Computer Graphics: from Pixels to Scenes

Matthew O. Ward
Computer Science Department
Worcester Polytechnic Institute

Abstract

Computer graphics is the process of creating a visual presentation of an object or scene using a mapping process from the object or data space to the image space. In its most abstract form, it involves deciding what color to set each value in a two-dimensional array which is then output to a screen or printer. Traditionally, graphics is taught by starting with 2-D line drawings and proceeding to 3-D wire frame images and finally shaded surfaces. This mimics to some extent the evolution of the field, which was based predominantly on the hardware technology available.

Given the current predominance of raster-based graphics and the computational capabilities of the computers in common use, it is time for this order of presentation to be reevaluated. This manuscript approaches the teaching of graphics by starting with the generation of a pixel and builds the framework for the modeling and rendering of more and more complex 3-D objects, with minimal emphasis on two- and three-dimensional line drawing. It is hoped that this order of presentation will allow students to more quickly and effectively learn about the synthesis of pseudo-realistic 3-D images.

Contents

Chapter 1

A Digital Image is Worth a Megabyte (or so)

A **digital image** (also called a computer picture or raster) is a two-dimensional array of entities known as **pixels** (picture elements). A pixel is simply a numeric value, and with this value is associated a color or intensity. This image may be entered into the computer using some form of input device (a digital camera, scanner, or other sensing mechanism) or created synthetically using a **rendering** algorithm applied to an abstract scene representation which is typically formed by a **modeling** procedure. This latter method is the fundamental process of the field of *Computer Graphics*.

A **color map** (also called a lookup table, or LUT) is a table in the computer which associates a numeric pixel value with a color or intensity to be displayed on the output device. A color is typically specified in the computer as a mixture of varying levels of red, green, and blue, and thus a color map will usually contain three components for each possible pixel value. A typical workstation or PC color map will have 256 rows in it, and thus pixels would be numbered 0 to 255 (which can be stored in an 8-bit field). This is referred to as *pseudo-color*, since it is not capable of displaying photo-realistic images (except in black and white). *True-color* systems normally support 16 million colors, which provides for simultaneous display of all possible mixtures of 256 levels for each primary color. The 3 values specifying the contributions of the primary colors may be integers (usually in the range 0 to 255) or real numbers (often in the range 0.0 to 1.0), though this is very system-dependent. A value of 0 indicates there is no contribution from that primary color, and the highest value means full intensity for that color. Shades of grey can be generated by having equal amounts of the primaries. In most systems, users can either set an individual color index to a particular mix of the primaries, set the entire color map, or request the index of an existing color which most closely matches a specific red/green/blue combination. The RGB (red, green, blue) color space is only one of several which are found in computer graphics. Others include HSV and YIQ. We'll look at some of these later.

To generate an image on the screen, the typical steps would be as follows:

1. Initialize or open the graphics display. In **X**, this would be done with a call to *XtInitialize()*. In **OpenGL** you would use the function *glutInit()* (see the documentation and examples of X, Java, and OpenGL for details on the parameters to these functions).

2. Create an area to draw into. Normally you would specify some attributes of this area, such as the width and height. In **X** this is done with *XtCreateManagedWidget()* using a widget of type *XwworkSpaceWidgetClass*. In **OpenGL** you use *glutCreateWindow()*.

3. Allocate space to hold a raster image. This may be done statically or dynamically. Some graphics packages provide utilities to support this, such as the *XCreatePixmap()* function in **X**. For other systems, users allocate space as for a normal 2-D array.

4. Fill the raster array with values. This may be done a single pixel at a time or with block moves (e.g. the *XCopyArea()* function in **X**).

5. Set the color map. Most graphics packages contain default color maps. You can also set the colors in one or more entries of the color map. **X** allows you to request color map indices, and once these are granted (if available), you can place RGB values into it. You can also retrieve the index of the color in the existing color map which is closest to a particular RGB combination. Finally, you can take control over the entire color map. Be wary, though. Your windowing system uses the current color map, and so if you change the color associated with an index used by the window manager, that change will be reflected in anything the window manager has drawn with that index. Thus all other windows may suddenly go black if you change certain color map entries. Once you move the mouse out of the window associated with your graphics application the default color map should return. One good strategy is to leave the bottom few colors alone when controlling the entire color map. So, for instance, instead of pixel values having the range of 0 to 255, you might consider using a smaller range, such as 50 to 255. For many images this would not be a major sacrifice. For some graphics languages, such as **OpenGL**, you can either use an indexed color map or set the RGB values for each pixel.

6. Display the raster. In most windowing systems, in order to get anything to happen you must *realize* the windows and widgets associated with your graphics program, and then start up what is called the *Event Loop*. This causes widgets and such to display and allows events such as mouse clicks to be handled. In **X** you call *XtRealizeWidget()* followed by *XtMainLoop()*. In **OpenGL** you call *glutMainLoop()*.

Once you can set pixels of arbitrary color in a raster, the next skill to master is drawing lines (though most primitive graphics packages come with line-drawing commands, it is interesting and often useful to understand how it is done). Assume you want to draw a line from point (x_1, y_1) to (x_2, y_2). We need to determine which pixels between those points should be set to the appropriate color. For any arbitrary line, we can assume one of the variables (x or y) will always be incrementing or decrementing as you move between the two points. The other variable will either stay the same or change by plus or minus 1. If we assume a slope greater than 1.0, this means the y value of consecutive pixels will always be incrementing. The value of x will change proportional to the inverse slope, i.e. $x_{i+1} = x_i + 1.0/m$, where m is the slope. We either round the result or truncate it to get the integer x coordinate. The problem with this method is the need for floating point operations, which are generally much more expensive than integer operations.

