joined splines rather than a higher order polynomial. The degree of **continuity** between adjacent spline curves is an important property.

- Zero-order continuity: the splines meet, but there may be a sharp corner at the join
- First-order continuity: the splines meet and the tangent directions are identical at the join
- Second-order continuity: the splines meet and the rate of change of the curve tangents are identical at the join

For graphics and most physical control systems, second-order continuity is sufficient to avoid jerkiness. First order continuity can result in jerkiness (think about alternately hitting the gas and the brake on a car) if used to control physical motion (e.g. animations, or a camera).

Splines can be represented in three (mathematically identical) ways. The different representations are useful for different tasks.

- The set of boundary and control point conditions defining the spline. These are useful for graphic design and spline control by people.
- The matrix that characterizes the spline. This is useful for manipulating and drawing the spline.
- The set of blending functions that characterize the spline. These are useful for understanding how each control point affects the shape of the curve.

7.5.1 Cubic Splines

Cubic splines are collections of 2nd-order curves. Cubic splines can have up to 2nd-order continuity. They are defined by a set of 3rd-order polynomials in (x, y, z).

$$\begin{aligned} x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned} \tag{93}$$

We can write the same equation in matrix form, which separates out the functions of the parameter u from the coefficients $(\vec{a}, \vec{b}, \vec{c}, \vec{d})$.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} u^3 \\ u^2 \\ u \\ 1 \end{bmatrix} \tag{94}$$

If we have a control point at each end of a curve segment, then $N + 1$ control points defines N curve segments. For each curve segment, there are 4 unknowns per dimension, or 12 unknowns for a 3D curve.

Consider a single 3D spline with 2 control points.

- The control points each provide 3 constraints, one in each dimension, for a total of 6 equations.
- If we also define the tangent (direction) of the curve at each control point, that provides another 3 constraints per point, for a total of 12 equations.

Therefore, if we have the ability to specify two control points and an orientation at each control point, we can define a unique cubic spline connecting them.

7.5.2 Natural Cubic Splines

Natural cubic splines have the following properties.

- $N + 1$ points defined N curve segments
- Interpolating: the curve goes through the control points
- C2 continuity: both the tangents and their rate of change the control points are identical for adjacent splines

We need $4N$ constraints in order to define all of the curve segments. The above conditions provide almost enough constraints to calculate a unique curve. Each of the $N - 1$ interior point provides 4 constraints per dimension, for a total of $4N - 4$ constraints per dimension.

- Curve i ends $(u = 1)$ at point P_i
- Curve $i + 1$ begins $(u = 0)$ at point P_i
- The tangents of curves i and $i + 1$ are identical at $u_i = 1$ and $u_{i+1} = 0$
- The rates of change of the tangents of curves i and $i + 1$ are identical at $u_i = 1$ and $u_{i+1} = 0$

In addition, the exterior control points provide an additional 2 constraints per dimension.

- Curve N ends $(u = 1)$ at point P_{N+1}
- Curve 1 begins $(u = 0)$ at point P_1

The final two constraints are generally provided by constraining the tangents of the exterior control points. All of the constraints can be written in a single large matrix, solving for the N sets of spline coefficients $(\vec{a}, \vec{b}, \vec{c}, \vec{d})$. One implication of needing to solve for all of the spline coefficients simultaneously is that each control point affects the shape of the entire curve. It's not possible to move one control point and only affect a portion of the curve. The blending function for each control point, therefore, extends from the start to the end of the spline. This is not necessarily the case for every type of spline, but it is true for natural cubic splines. The result makes sense intuitively if you think about the spline as a thin piece of wood. You can't move one part of the piece of wood without the effect modifying its entire shape, even if it's only a small amount.

7.5.3 Hermite Splines

Hermite splines are a variation on natural cubic splines.

- Hermite splines interpolate the set of control points
- Each spline is defined by two control points and two tangent vectors
- Each control point only affects the two splines it anchors

We need 4 constraints per dimension to define a cubic polynomial control function. The location of the two control points (p_k, p_{k+1}) and the tangent vectors at each control point (Dp_k, Dp_{k+1}) provide the four constraints. Note that the tangent vectors for a spline are defined by the derivative of the spline parametric functions.

$$\begin{aligned} P(u) &= \left[\begin{array}{ccc} H_x(u) & H_y(u) & H_z(u) \end{array}\right] \\ P'(u) &= \left[\begin{array}{ccc} H'_x(u) & H'_y(u) & H'_z(u) \end{array}\right] \end{aligned} \tag{95}$$

$$\begin{aligned} H(u) &= au^3 + bu^2 + cu + d \\ H'(u) &= 3au^2 + 2bu + c \end{aligned} \tag{96}$$

$$\begin{aligned} P(0) &= p_k \\ P(1) &= p_{k+1} \\ P'(0) &= Dp_k \\ P'(1) &= Dp_{k+1} \end{aligned} \tag{97}$$

Using the matrix form of the spline equation, it's straightforward to write the four constraints. The general matrix form is given in (98).

$$\begin{aligned} P(u) &= \left[\begin{array}{cccc} u^3 & u^2 & u & 1 \end{array}\right]\left[\begin{array}{cccc} a & b & c & d \end{array}\right]^t \\ P'(u) &= \left[\begin{array}{cccc} 3u^2 & 2u & 1 & 0 \end{array}\right]\left[\begin{array}{cccc} a & b & c & d \end{array}\right]^t \end{aligned} \tag{98}$$

At the control points u is either 0 or 1, so the four constraint equations generate the matrix shown in (99). Note that we know the control points and the tangent vectors (provided by the user), so the only unknowns are the coefficients of the Hermite polynomial.

$$\left[\begin{array}{c} p_k \\ p_{k+1} \\ Dp_k \\ Dp_{k+1} \end{array}\right] = \left[\begin{array}{cccc} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{array}\right]\left[\begin{array}{c} a \\ b \\ c \\ d \end{array}\right] \tag{99}$$

Solving for the coefficients simply requires inverting the square matrix and pre-multiplying both sides by the inverse. The inverse is the characteristic matrix for a Hermite spline. We can then substitute the product of the control conditions and the Hermite matrix back into the Hermite equation in (98).

$$P(u) = \left[\begin{array}{cccc} u^3 & u^2 & u & 1 \end{array}\right]\left[\begin{array}{cccc} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{array}\right]\left[\begin{array}{c} p_k \\ p_{k+1} \\ Dp_k \\ Dp_{k+1} \end{array}\right] \tag{100}$$

(100) is a complete description of the Hermite spline between the two control points and their two tangent vectors. To find any point on the spline, simply put the u value into the equation and execute a matrix multiplication and a matrix-vector multiplication. Note that the right-most matrix is 4xN, where N is the number of dimensions, since each point and each tangent vector represent points in the N-dimensional space.

Each spline variation has its own version of the matrix equation with a characteristic matrix.

7.5.4 Cardinal Splines

One example of a spline variation is the Cardinal spline.

- Cardinal splines interpolate the set of control points.
- The tangent at control point p_k is defined by its adjacent control points p_{k-1} and p_{k+1}.
- There are two extra control points at the end of the spline that determine the tangent at the actual end points.
- The spline definition also contains a term t that controls the tension within the spline.

The constraints can be expressed as follows.

$$\begin{aligned}
P(0) &= p_k \\
P(1) &= p_{k+1} \\
P'(0) &= \frac{1}{2}(1-t)(p_{k+1} - p_{k-1}) \\
P'(1) &= \frac{1}{2}(1-t)(p_{k+2} - p_k)
\end{aligned} \tag{101}$$

We can set up and solve for the characteristic matrix for Cardinal splines, just as we did for Hermite splines. Letting $s = \frac{1-t}{2}$, the resulting matrix equation for Cardinal splines is given in (102).

$$P(u) = \left[\begin{array}{cccc} u^3 & u^2 & u & 1 \end{array}\right] \left[\begin{array}{cccc} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{array}\right] \left[\begin{array}{c} p_k \\ p_{k+1} \\ \frac{1}{2}(1-t)(p_{k+1} - p_{k-1}) \\ \frac{1}{2}(1-t)(p_{k+2} - p_k) \end{array}\right] \tag{102}$$

Because the adjacent control points determine the tangent orientations, moving a control point affects two curve segments to either side. The tension parameter permits more or less loopy splines, but the degree of control is not as great as with Hermite spline since the tangents at each control point cannot be specified directly. Unlike the Hermite spline, however, the only constraints required are the control points, which completely determine the curve. The interface for controlling a Cardinal spline can, therefore, be simpler.

There are a number of other interpolating spline types that provide additional parameters or different control techniques. In general, which spline to use depends upon the needs to the designer. If smoothness is critical, natural cubic splines guarantee smooth second derivatives, while Hermite and Cardinal splines do not. If a simple user interface is required, Cardinal splines require only control points.

7.6 Bezier Curves

Bezier curves are approximating splines that are useful for free-form design. They are commonly used in computer graphics for both curves and surfaces because there is a fast and simple algorithm for converting them to line segments/polygons for drawing. The classic teapot is an example of a surface defined by a set of Bezier curves.

Bezier curves have the following properties.

- The curve goes through the first and last control points
- All curve characteristics are determined by the control points
- The curve lies within the convex hull of the control points
- Every point on the curve is a weighed sum of the control points
- All control points affect the entire curve

The order of a curve is determined by the number of control points. A curve of order n will have $n + 1$ control points labeled from 0 to $k = n$. A cubic Bezier curve, for example, is order 3 and has 4 control points.

Since every point on the curve is a weighted sum of the control points, one way of defining the curve is to define the function that specifies the weight of each control point along the curve. The Bezier basis functions are the Bernstein polynomials.

$$BEZ_{k,n}(u) = \frac{n!}{k!(n-k)!}u^k(1-u)^{n-k} \tag{103}$$

The four blending functions for a cubic Bezier curve, for example, are given below and shown in figure 13.

$$\begin{aligned} BEZ_{0,3}(u) &= (1-u)^3 \\ BEZ_{1,3}(u) &= 3u(1-u)^2 \\ BEZ_{2,3}(u) &= 3u^2(1-u) \\ BEZ_{3,3}(u) &= u^3 \end{aligned} \tag{104}$$

The characteristic matrix for a cubic Bezier curve is given by (105).

$$M_{BEZ} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{105}$$

The complete definition of the cubic Bezier curve is given by the characteristic matrix, the u vector, and a matrix of the four control points.

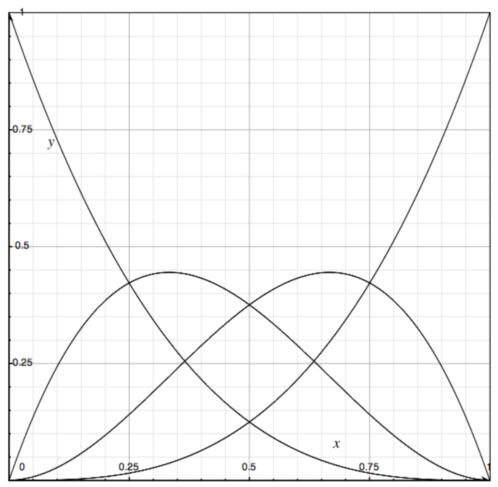


Figure 13: Bezier basis functions for a 3rd order curve.

$$P(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{0,x} & p_{0,y} & p_{0,z} \\ p_{1,x} & p_{1,y} & p_{1,z} \\ p_{2,x} & p_{2,y} & p_{2,z} \\ p_{3,x} & p_{3,y} & p_{3,z} \end{bmatrix} \tag{106}$$

7.6.1 Drawing Bezier Curves

One method of drawing Bezier curves is to take small steps in u and draw lines between the resulting points. The computation requires a large number of multiply and add operations. It turns out we can do much better taking a different approach to drawing the curves. The primary observation that enables us to speed up the process is that the control points are an approximation to the curve. Drawing straight lines between the control points is not an unreasonable thing to do, if the control points are close enough to the curve.