A nifty and efficient algorithm was developed by Bresenham which involves only integer arithmetic. The general goal was to develop a *decision function* which indicates whether one should change (increment/decrement) the value of the variable or leave it the same based on whether the value of the function was positive or negative. As each pixel is generated, the decision function is modified and reevaluated for the next point. The "best" decision function would be to compute the distance from the center of the 2 potential continuation pixels to the real line, but this would be more computationally intensive than the previous method. So instead, Bresenham decided to use an approximation to the distance, namely either the vertical or horizontal difference between the approximated and real points. By suitable mathematical manipulation, the floating point numbers can be eliminated from the calculation of the decision function.

Assume that the slope of the line is between 0 and 45 degrees. This means that x will increment at each pixel along the line, while y may increment or stay the same. If (r, q) is the current location, the choices for continuation are $(r + 1, q)$ or $(r + 1, q + 1)$. Without loss of generality, we can translate the start of the line to the origin, giving the equation $y = mx$, where $m = \Delta y/\Delta x$. The two approximate differences, s (y is unchanged) and t (y is incremented), can be computed as follows:

$$s = y_{real} - y_{approximated} = m(r + 1) - q$$

$$t = y_{approximated} - y_{real} = (q + 1) - m(r + 1)$$

The difference of these two approximate distances can be used to form our decision function, since if s is greater than t we would increment y and otherwise leave y constant. This decision function can be simplified by noting that the x-component of the slope will be positive and can be divided out without changing the sign of the resulting decision function.

$$(s - t) = 2m(r + 1) - 2q - 1$$

$$\Delta x(s - t) = 2r\Delta y + 2\Delta y - 2q\Delta x - \Delta x \; if \; \Delta x > 0$$

Thus d_i, the decision function at the i'th point is given as:

$$d_i = 2(r\Delta y - q\Delta x) + 2\Delta y - \Delta x.$$

However, $r = x_{i-1}$ and $q = y_{i-1}$, giving

$$d_i = 2x_{i-1} * \Delta y - 2y_{i-1} * \Delta x + 2\Delta y - \Delta x.$$

We now do a bit of mathematical trickery known as *forward differencing*, which entails computing the next decision function based on the value of the previous one (a common step in graphics).

$$d_{i+1} = 2x_i * \Delta y - 2y_i * \Delta x + 2\Delta y - \Delta x$$

$$let \; x_i - x_{i-1} = 1$$

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x(y_i - y_{i-1})$$

if $d_i \geq 0$ *select* t *and* $y_i = y_{i-1} + 1$ *and* $d_{i+1} = d_i + 2(\Delta y - \Delta x)$

else select s *and* $y_i = y_{i-1}$ *and* $d_{i+1} = d_i + 2\Delta y$

initially, $d_1 = 2\Delta y - \Delta x$

Note that the initial decision function is simply the subtraction of 2 integers, and all new decision functions are formed by either adding $2\Delta y$ or $2(\Delta y - \Delta x)$ (both constants) to the previous one, based on the sign of the results. The same formulation can be used to derive equations for lines in any of the other seven octants. A very efficient formulation compared to the initial one presented!

The last of the basic raster operations we need for now is the ability to draw a filled polygon (a polygon is simply specified as a list of vertices, with the last vertex being the same as the first). One method of performing this task is known as *recursive flood or boundary filling*. The idea is that if you want to fill a polygon with a particular color index, you first draw the boundaries of the polygon (using the above-described line drawing operations), then select an interior point to set to the desired color. You then call a fill algorithm with each the immediate neighbors of this point, where the fill algorithm checks to see if the point passed has been set to the desired color yet, and if not, it is set and all of its neighbors are recursively called with the fill algorithm. Although easy to implement, the algorithm has some problems.

1. Selecting an interior point can be difficult, especially for complex shapes. One method to determine if a point is in the interior is to scan in all directions from the point, counting how many times you cross a border. An odd number of crossings means the point is likely in the interior.
2. Care must be taken to insure that the recursive fill does not "escape" the polygon. Each pixel can be viewed as having either four or eight neighbors, depending on whether diagonal adjacency is to be considered. While eight-way recursion may be performed in some instances, it depends on how the boundary lines have been constructed. The term *four-connected* implies that diagonal connectivity is not sufficient in constructing lines, and thus more pixels are set for the lines. In this case, 8-way recursion may be performed. In *eight-connected* lines, however, it is only safe to perform 4-way recursion, as it is possible to escape the polygon along a diagonal neighbor.
3. Recursion can be frightfully inefficient, and for large polygons could exhaust the stack space of your machine.

A more efficient and powerful method for performing region filling is known as the *Scan Line Fill Algorithm*. The algorithm starts with a list of edges which make up the polygon, and this list is used to compute intersection points along each scan-line (a row of the image). Each edge will be crossed 0 or 1 times for a given scan-line (horizontal edges are ignored). The intersection points are computed and sorted in x values. Then pairs of consecutive values (corresponding to entrance and exit points) are used to fill in pixels in the interior of the polygon (this filled in region along a scan-line is often called a *run*). The polygon is completely filled by starting at the row containing the vertex with the highest y-value and proceeding through the one with the lowest y-value. The

only sticky point of the algorithm is when a vertex is a local minima or maxima, in which case the x-value of the vertex is duplicated in the list of intersections. This way the run is only 1 pixel long.

Although this algorithm sounds complicated, it can be made quite efficient with appropriate cleverness. For example, the x-values for the intersection points could be computed using the normal equation for the intersection of two lines, but it would be far more efficient to simply use one of the line generation algorithms to compute the x-value for the edge as the y-value changes. Other efficiencies are also possible.

An implementation of this algorithm can be found in the software/utilities directory. It is an important one to learn, as it can be used in a variety of other phases of the graphics rendering process (smooth shading, hidden surface removal, texture mapping).

If you look at the results of the line drawing and polygon filling algorithms we've discussed thus far, you'll notice that almost all edges exhibit an annoying stair-stepping appearance. These so-called 'jaggies' are the result of a process known as *aliasing* - which often occurs when a continuous entity (such as a line) is represented in a discrete fashion (with pixels). For all edges that are not vertical, horizontal, or exactly 45 degrees, we need to decide which pixels should be drawn and which shouldn't, resulting in all lines being represented as a sequence of short lines that are either horizontal, vertical, or at 45 degrees.

A process known as *anti-aliasing* can be used to smooth out the jaggies. There are many algorithms that can be used, but most are based on the notion that edges can partially cover pixels, and that by varying the intensity of the pixel proportional to its coverage, we can blur the edge as a means of softening the stair-stepping effect. A simple method for doing this is to augment the Bresenham algorithm such that both of the two possible continuations for the line are set, with each given a proportion of the overall pixel value. Thus if the real line is twice as close to the center of one continuation pixel than the other, then the closer one would get 2/3 of the line intensity and the further would get 1/3. Another approach is called *super-sampling*, where the resolution of the image is artificially increased (e.g., instead of having a $100x100$ grid, you process it as a $200x200$ or $400x400$ grid), and the pixels for representing the line are computed as before. Then, based on the path taken by the high-resolution pixels (through the center of the original pixel or across one of its corners), the intensity of the pixel can be varied. Like many processes in graphics, there is a trade-off between quality and computational requirements for anti-aliasing.

Reading Topics: Introduction to raster graphics, color maps (lookup tables) Hill Chapters 1, 2, and 10.1 through 10.8

Project 1: Create (by whatever means you prefer) a 2-D array of pixels and display it using low-level graphics routines provided by your system or the graphics library mentioned in the syllabus. The image may simply be of random values, a predefined pattern of values (e.g. a ramp), or an existing digital image (you can use a digital camera to acquire it or grab one off of the Net). Experiment with changing the color map, either by reading in a file with a table of RGB values or via random or user-specified changes of particular color indices. An interesting affect may be obtained by "cycling" the colors, i.e. rotating the values up or down the table. One warning: as mentioned above, your windowing system uses some of the color map indices. If you change the values in these indices, you may not see the windows outside your current window unless you

move the mouse out of the window (scary at first!). [Hint: the Java and OpenGL demonstration programs provided in the *software* directory all deal with rasters and/or colormaps and would be useful for this project.]

Chapter 2

The Mathematics that Make Graphics Work

Vector analysis and manipulation form an integral part of many aspects of computer graphics. Basically, we can think of a **vector** as a displacement, which has a magnitude and a direction, but not a position. It is represented as an ordered pair (Δx Δy) in 2-D and an ordered triplet in 3-D (Δx Δy Δz), and is often confused with the representation of a point (x, y) or (x, y, z). Vectors have many properties: they can be added (just sum the corresponding components), scaled by a constant (multiply each component by the constant), and multiplied in a couple ways. A *dot product* (also called scalar or inner product) is formed by multiplying the corresponding terms of 2 vectors and summing the results. The result is a scalar value. A **cross product**, which is only valid in 3-D, is formed in a manner similar to that used for computing components of a determinant - namely you stack one vector over the other and, for each position, you cross out the column corresponding to that position, multiply the remaining diagonals, and subtract the 2 results of multiplication (flipping signs for the middle one). The result is another vector. Another computable property of a vector is its **magnitude**, which is the square root of the dot product of a vector with itself. We can create a **unit vector** by dividing each component of the vector by its magnitude. Equations for each of these functions are given at the end of this chapter.

A useful property to remember is that the dot product of two unit vectors is equal to the cosine of the angle formed by the vectors. This will be used in determining how to shade a surface. Also, the cross product of two vectors gives you a vector orthogonal (normal or perpendicular) to both vectors, which will be useful for computing the equation of a plane. A more complex relationship between vectors is the **projection** of one vector onto another. The projection of vector A onto vector B can be envisioned as the shadow of A cast on B. Thus the projection will have its direction common with B, but with a magnitude depending on the angle between A and B and the magnitude of A. The equation for the projection is

$$\frac{A \cdot B}{B \cdot B} B \tag{2.1}$$

which is obviously simply a scale factor applied to B. With the projection we can derive the angle of reflection for a vector hitting a surface, as follows. Assume vector V is hitting a surface whose normal (the vector perpendicular to the surface) is N. If P is the projection of V onto N, the

reflection vector $R = V - 2P$ (try to prove it to yourself before looking at the derivation at the end of the chapter).