It still doesn't make sense to follow this approach unless we can somehow add control points quickly. It turns out that any Bezier curve can be subdivided in half, with the same number of control points for each half as were on the original. The new curve is identical to the old one, but there are now two curves using $2N - 1$ control points instead of one curve with N control points.

de Casteljau Algorithm

The de Casteljau algorithm is based on the fact that any point on the curve's surface can be defined as a recursive procedure on the control points.

$$p_i^r = (1-u)p_i^{r-1} + u p_{i+1}^{r-1} \tag{107}$$

The level of the control points is given by $r \in [1, n]$, where n is the order of the curve. The initial control points are given by p_i^0 where $i \in [0, n]$. The point $p_0^n(u)$ is the point on the curve at parameter value u.

The level 0 control points are the original control points. The level 1 control points are defined as weighted sums of the level 0 control points, and so on. At each higher level, there is one less control point. When calculating the value of the curve, the p_i^r defined along the way are the control points for the curve from $P(0)$ to $P(u)$ and from $P(u)$ to $P(1)$.

The interesting case is when we calculate $P(\frac{1}{2})$. The resulting control points are given by (108) and shown in figure 14

$$
\begin{aligned}
q_0 &= p_0 \\
q_1 &= \frac{1}{2}(p_0 + p_1) \\
q_2 &= \frac{1}{2}q_1 + \frac{1}{4}(p_1 + p_2) \\
q_3 &= \frac{1}{2}(q_2 + r_1) \\
r_0 &= q_3 \\
r_1 &= \frac{1}{2}r_2 + \frac{1}{4}(p_1 + p_2) \\
r_2 &= \frac{1}{2}(p_2 + p_3) \\
r_3 &= p_3
\end{aligned} \tag{108}
$$

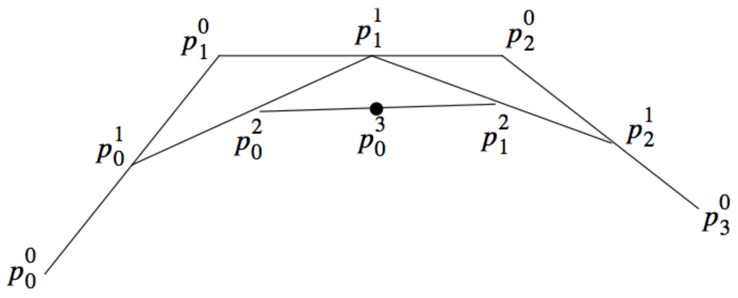


Figure 14: Control points for a Bezier curve divided in half.

Since the control points define the convex hull of the curve, as they get closer together the difference between the control points and the curve gets smaller. The subdivision stops when the control points are sufficiently close or at some arbitrary level of subdivision. It is also possible to implement an adaptive subdivision scheme where the subdivision for a particular curve segment can stop when the two inner control points are close enough to the line connecting the outer two control points.

7.6.2 Bezier Surfaces

Bezier surfaces are defined by orthogonal sets of Bezier curves. The same properties apply to Bezier surfaces as apply to the curves.

- The control points approximate the surface.
- The surface is completely defined by the control points.
- The surface goes through the control points on the corners.
- The surface lies within the convex hull of the control points.

$$P(u,v) = \sum_{j=0}^{m}\sum_{k=0}^{n} p_{j,k} BEZ_{j,m}(v) BEZ_{k,n}(u) \tag{109}$$

Drawing Bezier surfaces is slightly more complex than drawing curves, but not much. The de Casteljau algorithm provides a fast way to subdivide the defining curves and generate a large set of control points. The resulting control points create a dense grid of small Bezier surfaces. When the surfaces are small enough, we can generate and draw two triangles for each surface, using only the four corner control points. Uniform subdivision is simple to implement, although it can be more costly than necessary for simple surfaces.

Adaptive subdivision stops the process when the four inner points are well approximated by the four corners. The problem with adaptive subdivision is that it can result in cracks occurring along the surface. If one patch stops subdividing, but its neighbor does not, then edge connecting the two patches is no longer identical. The patch that continued subdividing will write multiple triangles along the edge, some of which may not line up exactly.

One solution is to enforce collinearity of the control points along an edge that has stopped subdividing. While this generates a slightly different surface, the approximation is generally not noticeable.

The tricky part of the subdivision process is that it is view space that is relevant to the subdivision, not world or model space. Therefore, the subdivision algorithm must have access to the VTM, assuming the points have been transformed to the canonical view volume space. Given the VTM, the process can proceed as follows.

- Transform the control points to screen space.
- Compute the bounding box in screen space.
- If the largest dimension of the bounding box is small enough, create two triangles from the four corner control points.
- Else, split the surface into four surfaces using the de Casteljau process.

When we get to shading, we will need to obtain the surface normals for the Bezier surface. As noted above, the tangent of a spline is given by the derivative of the parametric equations. The Bezier surface equation can be written as a matrix equation using the Bezier characteristic matrix.

$$P(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M_{BEZ} \begin{bmatrix} p_{0,x} & p_{0,y} & p_{0,z} \\ p_{1,x} & p_{1,y} & p_{1,z} \\ p_{2,x} & p_{2,y} & p_{2,z} \\ p_{3,x} & p_{3,y} & p_{3,z} \end{bmatrix} M_{BEZ}^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{110}$$

Therefore, the tangent plane (a surface has a tangent plane, rather than a vector) is defined by the partial derivatives of (110) in u and v.

$$\frac{\delta}{\delta u}P(u,v) = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} M_{BEZ} \begin{bmatrix} p_{0,x} & p_{0,y} & p_{0,z} \\ p_{1,x} & p_{1,y} & p_{1,z} \\ p_{2,x} & p_{2,y} & p_{2,z} \\ p_{3,x} & p_{3,y} & p_{3,z} \end{bmatrix} M_{BEZ}^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{111}$$

$$\frac{\delta}{\delta v}P(u,v) = \begin{bmatrix} u^3 & u^2 & u & 0 \end{bmatrix} M_{BEZ} \begin{bmatrix} p_{0,x} & p_{0,y} & p_{0,z} \\ p_{1,x} & p_{1,y} & p_{1,z} \\ p_{2,x} & p_{2,y} & p_{2,z} \\ p_{3,x} & p_{3,y} & p_{3,z} \end{bmatrix} M_{BEZ}^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix} \tag{112}$$

The cross product of the two tangent vectors gives the surface normal at that point on the surface.

7.7 Subdivision Surfaces and Fractal Objects

Octahedrons to spheres, ellipsoids, or asteroids

Triangles to potato chips

Pyramids to mountains

8 Hidden Surface Removal

Definitions:

- Surface Normal $\vec{N}$ - points away from the surface, orthogonal to the tangent plane
- View Vector $\vec{V}$ - points from the surface to the viewer
- Light Vector $\vec{L}$ - points from the surface to the light source

8.1 Backface Culling

As noted previously, we can use the surface normal and view vector of a polygon to determine whether it is facing the viewer or not. If $\vec{N} \cdot \vec{V} \leq 0$ then do not draw the polygon. The same concept can be used to cull lines on the back side of a polyhedron: just draw the lines associated with polygons that face the viewer.

Backface culling is an important precursor to other hidden surface removal clipping processes and generally eliminates half of the polygons in a scene. Not all polygons should undergo backface culling, however. Sometimes it is important to be able to see both sides of a polygon if represents a two-sided surface such as a wall. In general, each polygon will have a cull flag that indicates whether it should be removed.

8.2 Painter's Algorithm

The Painter's algorithm is a simple approach to hidden surface removal: sort the polygons and then draw them from back to front. For many scenes, it works well enough, and it is useful for quick renderings of a scene.

1. Sort the polygons by depth from the viewer–generally in CVV space where depth is given by the z-coordinate
2. For polygons that overlap in depth
   - If the bounding boxes are disjoint in any projection onto a face of the CVV bounding cube, order doesn't matter.
   - If the projections onto the view plane do not overlap, order doesn't matter
   - if one polygon is completely in front of the other relative to the viewer, the polygons do not need to be split.
   - If the polygons intersect, either divide them at the intersection line or recursively cut them in half until there are no intersections

Since sorting is $O(N \log N)$, the approximate cost of rendering is not much more than linear in the number of polygons. We have to re-sort the polygons every time we change viewpoints.

8.3 BSP Trees

We'd like to have a method of drawing that invariant to viewpoint so that all we have to do during drawing is traverse the list of polygons ($O(N)$ process).

- The data structure must be invariant to viewpoint
- The polygons must still be sorted spatially
- All intersections must be eliminated

Binary space partition [BSP] trees are one method of creating an $O(N)$ drawing algorithm.

Consider a set of polygons

- Each polygon is apiece of an infinite plane that splits the world into two
- The viewer is on one side of the world
- All the polygons on the same side as the viewer are 'in front' of the dividing polygon
- All the polygons on the opposite side as the viewer are 'in back' of the dividing polygon

Given a single polygon, we want to draw all the polygons 'in back' before we draw the polygons 'in front'. If the entire scene is organized into a single tree, then we can traverse the tree following this simple rule and draw all the polygons back to front in $O(N)$ time. We don't really care how long it takes to build the BSP tree, because it only gets built once off-line.

How do we figure out what side of the polygon the viewer is on? Remember the plane equation?

$$f(\vec{p}) = Ap_x + Bp_y + Cp_z + D \tag{113}$$

If $f(\vec{p}) = 0$ then the point $\vec{p}$ is on the plane. If $f(\vec{p}) < 0$ then it's one one side of the plane, and if $f(\vec{p}) > 0$ it's on the other side. So we can now phrase the 'in front' and 'in back' rules relative to a dividing polygon in terms of the plane equation.

- Draw polygons with the opposite sign for $f(\vec{p})$ as the viewer.
- Draw the dividing polygon.
- Draw the polygons with the same sign for $f(\vec{p})$ as the viewer.

Drawing Algorithm

```
function draw(BSPTree bp, Point eye)
  if( empty(bp) )
    return

  if( f(bp->node, eye) < 0 )  // viewer on negative side
    draw( bp->negative, eye );
    drawPolygon( bp->node );
    draw( bp->positive, eye );
  else
    draw( bp->positive, eye );
    drawPolygon( bp->node );
    draw( bp->negative, eye);
```

8.3.1 Building BSP Trees

We can build the BSP tree by picking a polygon as the root and then inserting the remaining polygons into the tree. To find the location for a new polygon, traverse down the tree using the plane equation test to find the leaf on which to add it.

- Intersecting polygons must be split prior to building the tree
- The data structure is independent of viewpoint
- The order in which polygons are inserted can significantly affect shape of the BSP tree
- Since the tree is only traversed during drawing, balance doesn't matter
- Order of insertion can matter since in realistic scenes a polygon in the tree will divide at least one of its descendants. Intersected descendants must be split in order to be inserted properly into the tree.

The basic idea is to calculate the intersection between the plane and the polygon. The intersection forms a line connected two of the edges. For the base case of a triangle the intersecting line divides two of the vertices from the third.

- The pair of vertices on the same side of the dividing plane are a and b.
- The separated vertex is c
- The ends of the intersecting line are A and B.

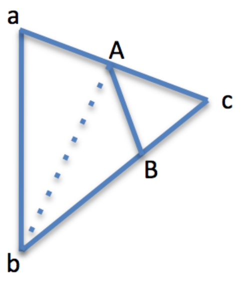


Figure 15: Example of an intersection line on a triangle.