We often blur the distinction between coordinates and vectors to take advantage of some of the nice properties of vectors, especially in conjunction with *transformation matrices*. In computer graphics, we often define objects in terms of vertices (which are used to form edges and surfaces), and can manipulate the location, size, and orientation of the surfaces or edges by applying transformations to the vertices. Some transformations, such as *translation*, simply add an offset to one or more of the coordinates. Others, such as *scaling*, multiply one or more coordinates by a constant. The most general transformation on a coordinate might include scaling, translation, and offsetting one coordinate proportional to the value of another coordinate (which occurs in *rotation*). To encapsulate all of these transformations into a single package, we use a vector-matrix formulation of the problem. If we have an N by 1 matrix (vector) and we multiply it by an N by N matrix (transformation), we get another N by 1 matrix. In chapter 5 we will see matrix formulations for all the normal transformations we might want to do to an object, and in Chapter 7 we will develop the transformations necessary for generating arbitrary views with perspective distortion. We can even combine transformations on objects by simply multiplying the transformation matrices together (this is called *composition*), and then multiplying all vertices (vectors) by this matrix. For now, we should just be aware of the significance of matrices and vectors and anticipate their use.

Another prevalent concept in graphics is the notion of parametric equations. We can think of the equation of a line in 2-D as $y = mx + b$. Where m is the slope and b is the value of y where the line intersects the y-axis. We can also look at it by considering the variables separately. Thus a line from (x1, y1) to (x2, y2) can be represented as $x(t) = x1 + \Delta x * t$ and $y(t) = y1 + \Delta y * t$, where $\Delta x = x2 - x1$, $\Delta y = y2 - y1$, and t is a parameter which may or may not be bounded at one or both ends. If t is unbounded at both ends, this gives the equation of a line, while if t is bounded at one end, the result is the equation of a ray (used in ray tracing). If t is bounded at both ends (often in the range 0.0 to 1.0) this gives the equation of a line segment, which we'll see in clipping algorithms. Δx and Δy form a vector $(\Delta x\ \Delta y)$, so this is sometimes called the point-vector form of a line (often shown as $p(t) = p1 + ct$, where p1 is the starting point and c is the directional vector). Any point along the line can be specified by setting the parameter t to some value. This easily extends to 3-D lines by simply adding another equation. Another formulation of a line is called the point-normal form. In this case, we use the knowledge that the perpendicular to a slope $(\Delta x\ \Delta y)$ is $(-\Delta y\ \Delta x)$ and express the line as a dot product $(P \cdot N = D)$, where P is the point and N is the normal. The nice thing about this equation is it works in both 2-D (for lines) and 3-D (for planes).

These parametric equations (and variants on them) are critical in efficient algorithms for clipping (eliminating lines that shouldn't be displayed), ray tracing, shading, and other important components of the graphics pipeline. Many formulations of equations to intersect lines or rays with other objects will reduce to closed-form solutions for computing the value of the parameter t.

For example, if we wish to determine if and where two line segments intersect, we can use the point-normal form of the line containing one segment $(N \cdot P = D)$ in conjunction with the point-vector form $(P(t) = P_1 + Ct)$, where $C = P_2 - P_1)$ of the other segment. If $N \cdot C = 0$, this means the two segments are parallel and thus cannot intersect. Otherwise there is some value of t for which the lines intersect, and if that value is in the range $[0.0 \rightarrow 1.0]$ and the point of intersection falls between the endpoints of the first line, we have our solution. We solve for t as follows:

$$D = N \cdot P(t) = N \cdot (P_1 + Ct) \tag{2.2}$$

$$D = N \cdot P_1 + N \cdot Ct \tag{2.3}$$

$$(D - N \cdot P_1)/N \cdot C = t \tag{2.4}$$

Reading Topics: Vectors and vector operations, representations of lines and planes, introduction to vector graphics, Hill Chapter 4.1 - 4.6, 4.10 (Case Study 4).

Project 2: starting at a user-specified point within a box, draw a line whose initial direction is specified by a vector (input by the user) to the first wall it encounters. Compute a new direction by reflecting the original vector off of this wall and continue drawing the line. Continue this process for either a fixed number of bounces or until the user terminates the program. You should use parametric equations in your calculations where possible. The command to draw a line is *XDrawLine()* in **X** and *drawLine()* in **Java**. You will need to set the drawing color (called the foreground color) prior to any line drawing commands. See the documentation on your graphics package for specifics. [this could be the basis for a simple video game, huh?]

Summary of Useful Vector Math

Given two 3-D vectors A and B, where $A = (a_1 \;\; a_2 \;\; a_3)$ and $B = (b_1 \;\; b_2 \;\; b_3)$, and a scalar value S,

$$addition: \;\; A + B = (a_1 + b_1 \;\; a_2 + b_2 \;\; a_3 + b_3) \tag{2.5}$$

$$dot\ product: \;\; A \cdot B = a_1 * b_1 + a_2 * b_2 + a_3 * b_3 \tag{2.6}$$

$$scaling: \;\; S * A = (S * a_1 \;\; S * a_2 \;\; S * a_3) \tag{2.7}$$

$$magnitude\ of\ a\ vector: \;\; |A| = \sqrt{(a_1^2 + a_2^2 + a_3^2)} \tag{2.8}$$

$$the\ unit\ vector\ of\ A = (\frac{a_1}{|A|} \;\; \frac{a_2}{|A|} \;\; \frac{a_3}{|A|}) \tag{2.9}$$

$$cross - product: \;\; AxB = (a_2b_3 - a_3b_2 \;\; a_3b_1 - a_1b_3 \;\; a_1b_2 - a_2b_1) \tag{2.10}$$

$$magnitude\ of\ the\ sum: \;\; |A + B|^2 = |A|^2 + 2(A \cdot B) + |B|^2 \tag{2.11}$$

Deriving the Equation for Reflecting a Ray off a Surface

Assume vector V is hitting a surface with normal vector N. Let E be the projection of V onto the surface, and M be the projection of V onto N. The reflection vector R is computed as follows.