Given the situation shown in figure 15, we can create three new triangles (a, b, A), (b, B, A), and (c, A, B). Note that the ordering of the points matters in order to keep the surface normals pointing the same direction as the original triangle.

To find the intersection points, use a parametric representation of each edge of the triangle and find the parameter of the intersection with the dividing plane. If the intersection parameter is between 0 and 1, then the edge intersects the plane. Note that it's important to handle vertices that are very close to the dividing plane properly. If, for example, the point c is within ϵ of the dividing plane, the splitting the polygon creates two large triangles and one almost non-existent one. A better strategy is to force any vertex within ϵ of the dividing plane onto the dividing plane for the purposes of comparison. Then if the three points have the same sign for the plane function or are zero, they don't need to be split.

We only need to worry about intersections between a polygon and its descendants, but that means different trees will have different numbers of polygons. Since drawing depends on the number of polygons, common practice is to create several BSP trees and then pick the one with the smallest N.

8.4 Z-buffer Algorithm

The z-buffer algorithm is an alternative $O(N)$ hidden surface removal algorithm that trades memory space for time. The basic idea is to keep track of the closest object at each pixel. If a new polygon is inserted into the scene, only the parts of the polygon that are in front of the current closest surfaces are drawn.

The basic Z-buffer algorithm is as follows.

1. Create a depth buffer the size of the image
2. Initialize the depth buffer to the depth value of the back clip plane (1)
3. Initialize the image buffer to the background color
4. For each polygon
   (a) Draw the polygon using the scanline fill algorithm
      - Interpolate the depth value along each edge in addition to the x-intersect
      - Interpolate the depth value across each scanline
      - Discard the pixel if its depth value is greater than the existing depth value
      - Discard the pixel if its depth value is less than the front clip plane depth $F' = (d + F)/B'$

The z-buffer algorithm is easily implemented in hardware or software and provides a fast and efficient way to handle intersecting surfaces. There is no need to divide triangles or polygons because the depth values are calculated and tested at each pixel.

Note that when two differently colored surfaces join at an edge there is an ambiguity about which surface to draw. This can result in stippling along edges with odd colors popping up randomly. One way to solve this problem is to store, in addition to the depth of the particular point, the depth of the center of its polygon. Then, if a pixel is within ϵ of the current depth, only draw it into the z-buffer if its polygon center is also closer to the viewer.

8.4.1 Handling Perspective Viewing

The scanline fill algorithm linearly interpolates values from one vertex to another along each edge and then across each scanline. The interpolation takes place in image space, meaning the derivative values are measured with respect to pixels. Consider the case of railroad tracks in perspective viewing. When the tracks are near the viewer, stepping up one scanline corresponds to a small change in the depth of the tracks. When the tracks are approaching the horizon, however, stepping up one scanline corresponds to a large change in the depth of the tracks because they are compressed visually into a small area of the image. Linear interpolation does not correctly calculate the z-values under perspective viewing. Parallel projection does not have the same problem.

The way to fix the problem is to use $\frac{1}{z}$ in place of z when working with depth values. Since perspective projection divides x and y by z, the expression $\frac{1}{z}$ interpolates linearly in image space.

- Store $\frac{1}{z}$ in the depth buffer instead of z values
- Initialize the depth buffer to $\frac{1}{B'}$, or the inverse of the depth of the back clip plane.
- Calculate, store, and interpolate $\frac{1}{z}$ in the scanline fill algorithm when drawing polygons

- Discard a pixel if its $\frac{1}{z}$ value is less than the current value in the depth buffer.
- Discard a pixel if its $\frac{1}{z}$ value is greater than $\frac{1}{F'}$.

The z-buffer algorithm can be made faster if the polygons are approximately sorted from front to back. Drawing the polygons in front first reduces the number of shading calculations and assignments to the image. Polygons that are behind others will scan but not draw. It is quite reasonable to build a BSP tree and then execute z-buffer rendering. It's not necessary to split polygons in the BSP building process for z-buffer rendering, since the need for sorted polygons is only approximate.

8.5 A-buffer Algorithm

A variation on z-buffer rendering is the A-buffer algorithm. A standard z-buffer doesn't permit proper transparency or other effects. An A-buffer solves this problem by storing pointers to polygons in addition to their depth values.

1. Initialize an A-buffer of linked lists to all empty lists
2. Initialize the image to the background color
3. For each polygon
   (a) Use scanline fill to interpolate depth values across the polygon
   (b) At each pixel
      - Try inserting a pointer to the polygon and its depth value into the linked list
      - If the polygon will be inserted behind an opaque polygon, discard it
      - Otherwise, insert the polygon at the appropriate location in the list
4. For each pixel
   - Calculate the color at each pixel using a recursive coloring scheme where t_i is the transparency of polygon p_i and p_0 is the first polygon in the list.

     $C_i = (1 - t_i)C_{p_i} + t_i C_{i+1}$

8.6 Simultaneous Polygon Rendering

Using z-buffer rendering, it is possible to build a single scanline fill algorithm that handles all polygons simultaneously. The algorithm takes in a complete list of the visible polygons after backface culling. The algorithm then processes each polygon's edges, building the complete edge list for all polygons.

The major change to the algorithm is the addition of an active polygon list while processing each scanline. The active polygon list is the set of polygons covering the current pixel. For each scanline, the active polygon list is set up on the left side and updated at every edge that intersects the scanline. At each pixel, the algorithm increments the depth values for all active polygons and resorts the lists by depth. Because the algorithm keeps track of all polygons covering a pixel, it makes the shading calculation only once for each pixel and can incorporate both transparency and other effects.

9 Illumination and Reflection

The system we've put together so far tell us where stuff is in the image and what surfaces are visible. Now we have to decide what the surface should look like. The appearance of a surface is a function of the color of the surface, the color of the light hitting the surface, and the relative geometry of the viewer, surface, and light sources.

When modeling colors for shading calculations, it is important to set up the material and illumination colors within the range $[0, 1]$. The surface colors then correctly represent the percent of each color reflected from the surface. Light sources can have values greater than 1, but that often results in saturation of the image as there are often multiple light sources in the scene, and their contributions are summed to obtain a final result.

If we are using a floating point representation of colors in a scene, then we can scale the result as needed to put it into a displayable range. The field of high dynamic range imaging uses a variety of techniques to create a viewable representation of an image with many times the dynamic range available on display systems. Those can be applied to an computer graphics image as easily as to a digital photograph.

9.1 Modeling light sources

There are many kinds of light sources in the world. A general representation of light sources is possible, but difficult to implement. Instead, graphics tends to use a small number of light source types that are easily parameterized. While the models can be restrictive relative to the real world, appropriate combinations of the simple models can lead to realistic and effective lighting.

Note that for all lighting models, the intensity of the light source should be in the range $[0, 1]$.

9.1.1 Ambient light

A lot of light is reflected from surfaces in a scene, scattered in many directions. While ambient light varies, and is affected by the relative geometry of surfaces, capturing that variation is extremely challenging. An approximation to real ambient illumination is to model it as a constant intensity source with no dependence on geometry. Ambient lighting is simple to represent.

- The illumination color $C_a = (R_a, G_a, B_a)$

9.1.2 Directional light

Real light comes from a particular light source. A simple approximation to a light source like the sun is to say that all of the light rays are coming from the same direction. Another way to think about it is that the light source is at infinity (or a very long distance from the scene). Directional light is convenient because the light vector is constant across the scene, reducing the amount of computation required. When using parallel projection, a useful light source direction is to use the DOP, which guarantees that all visible surfaces are lit. Directional lighting can be represented using two fields.

- The illumination color $C_d = (R_a, G_a, B_a)$
- The illumination direction $D_d = (d_x, d_y, d_z)$

9.1.3 Point light sources

Point light sources represent light sources within a scene. The light vector must be computed for each scene location since it is different for every location. A useful location for a point light source is the COP, which guarantees that all visible surfaces are lit. A light source anywhere else will (should) cause shadows that are visible to the viewer. A point light sources requires two fields to represent.

- The illumination color $C_p = (R_a, G_a, B_a)$
- The illumination location $P_p = (p_x, p_y, p_z)$

9.1.4 Spot light sources

Often, lights have lamp shades that control the spread of illumination. We may also want to simulate an actual spotlight in a scene (for example, on the bridge of the Enterprise). A spot light has a color, a location, a direction, and an angle of spread.

- The illumination color $C_s = (R_a, G_a, B_a)$
- The spot light location $P_s = (p_x, p_y, p_z)$
- The spot light direction $D_s = (d_x, d_y, d_z)$
- The spread angle α

To calculate the visibility of the light source from a surface point $P = (x, y, z)$, do the following.

1. Calculate the light vector $L = P_s - P$
2. Calculate the dot product of the negative light vector and the spot light direction $t = (-L) \cdot D_s$
3. If $t < \cos(\alpha)$ then the spot is not visible from the surface point P

The above procedure produces a sharp cutoff at the edge of the spotlight. An alternative that creates a soft falloff is to make the light source intensity proportional to $cos^n(\alpha)$ where n determines the sharpness of the falloff.

9.1.5 Area light sources

Area light sources can be as complex as necessary to model the scene. Two simple examples of area light sources are polygons and spheres. Both are simple to sample with shadow rays and the combination can model a large number of realistic light sources. Note that the projection of a sphere is always a circle, making the sampling process dependent only upon the visible radius of the sphere, which falls off as $\frac{1}{d}$ where d is distance to the light source.

Area light sources require sampling, regardless of whether the system incorporates shadows.

9.2 Implementing light sources

Light sources are part of your scene as much as polygons, lines, or points. Nevertheless, you can always take a shortcut and insert the light sources in World space, make the shading calculations for each vertex in world space, and make the light sources independent of any modeling transformations. If you are lighting the scene with a light source at the viewer location $(0, 0, 0)$ or using a directional light source, no transformations are required.

However, often we want to attach lights to locations in the scene. Lamps should have their bulbs screwed in. Wall sconces need to stay on the wall, or they may get copied to multiple locations along a corridor. The obvious way to implement light sources is to integrate them into the hierarchical modeling system. The not so obvious problem is that while traversing your model, you may want to render polygons before you know where all the light sources are located.

The solution is to make a light source pass through your module prior to making a rendering pass. Note that this makes yet another argument for separating your module traversal from the actual polygon drawing stage. The module traversal stage calculates all of the light source locations (in CVV space) and generates a list of the visible polygons (in screen space with either colors or surface normals and CVV coordinates at each vertex) that need to be drawn. The drawing stage then writes the polygons to the image using all of the appropriate shading.

9.3 Modeling Reflection

Reflection from a surface is a complex phenomenon. The light we see coming from a surface point results from a number of different interactions between light and matter. Some surfaces, or surface types are simpler than others. The reflection from a shiny metal mirror, for example, can be explained fairly simply using a single type of reflection. The reflection of a piece of velvet, however, exhibits many different kinds of reflection with non-intuitive properties.

As with most things in graphics, there are a range of models for calculating reflection. Most of them are variations on or combinations of simple models that achieve effects that look good enough. However, to accurately model the appearance of velvet, grass, brass, or clouds, much more complex reflection (and transmission) researchers have developed complex models designed to capture the actual physics of the light-matter interaction.

9.3.1 Matte materials

Matte reflection occurs when light passes into a material, interacts with pigment particles, and is reflected in random directions. The interaction between the light and the pigment particles removes some fraction of the light energy. When the amount of energy removed from the incoming light is different for different wavelengths, the the surface appears colored. The material, therefore, acts as a filter on the incoming light. We can represent the filtering action as a band-wise multiplication of the incoming light energy and the color of the material. In general, materials do not add energy to a wavelength, so materials color values should be in the range [0, 1].

Note that because the outgoing illumination scatters randomly, the appearance of the surface is not dependent upon the viewing direction. The brightness of the surface is only dependent upon the amount of energy striking the surface. The energy per unit area coming from a light source is dependent upon the orientation of the surface normal relative to the light source. If the surface normal is pointing directly at the light, the surface is receiving the maximum amount of energy. At angles greater than $90°$, the surface receives no energy. The energy per unit area follows a cosine curve.

A simple model of matte reflection is Lambertian reflection, parameterized by the light source color C_L, the body color C, and the angle between the surface normal $\vec{N}$ and the light source direction $\vec{L}$

$$I = C_L C_b cos\theta = C_L C_b (N \cdot L) \tag{114}$$

Note that a fluorescent material takes energy from one wavelength and converts it to a different wavelength. A simple multiplication cannot model fluorescence. Instead, the color of fluorescent materials must be represented as a 3x3 matrix for RGB models to allow for cross-band effects.

9.3.2 Inhomegeneous dieletrics

Inhomogeneous (more than one material) dielectric (non-conductive) materials constitute a significant number of materials we encounter. Paint, ceramics, cloth, and plastic are all examples. Each has some kind of substrate material that is generally clear with embedded pigment particles that impart color. There are good manufacturing reasons for favoring such materials–the substrate can be the same across all paints, for example, with the only difference between paints being the type of pigment.

Inhomogeneous dielectrics exhibit two kinds of reflection. Just like matte materials, some of the incoming energy is absorbed by the pigment particles and scattered in random directions. Some of the energy, however, is reflected at the boundary with the substrate. The surface reflection we tend to perceive as highlights on the object, and the color of the surface reflection is generally the same as the illuminant since the substrate material generally exhibits the property of neutral interface reflection: it doesn't change the color of the incoming light.

Unlike body reflection, surface reflection depends upon the viewing angle. Changing your viewpoint on a shiny object causes the location of the highlights to change. The effects of a highlight also tend to be very local. There are a range of models, from simple to complex, for surface reflection.

9.3.3 Metals

Metals exhibit only surface reflection. A rough surface spreads out the surface reflection, while a smooth surface reflects the illumination sharply. The challenge in accurately modeling metals is that shiny metals reflect the environment around them more clearly than a plastic or painted surface. For inhomogeneous dielectrics, if the surface reflection is calculated only for the light source, the effect is usually sufficient for most graphics applications. For metals, however, realistic rendering requires making use of the entire scene while calculating the surface reflection effects. As with most of graphics, there are heuristic ways of simulating mirror-like reflection that are fast and make use of the standard rendering pipeline.

9.3.4 Models of reflection

Ambient reflection

The effect of ambient reflection is the element-wise product of the ambient illuminant color C_{L_a} and the surface's body color C_b. As with all of the shading calculations, (115) is executed for each color channel.

$$I_a = C_{L_a} C_b \tag{115}$$

Body reflection

The most common model for body reflection is to use the Lambertian equation, which relates the outgoing energy to the color of the illuminant C_{L_d}, the body color of the surface C_b, and the angle between the light vector $\vec{L}$ and the surface normal $\vec{N}$.

$$I_b = C_{L_d} C_b \cos\theta = C_{L_d} C_b (\vec{L} \cdot \vec{N}) \tag{116}$$

Surface reflection

The most commonly used model for surface reflection is the Phong specular model. Surface reflection is strongest in the perfect reflection direction. The incident light gets reflected around the surface normal like a mirror, and the amount of the outgoing light seen by the viewer is dependent upon how close they are to the reflection direction. The perfect reflection direction vector $\vec{R}$ is given by (117).

$$\vec{R} = (2\vec{N} \cdot \vec{L})\vec{N} - \vec{L} \tag{117}$$

Phong surface reflection models the amount of surface reflection seen by the viewer as proportional to the cosine of the angle between the reflection direction $\vec{R}$ and the view vector $\vec{V}$. Putting a power term n on the cosine enables the modeler to control the sharpness of the highlight.

An alternative method of measuring the energy reflected at the viewer is to calculate the halfway vector between the viewer and light source and compare it to the surface normal. If the surface normal is equal to the halfway vector, then the viewer is at the perfect reflection direction. The halfway vector appears like it should be more complex to calculate, but the calculation is often approximated by taking the average of the light and view vectors, which is faster.

$$H = \frac{L + V}{||L + V||} \approx \frac{L + V}{2} \tag{118}$$

The complete equation for the Phong model of surface reflection is given by (119)

$$I_s = C_{L_d} C_s \cos^n \phi = C_{L_d} C_s (V \cdot R)^n \approx C_{L_d} C_s (H \cdot N)^n \tag{119}$$

Integrated light equation

The complete lighting equation for a single light source is the sum of the ambient, body, and surface reflection terms.

$$I = C_{L_a} C_b + C_{L_d} \left[C_d (L_i \cdot N) + C_s (H_i \cdot N)^n \right] \tag{120}$$

When there are multiple light sources, the body and surface reflection terms are the sum of the contributions from the different light sources.

$$I = C_{L_a} C_b + \sum_{i=1}^{N_L} C_{L_{di}} \left[C_d (L_i \cdot N) + C_s (H_i \cdot N)^n \right] \tag{121}$$

In the real world, the energy from an illuminant falls off as the square of the distance from the light source. In graphics-land, however, using a square fall-off rule tends to result in scenes that are too dark. Instead, we can use a falloff model that incorporates a constant and linear term. By tuning the parameters (a_0, a_1, a_2) we can obtain the desired effect. For a dungeon, setting $a_0 > 0$ and $a_2 > 0$ will create a dim, claustrophobic effect where the lighting has a very limited effect. For a big outdoor scene, setting $a_1 > 0$ but leaving a_0 and a_2 close to zero will result in a brighter world.

$$f(d) = min \left(1, \frac{1}{a_0 + a_1 d + a_2 d^2} \right) \tag{122}$$

Combining the falloff equation with the lighting equation we get the complete shading equation for a constant ambient model, Lambertian body reflection, and Phong surface reflection.

$$I = C_{L_a} C_b + \sum_{i=1}^{N_L} f(d_i) C_{L_{di}} \left[C_d (L_i \cdot N) + C_s (H_i \cdot N)^n \right] \tag{123}$$

9.3.5 Information Requirements

Given our pipeline and models, what data do we need for a lighting calculation at polygon vertices?

- Surface point in 3D after LTM and GTM transformations (world coordinates)
- Light source location(s) (world coordinates)
- Light source color(s) C_L
- Viewer location (COP in world coordinates)
- Polygon material properties (defined by model)
  - Body reflection color C_b
  - Surface reflection color C_s
  - Surface reflection coefficient n

9.3.6 Surface normals

All of these calculations depend upon surface normals. How do we get them?

- For each polygon, use the plane equation
- For each vertex, average the surface normals
- Specify them manually as part of the model
- Specify them automatically when the model is built (e.g. subdivision surfaces)

Surface normals are vectors and shouldn't be translated. Set the homogeneous coordinate to zero.

9.4 Shading

Shading is the task of determining what color to draw each pixel. In some cases, it is only necessary to determine the color of a polygon. More realistic shading, however, usually requires determining the color at the vertices or at each pixel individually. Speed and final image quality are competing factors in shading. The fewer color calculations, the faster the system will be. The more color calculations the system requires to achieve a particular effect, the slower it will be.

9.4.1 Flat shading

Make the calculation for a polygon using the plane normal. The polygon's color is constant across it's surface and no further processing is required in the scanline fill algorithm. Flat shading is useful for setting up scenes and doing fast rendering.

9.4.2 Gouraud shading

Calculate color at the vertices. Interpolate colors across the surface as $\left(\frac{R}{z}, \frac{G}{z}, \frac{B}{z}, \frac{1}{z}\right)$.

Gouraud shading is used OpenGL, and by most games. It is fast, because only colors get interpolated across the polygon and the color calculations are required only at the vertices.

Gouraud shading is subject to Mach banding–changing derivatives in color. Gouraud shading is also subject to aliasing for effects that have high frequencies such as surface reflection or shadows.

9.4.3 Phong shading

Calculate 3D coordinates and surface normals at the vertices. Interpolate $\left(\frac{x}{z}, \frac{y}{z}, \frac{1}{z}\right)$ and $\left(\frac{N_x}{z}, \frac{N_y}{z}, \frac{N_z}{z}, \frac{1}{z}\right)$ across the surface. Calculate the color at each pixel.

Phone shading generally eliminates Mach banding and aliasing effects since calculations are executed at each pixel.

Variations:

- Use directional lighting and parallel projection, which eliminates the need to interpolate surface position
- Calculate every N pixels and interpolate
- Calculate every N pixels and interpolate, but go back and redo the middle ones if the calculations are very different

9.4.4 Interpolating in perspective

Represent whatever you want to interpolate as a homogeneous coordinate.

$$\left[\begin{array}{cccc} R & G & B & 1 \end{array}\right] = \left[\begin{array}{cccc} \frac{R}{z} & \frac{G}{z} & \frac{B}{z} & \frac{1}{z} \end{array}\right] \tag{124}$$

The non-normalized version interpolates linearly in screen space under perspective projection. The normalized version does not. You can use the same technique to interpolate surface normals and (x, y, z) coordinates.

9.4.5 When to do a shading calculation?

One important question that arises is when do we need to do a shading calculation? A light source should contribute to shading on only one side of a polygon, unless the polygon is semi-transparent. The side facing away from the light source should not be lit by it. Likewise, if the polygon forms part of a closed shape, then there is no need to do a shading calculation for the interior side of the polygon for any light source outside of the shape.

Polygons that form part of a closed shape can be considered to be one-sided: only one side is relevant to shading. If the shape is constructed so that all normals point away from the visible side, then we can use the sign of the dot product of $L \cdot N$ to determine whether a shading calculation is necessary. $L \cdot N$ will be positive only if the light source is facing the visible side of the polygon.

For polygons where both sides are potentially visible, the surface normal alone does not indicate whether a shading calculation is necessary. Instead, we need to check if the viewer and the light source are on the same side of the surface. We can do this by checking if $V \cdot N$ and $L \cdot N$ have the same sign. If they do have the same sign, then the light's contribution to shading is visible and the normal should be negated so that $L \cdot N$ is positive. A reasonable structure for the shading calculation is as follows.

```
Within the Point Lighting case (and Spot, and Direct)
  Calculate L
  Normalize L

  Calculate theta = L*N
  if the polygon is one-sided and L*N is negative
    continue

  Calculate sigma = V*N
  if (theta < 0 and  sigma > 0) or (theta > 0 and sigma < 0)
    continue

  Calculate H
  Calculate beta = H*N
  if L*N < 0
    negate theta
    negate beta

  calculate shading
```

9.4.6 Physically Realistic Shading Models

...

10 Shadows

Shadows are challenging to add to a scene when using z-buffer rendering techniques. The fundamental piece of knowledge required to insert shadows is whether a light source is visible on a particular 3D point in the scene.

One approach to obtaining this knowledge is to calculate visibility from the point of view of the light source and insert the visibility information into the standard polygon rendering process. The other major approach is to calculate visibility from the point of view of each 3D point in the scene that needs to be drawn. The second approach is actually simpler to code, but can be costly in time. The first approach requires significant pre-processing, but the rendering process is fast.

There are a number of different variations on these two approaches. Two common approaches are shadow volumes–which add elements to a scene based on the point of view of each light source, and ray casting, which determines visibility of each light source from a 3D point when the point is rendered into the scene.