$V = E + M$

$R = E - M = V - 2M$

$M = \frac{V \cdot N}{N \cdot N} N$

$R = V - 2\frac{V \cdot N}{N \cdot N} N$

Chapter 3

When Light Hits a Surface

The entire field of graphics can basically be summarized by a single question, namely "what color should this pixel be?", repeated for each pixel of the image. The answer to this question depends on a lot of factors: the intensity and type of light sources, reflectivity and refraction characteristics (including texture) of a surface, and the relative positions and orientations of surfaces, lights, and the camera or viewer. If we assume for the moment that we are only considering grey-scale images [remember black-and-white TVs?], we can "normalize" the problem by saying that all pixels must have a value between 0 (black) and 1 (white). We can also say that a light source has an intensity between 0 and 1 (although "hot spots" from over-exposure can be simulated by capping the resulting pixel at 1, which allows light sources to have higher values). Finally, we can set the reflectivity (color) of a surface to be a value between 0 and 1.

It is known that a surface appears brightest when the light source is directly above it, which means about the only way to arrive at a pixel of color 1 is to have a light source of intensity 1 directly over a surface of reflectivity 1. If the surface is oriented so that its normal is not pointing directly at the light source, the color of the surface dims (we say it is attenuated). A simple model for this attenuation is Lambert's Cosine Law, which says the apparent color of the surface is proportional to the light source intensity, the surface reflectivity, and the cosine of the angle between the normal vector and a vector pointing at the light source. Because this light model does not depend on where the camera or viewer is, it is called *diffuse reflection* (i.e. it is scattered equally in all directions). If the absolute value of the angle is greater than ninety degrees, the light cannot directly illuminate the surface and the diffuse reflection is 0. Thus if N is the normal vector for a surface, L is a vector from a point p on the surface to the light source (assumes light source is concentrated at a single point), I_L is the intensity of the light source, and K_d is the diffuse reflectance of the surface, the intensity at point p is given by

$$I_p = I_L K_d \frac{N \cdot L}{|N||L|} \tag{3.1}$$

Another component involved in the shading of a pixel is the *specular reflection*, which accounts for shiny highlights and is based on both surface properties and the viewer location. One shading

model which includes specularity is the Phong Reflectance Model, which says that surfaces generally reflect a certain amount of the light intensity in a directed manner, without modifying the color of the light [which is why a red apple can have a white highlight]. The intensity of the highlight is proportional to a shininess or surface specularity constant K_S (again between 0 and 1) and the angle between the reflection vector R and a vector V which points at the viewer. The exact proportion is somewhat arbitrary (since this model is not based on optics to any great degree), but is usually set as the cosine of the angle between the 2 vectors raised to some power (the higher the power, the tighter the highlight). Assuming R and V are unit vectors, a typical formulation would be

$$I_p = I_L K_d \frac{N \cdot L}{|N||L|} + I_L K_S (R \cdot V)^{200} \tag{3.2}$$

Note that any value for I_p which exceeds 1.0 must be capped at 1.0, and any component resulting in a negative value must be set to 0.0.

A final term in computing the intensity of a point on a surface consists of a component called the **ambient intensity**, I_A. This experimentally derived constant is meant to encompass all the light that reaches a surface after it bounces off of other surfaces (which is why you can still see objects which are occluded from direct exposure from light sources). Again, this is a value between 0 and 1, and is usually just a user-defined constant. An advanced graphics topic, called *radiosity*, attempts to compute a real value for combined ambient and diffuse reflection for each surface. For our purposes, the final equation for intensity using all three components is

$$I_p = I_A + I_L K_d \frac{N \cdot L}{|N||L|} + I_L K_S (R \cdot V)^{200} \tag{3.3}$$

There are many variants on this computation. Some seek to incorporate the distance to the light source, as according to physics the energy reaching a surface drops off proportional to the square of the distance to the light source. Some attempt to model the optics of reflectance more accurately, as in the Torrence-Sparrow Specularity Model which represents the surface as a series of perfectly reflecting micro-facets whose variability in orientation helps control the resulting intensity. Others, as we will see in ray tracing, can incorporate both translucency (seeing through the surface) and the reflection of other objects on the surface. For our purposes, however, the more simplified computations above will be adequate.

To extend this model to work with color instead of grey-scale, we simply compute separate equations for red, green, and blue. Both the light source(s) and the object surfaces can have components from each primary. The number of distinct colors available, however, is limited by the size of your color map. So unless you have a 24-bit color system (true color, as opposed to pseudo-color), we will generally restrict ourselves to either grey scale or a small number of colors each with 32-64 levels, again normalizing our results to select the appropriate color. Color specification is a tricky business - people don't have really good intuition about the amount of red, green, and blue in a particular color. For this (and other) reasons, alternate color models have been derived. However, before we can examine color models we need to cover some background on color specification.

Color can be defined by three components: the location of the *dominant wavelength* (the spike in the spectrum for the color which indicates the *hue*), the *luminance*, or total power of the light (the area under the spectrum), and the *saturation*, or percentage of the total luminance found at the dominant wavelength. Adding white to a color decreases the saturation. Although these terms are easy to understand, it is difficult to determine these values given an arbitrary color.

The RGB scale is rooted in human color perception, based on the responsiveness of the various *cones* in our optic system (*rods*, another component, are sensitive to light level, and not color). All colors can be specified as a linear combination of three saturated primaries (though they don't have to be red, green, and blue). However, to combine to form another saturated color, at least one of the coefficients may have to be negative! To get around this quandary, the CIE Standard was created, which defines three "supersaturated" non-existent colors (X, Y, and Z) on which all other colors can be based with positive coefficients. If we assume the total of the linear combination is 1, this defines a plane in 3-D, where the color frequencies form an arc bounded by 0 and 1 in X, Y, and Z. In fact, since $x + y + z = 1$, we only need the X and Y values, which makes displaying the CIE Chromaticity Diagram on paper relatively easy. White is defined at position (.310, .316), and this is the midpoint between any pair of complementary colors. Also, the dominant wavelength of any color can be determined by drawing a line from white through the color and computing where the curve defining the pure spectra is intersected.

We can now choose three arbitrary points on this diagram to form a *color gamut*, which describes all possible colors which can be formed by linear combinations of the colors (basically, any color within the triangle defined by the points). Every distinct output device may have a different gamut, though the diagram provides a mechanism for matching colors between gamuts. If we choose a representative Red, Green, and Blue, we can envision this as a cube with three axes (though it is straightforward to convert back and forth between this and the CIE equivalent). The point $(0, 0, 0)$ corresponds to black, $(1, 1, 1)$ is white, and all points along the diagonal are shades of grey. The distance between a color and this diagonal also gives the saturation level.

The HLS (hue, lightness, saturation) color model is formed by distorting the RGB cube into a double hexagonal cone, with black and white at the extreme points and hue being specified by the radial angle (the primary and secondary colors form the vertices of the hexagonal cross-section). The line between black and white again gives the lightness value, and saturation is the radial distance from this line. The HSV model is similar, except it consists of a single cone, and white is found in the center of the cone base. Color specification based on either of these models tends to be more intuitive than the RGB model, though it is straightforward to transform colors in one model to another.

Two final color models of note are the CMY and the YIQ systems. CMY (cyan, yellow, and magenta) is used for *subtractive* color devices, such as many printers. The values specify how much color to subtract from white to attain the desired color (it actually subtracts the complement of the specified color). The full values for each produce black, instead of white as found in the RGB system. YIQ stands for luminance, hue, and saturation, and is used for broadcast TV (black-and-white TV's only look at the luminance component of the signal).

Once we've selected a color for a surface, we are now interested in using it to shade the object. We can examine shading models based on whether all the pixels within a polygon are colored the same

value (uniform shading) or if the values vary across the polygon. Uniform shading is adequate for distant polygons (since any variation based on direction to the light source will be minimal) and small polygons which are not approximations to curved surfaces. However, when a curved surface is being approximated by a mesh of planar polygons, uniform shading can lead to an interesting perceptual phenomena known as the *Mach band effect*. Basically, the human visual system tends to accentuate intensity differences between adjacent regions, causing the lighter of the two regions to appear even lighter near the border, and the darker region to seem even darker. Thus minor intensity variations can appear substantial, which reveals the planar aspect of the approximated surface. The most common solution to this problem is to make intensity variations very small, which can be accomplished by having smooth changes across each polygon as opposed to using uniform shading.

The two most common non-uniform methods rely on interpolation between vertices and across scan lines. In each case, you first compute an average normal at each vertex based on the polygons which share that vertex. In Gouraud shading, you then compute an intensity for a vertex based on the average normal, and then interpolate intensities along each edge. Once you have an intensity for each edge point, you can then interpolate intensities across each polygon along scan lines. In Phong Shading (yes, this is the same Phong who developed the specular reflection model) you interpolate the normal vectors themselves instead of the intensities. This involves a significant amount more processing (3 times as much interpolation, plus the calculation of intensity at each point), but gives you a better result, as assuming a linear interpolation of intensities does not accurately reflect the dependency of intensity on an angular relationship.

One common problem with computing the vertex normal by averaging is that significant ridges or valleys may get "smoothed out". Consider, for example, the average normal vector at the corner of a cube. This would point outward from the vertex at a 45 degree angle to each of the edges of the cube, and the resulting interpolation would generate a smooth shift in intensity rather than a crisp edge. Thus in performing the averaging operation one needs to insure that the polygons meeting at the vertex have similar normals. Otherwise one must simply use different normals for a given vertex based on which polygon is being filled.

As most real-world objects are not perfectly smooth and single-colored, it is useful to examine ways in which texture could be added to a given surface. Texture can be classified as either *geometry-based* or *reflectivity-based*. A geometry-based texture modifies the shape of the surface, such as by breaking each polygon into many small polygons and slightly modifying the positions of some or all of the verticies. We could, for example, get a ridged texture by imparting a square wave on the y-values while keeping x and z constant. A common way to create a "fake" geometry-based texture is to perturb the surface normals in a manner that emulates a geometric modification, without changing the geometry itself. This process is known as *bump-mapping*. This perturbation can either have a random component to it or follow a deterministic pattern.