10.1 Shadow Volumes

The basic idea of shadow volumes is to delineate the volume where a light source is blocked by a polygon. During rendering, if a surface point is inside a shadow volume, that light source is blocked from view.

For a triangle, the shadow volume for a light source is defined by three polygons formed by the rays from the light source through the triangle vertices. The shadow polygons are generally made large enough to extend to the edges of a cube encompassing the viewer and the entire scene. The shadow polygons are then incorporated into the rendering pass along with the surface polygons. Each shadow polygon needs to keep a reference to its light source.

Shadow volumes work best when using simultaneous rendering of all the polygons in the scene, with the shadow polygons included in the process.

- At each pixel, there will be a set of real polygons and a set of shadow polygons.
- Find the nearest real polygon to identify the surface that should be rendered
- Given the set of shadow polygons in front of the nearest real polygon
  - Count how many shadow polygons there are for each light source
  - Any light source with an even number of shadow polygons is visible
  - Any light source with an odd number of shadow polygons is invisible

An alternative process is to execute a multi-pass process.

- Render the real polygons into the z-buffer to identify which surfaces are visible
- Render the shadow polygons into a light buffer to identify which lights are visible at each pixel
- Make a final pass to calculate the shading given the light source visibility

If the light source changes, the shadow polygons need to be recreated. However, the shadow polygons are invariant to viewpoint changes. Only objects that move need to have their shadow polygons updated for static lighting conditions.

10.2 Ray Casting

Ray casting is conceptually simple, but can be time consuming. The idea is to cast a ray from each surface point to be drawn towards each light source. If the ray intersects any other object before reaching the light source, then the light is not visible on that surface point.

Ray casting requires that all of the polygons in the scene be in a single data structure (e.g. array or list). To avoid executing ray casting on surfaces that are not visible, the system will make an initial pass through the polygons to determine visibility, storing the polygon, surface normal, and 3D location. In the first pass, the system also stores every object in the scene in CVV coordinates.

The second pass through the system creates a ray from the 3D location of each point (in CVV coordinates) towards each light source. Each ray is tested against every polygon in the scene to determine light source visibility. Once visibility has been determined, the system can make the final shading calculations.

Ray casting can support transparent surfaces that partially block the light, but not refractive surfaces (no caustics).

10.2.1 Area light sources

Unlike shadow volumes, ray casting supports area light sources that cause shadows with a penumbra, or soft shadows. For point light sources, a single ray cast towards the point is sufficient to test visibility. Area light sources, however, require multiple rays cast in slightly different directions. The concept is to throw enough rays that the area of the light source from the point of view of the surface point is well sampled.

The percentage of the rays that reach the light source determine the percentage of the area source that is visible from the surface point. That percentage is a simple multiplier on the brightness of the light source. As less of the source becomes visible, the shadow gets darker.

10.2.2 Sampling

Properly sampling the light source is critical to avoiding aliasing errors. Consider, for example, a square light source sampled from a surface point, as shown in figure 16. While regular sampling correctly handles large polygons casting shadows, it can miss small polygons that fit between the sampling rays. If the small polygon is in motion, then its effect on the surface point will flicker in and out.

It turns out that regular sampling is pretty much the worst method of sampling a signal. The Nyquist criterion ($f_{\text{sampling}} > 2f_{\text{signal}}$) tells us exactly when aliasing will occur with regular sampling, and the only solution is to increase the sampling rate. For shadow casting, achieving an appropriate spatial sampling resolution can be prohibitively expensive computationally.

Inserting randomness into the sampling process reduces the sensitivity of the system to aliasing. Using the same number of samples as a regular sampling methodology, the randomness of the process spreads out the sampling error, replacing aliasing errors, which tends to be coherent, with random errors. Random errors tend to be less noticeable and our visual system already has mechanisms for filtering noise out of signals.

Random sampling with N samples just randomly selects which samples to collect. In the context of light sampling, we would implement random sampling by picking N random locations on the light source and shooting rays towards those points. Random sampling is fast and provides good estimates of the percentage

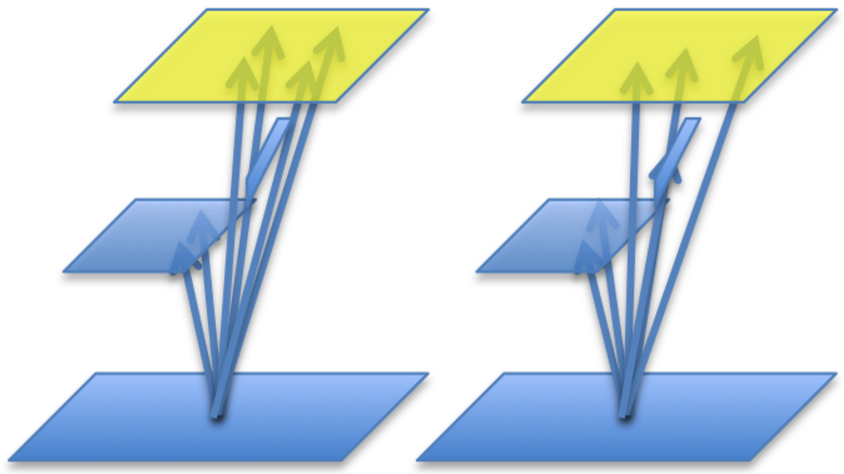


Figure 16: Figure showing regular sampling of an area source (left) and jitter sampling (right). The randomness of the sampling pattern means even small polygons will have an appropriate effect on the results on average.

of the light source on average over many collections of samples. Any single sample, however, can be significantly skewed in one direction. For example, it is equally possible for all of the samples to be clustered together as it is for them to be evenly distributed.

The theoretically optimal way to sample an area is to use **Poisson sampling**. Poisson sampling distributes samples randomly over the range, but ensures good coverage of the range by guaranteeing that none of the samples are too close. Given a minimum distance ϵ, the algorithm for Poisson sampling is as follows.

1. Initialize the list of sampling locations L to the empty set.
2. While the size of L is less than N
   (a) Pick a random location in the range p
   (b) For each element $q \in L$, if the distance between p and q is less than ϵ, return to step 2a
   (c) Add q to L

While Poisson sampling produces nice results, the cost of generating the set of samples can be high.

A compromise between random sampling and Poisson sampling is **jitter sampling**. Jitter sampling uses a regular grid, but then adds a random offset to each point on the grid. With appropriate selection of the random offsets, it is possible for two adjacent samples to be arbitrarily close. However, the underlying regular grid pattern enforces a good distribution of samples over the range. Jitter sampling is probably the most common sampling methodology in graphics because it comes close to matching the quality of Poisson sampling with the speed of random sampling.

11 Texture

Real surfaces are complex.

- Real surfaces have color variation, dirt, or color patterns on them
- Real surfaces are not perfectly smooth, but usually have a fine texture
- Real surfaces tend to reflect parts of their environment

It is possible to model surfaces at a fine level of detail. We could model a brick wall, for example, by modeling each bump and crevice of each brick and then building up the wall brick by brick. Using that level of detail guarantees that the wall will look good no matter how close or how far away the viewer might be. It also guarantees that the shading calculations will be correct, taking into account all of the various factors modeled in fine detail. Unfortunately, such a solution will not, in general, render in real time.

Alternatively, we could take a picture of a brick wall, use two triangles to model a rectangular wall and map the picture onto the polygons. Mapping a picture onto one or more polygons is a process called **texture mapping**. The result will look something like a brick wall. Under certain conditions, it may not be possible to tell the difference between the fine level of detail model and the texture mapped wall. The benefit of the texture mapped wall is that it can be rendered in real time. However, it should be clear that there are a number of issues that come up when working with texture maps.

- How do we implement the mapping from an image onto the face of a polygon?
- The fine level of detail allows us to model surface normal orientation and colors separately, but a simple texture map is just putting color variation on a flat surface. How can we add back in the surface normal variation?
- The fine level of detail models the color changes at the resolution of the model, but a simple texture map is limited by the pixel resolution of the texture, which is usually different than the resolution of the model. How do we handle different levels of detail with textures?

In general, texture mapping is the process of taking a parametric representation of the surface in (u, v) and transforming it to texture coordinates (s, t), which generally represent a rectangular coordinate system on an image. You can think of (s, t) as being pixel coordinates, although they are not generally integers.

$$\begin{aligned} u &= As + B \\ v &= Ct + D \end{aligned} \tag{125}$$

If you have two points in (u, v) and their corresponding locations in (s, t), then you can solve for (A, B, C, D), or the inverse transformation from (u, v) to (s, t).

$$\begin{aligned} s &= \frac{u - B}{A} \\ t &= \frac{v - D}{C} \end{aligned} \tag{126}$$

If you consider, for example, a Bezier surface parameterized by (u, v), then specifying the texture coordinates at two corners of the patch determines the complete texture map from the image onto the surface.

11.1 Z-buffer Texture Mapping

When working with a small number of polygons, or shapes defined by procedural algorithms (e.g. cube, sphere, or Bezier surface), it is also possible to specify the texture coordinates directly. Using this approach, each vertex of the polygon requires three pieces of information.

- Location (x, y, z)
- Surface normal (N_x, N_y, N_z)
- Texture coordinate (s, t)

The scanline fill algorithm then needs to interpolate texture coordinates, in addition to the other information. Each edge structure must be updated to incorporate the texture coordinate and change per scanline.

- sIntersect - s-coordinate along the edge
- tIntersect - t-coordinate along the edge
- dsPerScan - change in s-coordinate per scanline
- dtPerScan - change in t-coordinate per scanline

In the procedure to fill a scanline, the algorithm needs to interpolate the texture coordinates across the scanline.

- currentS - s-coordinate at the pixel
- currentT - t-coordinate at the pixel
- dsPerCol - change in s-coordinate per column
- dtPerCol - change in t-coordinate per column

Note that the currentT and currentS values specify the texture coordinates of the lower left corner of the pixel. It is possible to use them as a point sample into the texture, but that method quickly leads to aliasing as surfaces get further away from the viewer. The appropriate procedure is to calculate the texture coordinates of each corner of the pixel to identify the area of the texture map that corresponds to the pixel.

$$A_{(s,t)} = \begin{bmatrix} \left(s + \frac{ds}{dy}, t + \frac{dt}{dy}\right) & \left(s + \frac{ds}{dx} + \frac{ds}{dy}, t + \frac{dt}{dx} + \frac{dt}{dy}\right) \\ (s, t) & \left(s + \frac{ds}{dx}, t + \frac{dt}{dx}\right) \end{bmatrix} \tag{127}$$

The dsPerCol and dyPerCol fields provide $\frac{ds}{dx}$ and $\frac{dt}{dx}$. The vertical derivates, however, must be derived from the dsPerScan, dtPerScan, and dxPerScan values, since $\frac{ds}{dy}$ and $\frac{dt}{dy}$ can change for each scanline. Subtracting the expression for the texture coordinates at scanline $i+1$ from the texture coordinates at scanline i produces an expression for the vertical derivative for the scanline.

$$\begin{aligned} s_i &= s_0 + \Delta x * \text{dsPerCol} \\ s_{i+1} &= (s_0 + \text{dsPerScan}) + (\Delta x - \text{dxPerCol}) * \text{dsPerCol} \\ s_{i+1} - s_i &= (s_0 - s_0) + \text{dsPerScan} + (\Delta x * \text{dsPerCol} - \Delta x * \text{dsPerCol}) - \text{dxPerCol} * \text{dsPerCol} \\ \frac{ds}{dy} &= \text{dsPerScan} - (\text{dxPerScan} * \text{dsPerCol}) \end{aligned} \tag{128}$$

The final expressions for $\frac{ds}{dy}$ and $\frac{dt}{dy}$ are given in 129.

$$\begin{aligned}\frac{ds}{dy} &= \text{dsPerScan} - (\text{dxPerScan} * \text{dsPerCol}) \\ \frac{dt}{dy} &= \text{dtPerScan} - (\text{dxPerScan} * \text{dtPerCol})\end{aligned} \tag{129}$$

Once you know the bounding area of the texture map, you need to average the values within the quadrilateral. If the area is contained within a single pixel of the texture map, use the value of the pixel. In general, however, the quadrilateral will cross multiple pixels. There are several approaches to calculating the average value within the area.