Texture can also be imparted by modifying the diffuse or specular components of the shading equation. These methods would fall into reflectivity-based methods. Indeed, bump-mapping can be considered a reflectivity-based method that simulates a geometry-based technique. Texture-mapping, though, is commonly equated with the process of controlling the diffuse component of the shading computations. We can use either random variations or fixed patterns to set the diffuse reflectivity value. The key is to realize that the surface being viewed is rarely going to be aligned so

that the surface normal and viewing directing are parallel. Thus we might need to map the texture pattern in such a way that a variable number of pixels on the image are controlled by a single texture pixel, and vice versa. When we start working with parametric curved surfaces we will find it relatively straightforward to map the two parameters of a texture image to the two parameters of the vertex generation for the surface.

Reading Topics: Shading models, color theory, Hill Chapters 8.1 to 8.3, 8.5, and 12.

Project 3: Implement the Phong Reflectivity Model (not Phong Shading!) for a single 3-D polygon (a triangle will do). Input should be the vertices of the polygon (you should compute the normal), the location and intensity of the light source, the location of the camera (assume it is on the z-axis for now), and the ambient, diffuse, and specular components of the polygon. Assuming that you are using uniform shading, compute the appropriate color for the surface (you may need to set your color map to a ramp of grey values) and display it, either using a filled polygon command or the scan-line polygon fill algorithm described in Chapter 1 (becoming familiar with this algorithm will benefit you greatly for later projects). For now, just ignore the z component of each vertex when drawing the triangle. We'll look at projections later. In computing the vectors to the camera and light positions you should average the coordinates from the polygon's vertices to give an approximate center point.

You may wish to allow the user to interactively move the light source so you can view the changes in intensity to convince yourself the algorithm is working. Alternatively, you could just have the light move around in circles about the polygon (or rotate the polygon, if you've read ahead to the Chapter dealing with transformations). Note that if the diffuse component becomes negative (which means the dot product results in a value less than 0), it means the light is on the other side of the surface and you should only see the ambient component (i.e. you never subtract light!). We'll use this same property of the dot product later to determine which surfaces of a 3-D object are facing away from the viewer and thus will not be drawn.

Chapter 4

Building a Scene - Lots of Simple Parts

Now that you can render a single polygon, let's look at more complicated object models. The first step in this process is to determine the coordinate system in which polygons are to be specified. The simplest technique is to create what might be termed a 3-D screen coordinate system. If we generate points on the screen using x-y coordinates in the range of 0 to N, we could consider extending this such that all objects are defined by vertices whose coordinates fall within the (integer) range 0 to N. This is a bit restrictive, in that we are limited to a finite number of locations at which to place vertices. However, it does help simplify the transition between modeling and rendering.

When speaking of 3-D coordinate systems one can envision two distinct configurations of the three perpendicular axes. The first, called a *left-handed coordinate system*, has the positive x-axis going to the right, the positive y-axis going up, and the positive z-axis going away from the viewer. As the name implies, if you hold the thumb, index, and middle fingers of your left hand roughly perpendicular to each other, the thumb can indicate the x-axis, the index finger shows the y-axis, and the middle finger is the z-axis. Likewise, a *right-handed coordinate system* can be used, with the only difference being that the positive z-axis comes towards the viewer rather than away. Different text books and graphics systems vary as to which system is used for defining a scene, and sometimes this will differ from the coordinate system used to specify the camera view. We will deal with this potential confusion later on, but you should be aware of its existence, especially if you are using different text books to assist in learning the algorithms of computer graphics.

Given a scene defined within range of 0 to N, we can now map world coordinates (w_x, w_y, w_z) to screen coordinates (s_x, s_y) by simply mapping two of the world coordinates to the two screen coordinates. This, as we shall see later, is a form of *parallel projection*. There are 3 possible distinct mappings of this type (top, front, and side views) if we discount views which differ only by a 90 degree rotation (e.g. $(w_x, w_y) \Rightarrow (s_x, s_y)$ versus $(w_y, w_x) \Rightarrow (s_x, s_y)$). The other 3 orthogonal (along an axis) views (bottom, back, and the other side) can be obtained by subtracting each of the coordinates from N (think of it as a mirror in three dimensions). You should verify that you are capable of generating the 6 views of your polygon from the previous module prior to developing

a more complex model.

3-D modeling techniques can be roughly broken down into 2 categories: solid modeling or surface modeling. We will concern ourselves for now with surface modeling, which represents the outside surface of objects using planar polygon *patches*. In general, you do not want to manually enter the vertices and edges of every surface patch, especially since most non-trivial objects can have hundreds or thousands of patches. Thus it would be more effective to write a program which generates this information, either hard-coded or which converts a high level object description into a low level representation. Parametric surfaces (described in a later Module) is one way to do this. *Primitive instancing* (where you write functions for simple building blocks and create complex objects out of these primitive) is another. If you think of LEGO blocks, you get the basic idea. A scene would be described as a set of simple objects, each with its own size, position, color, and orientation. This is the route I recommend for this Module. Thus, for example, you might have a high-level description of your scene such as

```
BRICK Red (0, 0, 0) (10, 5, 20)
BRICK Black (0, 5, 0) (10, 5, 20)
PRISM Grey (0, 10, 0) (10, 3, 20)
```

which would consist of a grey prism on top of a black brick, which itself was on a red brick, located with one corner at the origin, 23 units in total height, and extending by 10 and 20 units in the x and z directions.