- Average the pixels within the quadrilateral (correct, but costly)
- Average the pixels within the bounding box of the quadrilateral (almost correct, slightly faster and much easier)
- Jitter sample within the quadrilateral (pretty good, faster)
- Jitter sample within the bounding box of the quadrilateral (pretty good, and faster and easier)

Don't forget that perspective induces distortions when interpolating values across an image. In order to properly interpolate texture coordinates, we need to interpolate the vector $\left[\begin{array}{ccc}\frac{s}{z} & \frac{t}{z} & \frac{1}{z}\end{array}\right]$. At each pixel, use the homogeneous coordinate to calculate the correct s and t values. Note that this also affects the ds/dtPerScan and ds/dtPerCol values, which also must be normalized.

11.2 Mipmapping: Real Fast LOD Texture Mapping

The key to good texture mapping is using appropriate sampling techniques to avoid aliasing. When the texture under a pixel covers multiple texture map pixels, we need to compute the average of those pixels. Computing averages of arbitrary quadrilaterals on the fly is expensive, however.

The key to making good texture mapping fast is to precompute all the averages the system will need during the rendering process. Precomputing the averages of all possible quadrilaterals of all sizes, however, is unworkable. Instead, real systems use a process called **mipmapping** that makes several approximations in order to balance the need for speed with the need to avoid aliasing.

Approximation 1: use squares instead of arbitrary quadrilaterals to reduce the number of precomputed averages

Approximation 2: use squares that are powers of 2 to handle different scales

Given the above approximations, it is possible to compactly store all of the precomputed averages required by the texture map process. Figure 17 shows the storage pattern. The full resolution $2^n \times 2^n$ texture image is stored with the R, G, B color channels separated as shown. Each pixel of the next level, which is $2^{n-1} \times 2^{n-1}$, is the average of four pixels in the original resolution image. The next level, $2^{n-2} \times 2^{n-2}$, is the average of four pixels in the level above it, and so on, until the final level is 1×1 and is the average of all the pixels in the full resolution image.

It is useful to label the levels of resolution from 0 at full resolution to n at the smallest resolution (1×1). The number of pixels in the full resolution texture map represented by a pixel at level q is $p = 2^q$. All of the

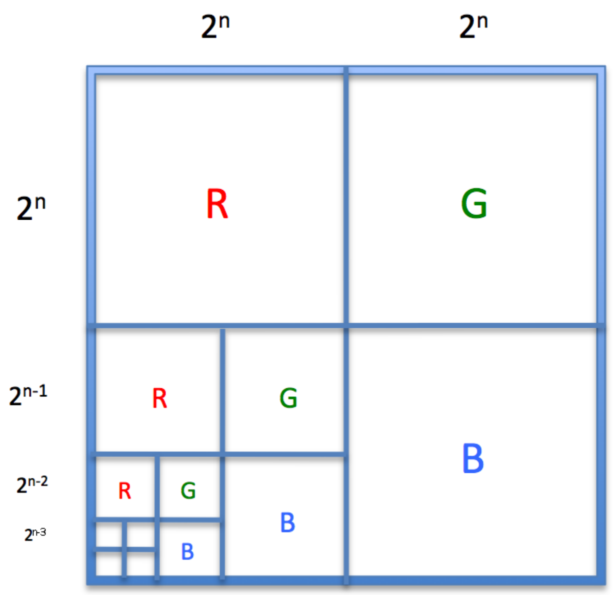


Figure 17: Storing precomputed texture averages for mipmapping.

levels should be precomputed before rendering, and the storage requirement is only 1.25 times that required for the original full resolution texture image. Games will commonly use 512×512 or 1024×1024 images with 9 or 10 levels, respectively. It is not uncommon for special effects studios to use texture images that are $4k \times 4k$, with 12 levels of detail.

The process for using the mipmap is as follows.

1. Compute the texture coordinates of the corners of the pixel
2. Compute the maximum dimension of the quadrilateral $d = \max(ds, dt)$
3. Compute the texture map level to use $d' = \log_2(d)$
4. The floor of d' and the ceiling of d' are the levels above and below the optimal square size
5. Grab the two pixels corresponding to the texture coordinates and compute their weighted average based on d'.
6. Use the weighted average as the color of the texture for the pixel.

Because all of the averages are precomputed, the process is fast and can be implemented in real time. Graphics cards provide native support for mipmapping and can hold large numbers of textures in memory simultaneously. A polygon may use multiple texture maps to achieve different effects.

11.3 Bump Mapping

Real surfaces often have fine texture that is not just color change. In many cases, the visual texture of a surface is caused by small changes in the surface height, and therefore its surface normals. The texture we see, therefore, is caused by shading variation rather than material variation and should react appropriately when the light source or object moves.

Bump mapping is the concept of perturbing surface heights and surface normals to achieve realistic textures. The actual vertex or surface points are not actually moved, but the virtual motion of the surface is the basis for calculating the perturbed surface normals. The basic idea is to use a surface perturbation function–such as waves or random motion–to calculate what the new surface normals should be.

Given: a parametric surface defined over (u, v), where the tangent vectors of the surface at (u, v) are (Q_u, Q_v).

The surface normal $\vec{n}$ at a point (u, v) is defined as the cross product of the tangent vectors.

$$\vec{n}(u, v) = Q_u \times Q_v \tag{130}$$

The assumption with bump mapping is that we are perturbing the surface point $Q(u, v)$ by an amount $P(u, v)$ in the direction of the surface normal. We assume that the size of the perturbation is small compared to the length of $\vec{n}$.

$$Q'(u, v) = Q(u, v) + P(u, v)\frac{\vec{n}}{||\vec{n}||} \tag{131}$$

The new tangent vectors at (u, v) are given by (132).

$$\begin{aligned} Q'_u &= Q_u + P_u\frac{\vec{n}}{||\vec{n}||} + P\frac{\vec{n_u}}{||\vec{n}||} \\ Q'_v &= Q_v + P_v\frac{\vec{n}}{||\vec{n}||} + P\frac{\vec{n_v}}{||\vec{n}||} \end{aligned} \tag{132}$$

Note that the last terms of (132) are very small and can be ignored, as the partial derivatives of the surface normal tend to be small. If we substitute the simplified perturbed tangent vectors back into the surface normal equation, we get the following new surface normal $\vec{n}'(u, v)$.

$$\vec{n}'(u, v) = Q_u \times Q_v + \frac{P_u(\vec{n} \times Q_{v)}}{||\vec{n}||} + \frac{P_v(\vec{n} \times Q_{u)}}{||\vec{n}||} + \frac{P_u P_v(\vec{n} \times \vec{n})}{||\vec{n}||} \tag{133}$$

The last term of (133) is zero, so the new surface normal is a function of three terms. The first term $Q_u \times Q_v$ is the original surface normal at the point. The second and third terms are functions of the partial derivatives of the perturbation function $P(u, v)$. Therefore, the information required for bump mapping is the partial derivative of the perturbation function. The partial derivatives are generally simple to calculate for algebraic functions. It is also possible to use a texture map as the basis for a perturbation function. In that case, it is necessary either to calculate the partial derivatives numerically while traversing the polygon, or to precalculate the partial derivatives are store them directly in the texture map.

11.4 2-Stage Texture Mapping

One of the biggest challenges in texture mapping is generating a mapping from a 2D texture image to a 3D object. For a single polygon, the mapping is straightforward. Mapping from a plane to a sphere, however, is not as simple.

2-stage texture mapping inserts an intermediate surface into the process. The idea is to pick a surface that is topologically similar to the final surface, but which has a simpler mapping to the plane. For example, a cube is topologically similar to a sphere, and there is a simple mapping from a plane to a cube.

Mapping the intermediate surface (e.g. the cube) to the final surface (e.g. the sphere) involves more choices. There are a number of different methods for calculating the mapping, each of which has different properties. We can define the mappings in terms of ray casting from one surface to the other.

1. Cast a ray out from the surface normal of the final surface to intersect with the intermediate surface
   - Works well for convex final surfaces
   - Concave surfaces can end up with strange texture maps
2. Cast a ray from the intermediate surface onto the final surface
   - Think of it as a shrink-wrap process
   - Works well for many surfaces
   - Have to decide what the domain for each intermediate surface section will be
3. Cast a ray from the center of the final surface out through the surface point
   - Similar to casting from the surface normal for convex surfaces
   - Gives somewhat better, and more predictable results for concave surfaces
4. Cast a ray from the reflected view direction off the final surface towards the intermediate surface
   - Creates the effect of the texture being reflected on the surface
   - If the viewpoint changes, the texture changes

11.4.1 Environment Mapping

Environment mapping makes use of the method of 2-stage texture mapping that reflects the view direction to calculating the final mapping. The purpose of environment mapping is to make a surface appear to reflect its surroundings.

1. Place the object in the scene
2. Construct the intermediate surface (cube) around the object
3. Placing the viewer at the center of the cube, generate views of the scene in all six directions with the sides of the cube as the view window
4. Use the six generated views as the texture map images
5. Use the reflection method of 2-stage texture mapping to map the images of the scene onto the final object surface

So long as the object and scene are static, the viewer can move around the scene and the object will appear to be reflecting the environment. If the object, or any part of the scene is moving, then the environment map must be recalculated for each frame. Given the speed of the rendering pipeline, however, and the low resolution generally required for the texture maps, regenerating the environment maps can still be implemented in real time using a graphics card.

11.5 Solid Texture Mapping

Textures such as marble and wood are much more difficult to map onto 3D objects, because we expect those surfaces to be seamless, and to exhibit coherence. A bit of grain or a seam in the marble may disappear and reappear due to the 3D nature of the texture. Properly modeling this kind of texture requires creating a 3D texture.

The basic idea is to generate a 3D volume of texture and then conceptually carve out the final shape.

- Generate a 3D texture and store it as a 3D grid of data (voxels)
- Place the grid so it completely surrounds the final shape
- To render a pixel on the shape, determine which voxels the surface patch corresponding to the pixel intersects and average their color values.

By modifying the relative location of the 3D texture and the surface, you can achieve different effects. Locking the texture block to the surface fixes the texture's appearance. Solid texture modeling works well even for extremely complex and detailed surfaces because the entire 3D texture volume is defined. If the texture is definable as a function, it is not necessary to use a voxel representation, which can avoid aliasing issues.

12 Animation

Animation began as a series of hand-drawn pictures shown in quick sequence. As the art moved into production houses, master animators would generate keyframes, which apprentice animators would interpolate. To simplify the drawing and enable individual animators to focus on a single actor, each actor (independent entity in the scene) would be drawn in a different transparent cel. The background would be drawn on a separate cel. The final scene would be shot by placing the cels in frames in depth order and taking a picture.

Overall, the animation process generally progressed along the following sequence.

- Storyboard shows sketches of key actors and their relative locations
- Master animators generate images of the key actors and their expressions at important points in the sequence
- Journeyman animators generate the in-between images of the key actors
- The different cels showing the actors and a cel for the background combine to form the final scene

Today, animators primarily use interactive scripting and animation systems. Users can specify paths or keyframes and let the computer interpolate the in-between motions.

12.1 Keyframe

Create every Nth frame, or frames where significant changes occur. The computer can then interpolate the position of the object for the in-between frames.

- Interpolate position
- Interpolate joint angles
- Interpolate control points

Sometimes, the animator specifies a motion path for points on the object, or for its centroid. Splines provide a useful method for specifying animation paths.

One of the big challenges in animation is interpolating orientation.

- Representation using Euler angles (3 rotations) has problems with singularities
- Representation using matrices (9 numbers) is complex and rotation matrices don't form a closed group
- Orientation in graphics systems is generally represented using unit quaternions
  - $q = a + bi + cj + dk$
  - i, j, k are orthonormal unit vectors
  - Unit quaternions live on the surface of a sphere
  - Each unit quaternion represents an orientation
  - Interpolating across the surface of the sphere lets us interpolate between orientations
  - Motion tends to be smooth and there are no singularities

12.2 Procedural

Write a script that defines the animation. This might be in a standard computer language or in a motion-specific language. Physically realistic behavior is normally procedural. In computer game engines, the scene designer usually can specify whether gravity acts on an object, whether that object interacts (collides) with other objects, the coefficients of friction of objects in the scene, and the mass of different objects. At each time step the game engine updates the position of each object based on its velocity and any forces acting on the object over the most recent time step.

Collision is one of the most difficult aspects of animation, because it is subject to time aliasing. Sampling the position of two objects at regular intervals, testing for collision, will have two problems.

- At the time the collision test returns a positive result, the two objects may already be intersecting. The actual collision will almost always occur between sample points.
- If the objects are moving fast enough, the collision test will never return a positive result. This is a case of the frequency of the objects' motion being too high for the sampling rate to capture.

Most game engines use a path-based method of computing potential collisions to determine when two objects will collide. The basic idea is to compute the path of each moving object over the next time step and intersect it with the path of all other objects in the scene. Approximating the path over a short time as a line is geometrically reasonable, and computing closest distance between two line segmentations is computationally reasonable. If the paths of two objects at their point of closest approach are far enough apart, no further computation is required. If the paths are close enough, then the game engine uses a more expensive, iterative approach to calculate the instant of collision. Once the time of the collision and the position of the two objects at that time are known, the engine can compute the resulting effects.

Detecting the exact time and manner of collisions between complex objects can also be a computationally expensive proposition. Most game engines use a simple bounding shape around objects to represent their collision boundary. Bounding shapes can be spheres, cylinders, boxes, cylinders with half-spheres on either end, or arbitrary convex polygons. Representing concave objects for collisions requires using multiple convex bounding surfaces. 3D engines almost always use regular shapes for bounding surfaces, while 2D engines may use arbitrary convex polygons.

12.3 Representational

The internal representation of the surface changes. The most common examples of representational animation are moving cloth, ropes, trees, or other objects that can change their internal shape, usually based on physical models. The trees and vegetation in A Bugs Life, for example, displayed physically realistic motion in response to a variable wind vector. The shorts at the end of the movie show nice examples.

12.4 Behavioral

- Create a set of actors
- Give each actor a state
- Give each actor a rule set
- Update each actor's state using the rule set

Rule sets in behavioral animation can be complex or simple. They generally take into account nearby actors and the environment to implement things such as obstacle avoidance or convergence to a location.

Examples

- Lion King stampede sequence
- LOTR battle scenes

12.5 Morphing

12.6 Stochastic

Particle systems

- Emitter surface
- Particle set
  - Position
  - Velocity
  - Age
  - Other attributes
- Update rule
  - Update position based on velocity
  - Update velocity based on a rule set
  - Update age
  - Update other attributes
  - Cull particles based on age or other attributes
- Rendering rule

http://www.rogue-development.com/pulseParticles.html

Examples of particle systems

- Star Trek II: The Wrath of Khan
- Lawnmower Man
- Mission to Mars

13 Global Illumination Models

Global illumination models address issues of light reflection and shadows in a comprehensive, uniform manner.

13.1 Backwards Ray Tracing

Backwards ray tracing sends rays out from the eye into the scene. By identifying which surfaces the rays hit and tracking them through multiple reflections or refractions, ray tracing models reflective surfaces and transparent surfaces. Using a ray casting approach at each reflection or refraction location, ray tracing can also correctly model shadows.

The general algorithm for each pixel is the following.

1. Calculate the ray $v_{i,j}$ from the eye through pixel (i, j).
2. Color the pixel with the result of calling RayIntersect($v_{i,j}$, PolygonDatabase, 1)

function RayIntersect(vector, database, β) returns Color

1. if $\beta <$ Cutoff return black
2. Intersect the ray with the polygons in the scene, identifying the closest intersection.
3. If there is no intersection, return the background color.
4. Calculate the surface normal at the intersection point on the closest intersecting polygon.
5. Set the return color C to black.
6. For each light source L_i
   (a) Send a ray (or many, if an area source) towards L_i
   (b) Intersect the ray with each polygon in the scene
   (c) If the ray intersects an opaque polygon, L_i is blocked.
   (d) If the ray intersects only transparent polygons, L_i is partially blocked.
   (e) Add the contribution of L_i to C.
7. Calculate the perfect reflection direction v_r
8. Calculate the magnitude of the surface reflection α
9. return C + α * RayIntersect(v_r, PolygonDatabase, $\alpha \times \beta$)

Because ray tracing shoots rays only from the eye, or from reflections towards a light source, in its standard form it does not model diffuse interreflection between objects or caustics caused by the lensing effects of transparent objects.

A computationally costly modification to ray tracing is to send multiple rays out from each reflection intersection, using some to modify the diffuse reflection and some to modify the surface reflection.

13.2 Forwards Ray Tracing

The most common method of rendering a scene is some type of rendering algorithm that takes advantage of the coherence of objects. Hardware pipelines are optimized to project polygons, usually triangles, from a 3D space into a 2D space and draw them individually onto the screen. The hardware rendering pipelines further divide the process into one of making shading calculations at vertices and then again on fragments, which are triangles or sub-areas of triangles.

The default rendering pipeline using a z-buffer for hidden surface removal does not inherently include shadows, transparency, reflection, or the interreflection of light between surfaces. Efforts to include these physically realistic effects are normally add-on modules that are designed to either fit within the hardware pipeline structure or are design elements that provide a sufficiently realistic result but do not represent an actual physical model of the effect.

Shadows, for example, can be built into a rendering pipeline by adding either shadow polygons or shadow volumes to the scene, enabling light sources to be turned on or off for a particular surface. Likewise, reflections can be modeled by generating textures that consist of scene renderings from the point of view of the object: environment mapping. Both of these achieve realistic effects but the rendering pipeline does not explicitly model the physical processes underlying the phenomena.

Ray tracing was the first rendering methodology developed to integrate hidden surface removal, shadows, reflection, and transparency in a single algorithm (Whitted, 1980). The basic concept of ray tracing is to send a ray from the eye location through each pixel of the image plane out into the scene. Intersecting the ray with each object in the scene indicates which surface is visible. Given the point of intersection, the algorithm can use additional rays to test the visibility of each light source and throw more rays into the scene in the perfect reflection direction and/or through the surface in the case of transparency.

The end result of throwing a ray into the scene is a tree of rays, some representing reflections, some representing transmissions. In addition, at each intersection point the algorithm knows which light sources are available. By traversing the tree of rays and collecting their contributions in a physically correct manner, the algorithm identifies the intensity and color at the source pixel. Repeating the process for each pixel generates an image of the scene that includes hidden surface removal, shadows, reflection, and transmission.

The general algorithm for each pixel is the following.

1. Calculate the ray $v_{i,j}$ from the eye through pixel (i, j).
2. Color the pixel with the result of calling RayIntersect($v_{i,j}$, PolygonDatabase)

```
function RayIntersect( vector, database) returns Color
    1. if $\beta <$ Cutoff return Black
    2. Intersect the ray with the polygons in the scene, identifying the closest intersection.
    3. If there is no intersection, return BackgroundColor
    4. Calculate the surface normal at the intersection point on the closest intersecting polygon.
    5. Set the return color $C$ to black.
    6. For each light source $L_i$
        (a) Send a ray (or many, if an area source) towards $L_i$
        (b) Intersect the ray with each polygon in the scene
        (c) If the ray intersects an opaque polygon, $L_i$ is blocked.
        (d) If the ray intersects only transparent polygons, $L_i$ is partially blocked.
        (e) Add the contribution of shading the surface with $L_i$ to $C$.
    7. Calculate the perfect reflection/transmission direction $v_r$
    8. Calculate the magnitude of the surface reflection/transmission $\alpha$
    9. return $C + \alpha$RayIntersect$(v_r, \text{PolygonDatabase})$
```

The primary task of a ray tracer is to intersect a ray with object models. Object models may be mathematical functions, polygons, splines, or even volumetric objects representing amorphous phenomena like gasses or clouds. Most of the ray tracer's time is spent figuring out which object the ray hits first. The ray tracer spends a much smaller amount of time computing shading values, the proper direction in which to send more rays, and the overall shading value of a pixel.

Because the bulk of a ray tracer's time–and code–are spent doing ray-object intersections, there are two primary methods of speeding up a ray tracer. First, you can try to reduce the number of ray-object intersection tests. The primary method of reducing the number of tests is to organize the scene in a hierarchy of simple bounding surfaces. If a ray does not intersect one of the enclosing bounding surfaces, then it does not intersect any of the objects within it. Bounding surfaces that have fast ray-object intersection calculations include circles and boxes.

The second approach to speeding up a ray tracer is to make the ray-object intersection calculation faster. There are a number of different techniques that fall in this category.

- Use object models that enable fast ray-object computations. Implicit algebraic surfaces, for example, tend to have fast solutions.
- Organize the computations to optimize the use of computer hardware like memory caches.
- Parallelize the computations across CPUs.

13.3 Radiosity

One of the major perceptual issues with computer graphics is that it is too clean. Standard z-buffer rending systems can use more complex shading and shadow algorithms to capture surface properties, shadows, and highlights, but they do not incorporate any of the interaction–interreflection–between nearby surfaces: light bouncing off one surface and illuminating a nearby surface. Ray tracing starts to address this issue by using a comprehensive rendering algorithm that incorporates specular reflection. Distributed ray tracing that shoots additional rays according to the overall surface reflection function–which includes both body and surface reflection–addresses the issue of diffuse-diffuse reflection directly, but at tremendous computational cost.

Radiosity is a different algorithm for solving the global illumination function that explicitly models how each patch of the world is related to every other patch (Goral et. al., 1984). The fundamental idea of the radiosity algorithm is that the illumination arriving at a patch–the irradiance–and the illumination leaving a patch–the radiance–are related by the albedo of the surface, modified by any self-emittance the surface may possess. The albedo is the percent of the incoming light reflected back to the world. The radiosity of a patch is the energy per unit surface leaving the patch. In rendering, this is proportional to the visible color of the patch from the point of view of the camera.

The incoming light is represented by an integral: it is the sum of the light arriving on the patch from all directions. That light can come from emitters–light sources–or from surfaces reflecting their own incoming light. If we assume a Lambertian reflectance function, which models reflected light as leaving equally in all directions, then the amount of light leaving the surface per solid angle is a constant and independent of the viewing direction. Therefore, the amount of light leaving one patch and arriving at another is related only to the geometry of the two patches. If they are both large and close together, much of the energy leaving one patch will arrive at the other patch. On the other hand, if the patches are distant and/or small, very little energy leaving one of the patches will arrive at the other patch.

If we create all of the relationships between all of the patches in the scene, we arrive at a large number of linear equations that represent the flow of energy around the system. Given a fixed set of emitting surfaces, there is a steady state solution to the system that defines the radiosity of each patch. The radiosity algorithm for solving for the steady state values comes from the domain of heat flow.

The radiosity of a surface consists of two parts: the energy emitted by a surface and the energy reflected by the surface.

$$B_i = E_i + \rho_i H_i \tag{134}$$

- B_i : the radiosity, which is the energy per unit surface area leaving surface i
- E_i: the self-emittance of the surface, which is non-zero only for light sources
- ρ_i : the albedo of the surface, or the percent of incoming energy that is reflected
- H_i : the irradiance, or radiant energy incident on the surface

The incident radiant energy on a surface patch is the sum of the radiant intensities of all surfaces j that are visible from surface i, modified by the geometry of the relationship between the surfaces. We can assume that the radiosity of a surface does not vary across a small patch.

$$H_i = \sum_{j=1}^{N} B_j \frac{A_j F_{ji}}{A_i} \tag{135}$$

- B_j : the radiosity of patch j
- A_j : the area of patch j
- A_i : the area of patch i
- F_{ji} : the fraction of energy leaving surface j that hits surface i

Equation (135) defines the relationship between patches in the scene. The incident energy from patch j is proportional to its radiosity and area, and it is inversely proportional to the area of patch i.

The form factor between two patches is reciprocal: if you know the relationship in one direction, then you know it in the other direction.

$$A_i F_{ij} = A_j F_{ji} \tag{136}$$

By substituting the definition of H into the radiosity equation and making use of the reciprocity relationship, we can redefine the radiosity equation and then solve for the emittance, putting all of the radiosity values together as in (138).

$$\begin{aligned} B_i &= E_i + \rho_i \sum_{j=1}^{N} B_j \frac{A_j F_{ji}}{A_i} \\ B_i &= E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij} \end{aligned} \tag{137}$$

$$E_i = B_i - \rho_i \sum_{j=1}^{N} B_j F_{ij} \tag{138}$$

Writing (138) for each surface patch in the scene creates a large set of linear equations that relate each patch to each other patch.

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2N} \\ \dots & \dots & \dots & \dots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \dots & 1-\rho_N F_{NN} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \dots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \dots \\ E_N \end{bmatrix} \tag{139}$$

Note the following properties of the radiosity matrix equation.

- All of the F_{ii} form factors are zero, making all of the diagonal terms 1.
- The E_i are all zero except for light sources.
- The sum of the form factors in a row is ≤ 1

Applying the equation in all three color channels–wavelengths–gives the radiosities for each color channel, which are proportional to the apparent intensity of each surface patch in a scene. Once calculated, for a static scene the form factors are fixed, which means they can be re-used if the colors, lighting, or albedos of surfaces change.

The radiosity matrix also has the property that the Gauss-Seidel iterative solution technique is a useful, and potentially fast, method of solving this set of equations. Note that the matrix may be very large, as radiosity is only effective if each patch in the scene is small. Modeling a flat wall, for example, must be done with many small patches, not a single large polygon.

13.4 Radiosity Algorithm

The complete radiosity algorithm is as follows.

1. Discretize the environment into small patches.
   - The patch size must be small enough to catch changes in the illumination field.
   - Large patches relative to the changing illumination conditions will cause aliasing.
   - Adaptive subdivision during the radiosity process is possible to implement.
2. Calculate the form factors between all of the patches.
   - There are $\frac{N^2}{2}$ form factors.
   - The algorithm can calculate the form factors on demand (lazy evaluation).
3. Solve the radiosity matrix using Gauss-Seidel iteration.
4. Use the radiosities in a standard z-buffer rendering pipeline.
   - Render the scene using Gauraud shading
   - Can add direct surface reflection to the scene with Phong shading.
   - The color of a vertex is the average of adjoining patch radiosities.

13.5 Radiosity Form Factors

Calculating the radiosity form factors is, in general, an order of magnitude more expensive than solving the radiosity matrix. Gauss-Seidel iteration is an $O(N)$ process, in practice, but there are $O(N^2)$ form factors, given N patches in the environment.

The generic method for calculating form factors is to use their geometric relationship and some calculus to calculate the flow of energy from one patch to another. For a convex space with no interior obstacles, such an approach is reasonable. However, in a general environment with obstacles, it quickly becomes a challenge to develop a closed form solution.

One of the most important developments in making radiosity practical was the development of the hemi-cube solution to the generation of form factors (Cohen and Greenberg, 1985). A hemi-cube is the half of a cube enclosing the visible side of a surface patch. The hemi-cube represents the complete illumination environment of the target surface patch. To discover the visibility of each surface patch from the target patch we can use a z-buffer algorithm to render the scene onto the hemi-cube from the point of view of the target.

Each pixel of the rendered images on the sides of the hemi-cube represents a solid angle of space, and the polygon inside that pixel is the surface contributing to incoming light from that direction. Therefore, the form factor between the target polygon i, and a projected polygon j is proportional to the number of pixels on the hemi-cube covered by polygon j. We can pre-compute the form factor contribution of each pixel on the hemi-cube, enabling us to compute the form factor F_{ji} (and F_{ij} by reciprocity) by summing the pixels covered by polygon j.

$$\text{X-Y plane (top)}: \Delta F = \frac{1}{\pi(x^2+y^2+1)^2}\Delta A \tag{140}$$

$$\text{Y-Z plane (side)}: \Delta F = \frac{z}{\pi(y^2+z^2+1)^2}\Delta A \tag{141}$$

$$\text{X-Z plane (side)}: \Delta F = \frac{z}{\pi(x^2+z^2+1)^2}\Delta A \tag{142}$$

Note that one hemi-cube operation calculates the form factors between the target patch and all other patches. Therefore, one hemi-cube rendering generates all of the form factors for a row of the radiosity matrix.

13.6 Re-ordering the Radiosity Solution

The major drawback to radiosity is the computational cost of both the form factors and the computation. Each row of the radiosity matrix is represented by (143), which defines the value of the radiosity of patch B_i in terms of its emittance, albedo, and incident energy.

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij} \tag{143}$$

The Gauss-Seidel iteration method of solving for the radiosity values updates each row of the matrix in turn. The new estimate of the radiosity is adjusted closer to its true value using a gradient-descent style solution. Generically, Gauss-Seidel iteration uses (144) to solve the equation $Ax = b$, where $x_i^{(k+1)}$ is the new value of the unknown variable, $x_i^{(k)}$ is the current value, a_{ij} is element ij of A and b_i is element i of b.

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij}x_{(k)}\right] \tag{144}$$

Re-written for radiosity, the iteration process looks like (145), since the diagonals all have value 1. The initial values of the radiosities B_i are set to the emittance values of the patch E_i. Note that most of the initial values will be zero; only light sources will have non-zero values.

$$B_i^{(k+1)} = E_i - \sum_{j=1}^{i-1} \rho_i F_{ij} B_j^{(k+1)} - \sum_{j=i+1}^{N} \rho_i F_{ij} B_{(k)} \tag{145}$$

Conceptually, (145) says that the new radiosity value is a function of all the patches sending light to the target patch i. Since Gauss-Seidel iteration starts with most radiosity values at zero, only emitting patches will modify the radiosities on the first iteration, and then only the surfaces visible from the light sources (non-zero form factors). The concept is that each patch is gathering light to it. However, it is a lot of computational effort in the first few iterations with very little progress.

Computationally, it is possible modify the G-S process to update columns of the matrix instead of rows. Conceptually, this reverses the distribution of energy so that each patch is shooting energy to its neighbors rather than gathering. While this does not seem like an immediate win, since most patches will be shooting zero energy to start, it is also possible re-order the order in which columns get updated so that patches emitting a lot of energy send that energy to other patches first. Re-ordering the computations and using a shooting concept is called progressive radiosity (Cohen, Chen, Wallace, and Greenberg, 1988). It significantly speeds up the radiosity computation and also enables quicker previews of the final scene.

Progressive Radiosity
(Cohen, Chen, Wallace, and Greenberg, 1988)

```
set all $B_i = 0$ and $\Delta B_i = 0$ except for emitting surfaces, which are set to $E_i$
for each Gauss-Seidel iteration
    build a max-heap based on $\Delta B_i A_i$
    for each patch $i$ in the max-heap
        calculate form factors $F_{ij}$ with a hemi-cube at patch $i$
        for each patch $j$
            $\Delta Rad = \rho_j \Delta B_i F_{ij} A_i / A_j$
            $\Delta B_j = \Delta B_j + \Delta Rad$
            $B_j = B_j + \Delta Rad$
        $\Delta B_i = 0$
```

14 Non-Photorealistic Rendering

Non-photorealistic rendering [NPR] is the use of computer graphics to create imagery that is not rendered according to an accurate physical model. NPR has also expanded to incorporate manipulating existing images to appear as though they were generated in a different manner.

NPR has been used to render scenes as though they were generated using paint, watercolor, pen-and-ink, or in a technical drawing style intended to enhance the viewer's understanding of the scene.

One of the first people to publish in this area was Strassman, who describes the area as follows.

> *As techniques like ray-tracing extend the ability of computers to render scenes with photographic exactitide, there will be a complementary advancement of techniques which allow computers to suggest scenes with artistic abstraction.*
>
> S. Strassman, "Hairy Brushes", SIGGRAPH, pp 225–232, 1986.

Later work has gone on to attempt to do more than "suggest" scenes with artistic abstraction and has attempted to model artistic abstraction using extremely detailed physical models that become difficult to differentiate from the real thing.

Work on artistic abstraction in computer graphics prior to Strassman was limited to heuristic techniques that involved pixel-level representations of brushes, such as small images or circles that could change size or hue in response to pressure. Such tools are commonplace in consumer-level computer drawing programs. However, they do not have any abstraction beyond the pixel-level characteristics and do not model brush hairs, paint or other physical aspects of artistic drawing.

Strassman models several different aspects of drawing, focusing on a model of Japanese sumi-e painting.

- Bristle - a class that defines a single hair
- Brush - a 1-D array of Bristle objects
- Stroke - a trajectory of position and pressure
- Dip - a description of the initial state of a class of brushes
- Paper - a mapping onto the display device

When drawing, the brush is aligned so its array of bristles is perpendicular to the stroke trajectory. At discrete steps along the stroke the system updates the state of each bristle, which defines its color, ink quality, relative position, or other property. Bristles can move relative to one another during a stroke. They can also steal ink from neighbors, enabling a Bristle that has no more ink to then steal some ink from a neighbor. The state of the bristles defines an image that is transferred to the paper. A bristle only contributes to the image if it is in contact with the paper with sufficient pressure and has ink. Each bristle's ink is of a single color, but different bristles may have different colors.

Strassman defines a stroke as a 2D cubic spline defined by the user. Each control point specifies not only the 2D location but also the pressure at that point along the stroke. When drawing a stroke, the system defines a series of small polygons that each encompass a short segment of the path. The width of each polygon perpendicular to the stroke is defined by the pressure on the brush at that part of the trajectory. A brush with more pressure produces a wider polygon.

A Dip is a snapshot of Bristle states. In sumi-e painting, by controlling the distribution of ink colors and amounts on a brush the same trajectory and brush can product very different results. The Dip object is the way Strassman organizes the initial state of a brush and enables the use of a collection of different Dips to generate an image.

The Paper object in Strassman's implementation is a simple image that fills in pixels based on messages from the Brush object. The Paper object could also implement a method of ink spreading and other characteristics for more realistic effects.

Strassman's GUI implementation enables the user to click to create trajectories, adjust pressure, and control the Dips. The only complete example of the system is in Figure 1, although the paper contains numerous figures showing how different parameters affect the appearance of different brush strokes.