If we assume we have 3 data structures (a vertex list, an edge list, and a polygon/patch list) to hold our model, we can write functions which add to these structures in a consistent fashion. The vertex list simply contains 3-D coordinates. The edge list has 2 indices into the vertex list for each edge. The polygon list has a variable number of indices into the edge list, along with color/reflectivity information. Each edge should belong to exactly 2 polygons. A critical point in setting up these structures is that you must be careful to list the edges of a polygon in a consistent order (e.g. counter-clockwise while looking down on the polygon) so that your surface normal gets computed correctly. If you are not careful, some normals will face outward and some will face inward, which will cause the shading algorithm to give you incorrect results. This highlights one of the strengths of primitive instancing, as once you figure out the appropriate normal vectors for each surface of your primitive, this will simply get copied into all instances of the primitive (excluding the case where rotational variation is permitted). Thus continuing our previous example, we might have a routine *MakeBrick(int col, int loc[3], int size[3])* which is passed a color, location, and dimension information, and adds new vertices, edges, and polygons to the above-mentioned data structures. To do this, it would include a *template* brick, which defines the components of a 1 by 1 by 1 brick located at the origin. This would be scaled, repositioned, and added to the lists.

```
.
.
/* add new vertices to the vert list, adjusting for size and position */
vert[new].x = brick[i].x * size[0] + loc[0];
vert[new].y = brick[i].y * size[1] + loc[1];
```

```
vert[new].z = brick[i].z * size[2] + loc[2];
new++;
.
.
```

We should now be able to create an image of our scene by simply rendering each polygon individually using the previous Module (in conjunction with the world-to-screen mapping described above). However, if we just render all polygons in the order in which we've specified them, we are ignoring two pieces of information: the orientation of the polygon and its distance relative to other polygons which may occlude or be occluded by it given the specified view point. From any arbitrary view of our scene, approximately half of all surfaces will not be visible because they are facing away from the camera. It is important to remove these surfaces (called *back faces*) prior to our rendering process. If we assume our views are limited to the 6 mentioned earlier, this is actually a very simple procedure. Let us set the camera to be at the coordinate $(N/2,\ N/2,\ 0)$, looking down the positive z-axis (in the middle of one of the faces of our cube-shaped world). Any polygon which has a positive z-component to its normal will, in this case, be oriented away from the camera, and therefore can be eliminated from consideration. In the more general case of arbitrary viewing, one would simply eliminate any polygon which results in a positive value from the dot product of the surface normal and the vector describing the direction of the view (from the camera to a point being observed). For our simple viewing along each axis, this vector will be a vector with two values of 0.0 and 1 value of 1.0 (e.g. (0 0 1)). The two coordinates corresponding to the zero values will be those that map to the two screen coordinates. We can consider the axis along which we are viewing to be the z-axis of our *viewing coordinate system*, to go along with the two screen coordinates.

Once the back faces have been removed, we want to render the remaining polygons in such a way that each pixel of the resulting image is shaded based on the polygon nearest to the camera which covers that pixel (if any). In other words, for each pixel you wish to determine a) which polygons of your scene would include that pixel if rendered, and b) which of these is closest to the camera at this point. There are three basic strategies (actually there are others, which you can read about) to solve this problem. The simplest to implement is called the **z-buffer** algorithm. Basically 2 2-D arrays are maintained; the first is the image you are generating and the second keeps track of the depth (or z value, in our case) of the current pixel to be generated at each location. The depth array is initialized to some large value (in our restricted world, this can simply be N+1). Each polygon is then rendered in turn, one pixel at a time. At each pixel you compute the depth value at that location on the 3-D patch. If this is smaller than the depth at that location in the depth array, compute the pixel intensity and fill it into the image array and update the depth array. This can be a slow process, as decisions are made on a pixel basis, but it works with arbitrarily complex objects and scenes.

To compute the z-value for a particular x-y point on a polygon, there are a few possible algorithms. The first is to compute the plane equation for the polygon (using the cross-product), inserting the x and y values, and solving for z. Care must be taken to insure that computations are done in the viewing coordinate system, and not the world coordinate system. An alternate approach is to augment the scan-filling algorithm to interpolate z-values along each edge and across each scan-line.

If you do not have polygons which *penetrate* other polygons (a polygon *penetrates* another polygon if

any intersection between the polygons is not explicitly represented in the edge list of both polygons), you can improve the speed of hidden surface removal by making decisions for multiple pixels along a given scan line at a time. This is known as the *scan-line hidden surface removal* algorithm. We extend the scan-line polygon fill algorithm by putting ALL edges into the calculations, keeping track of which edges belong to which polygons. Each polygon is also augmented with a flag which indicates whether, for the current scan-line and position, you are inside that polygon (this is just a toggle as you encounter the edges of a polygon). At each edge in the sorted edge list, you are either entering or exiting a polygon. If you are entering, you compute the z value at the point of entry. If it is closer to you than any polygon you are currently inside of, you color pixels in the polygon's color up to the next element in the sorted edge list. Otherwise you continue coloring based on the nearest polygon prior to hitting this one. When you exit a polygon, you simply start coloring in the next nearest polygon (or in the background color).

The previous algorithm makes decisions which affect multiple pixels along a single scan line. A third algorithm, known as *depth-sorting* or the *Painter's algorithm*, attempts to make decisions which affect an entire polygon. The basic idea is that if you draw the polygons from furthest to nearest, you should get the correct results. This algorithm again depends on the absence of penetrating polygons, and also can run into serious problems with polygons which have concavities (you can end up with situations where no ordering of polygons gives you correct results). This algorithm starts by sorting the polygons based on furthest z value. This produces the correct results for many situations, but not all. In situations where 2 (or more) polygons have overlapping ranges of z values, you need to do a few more tests. If the polygons don't overlap in x and y ranges as well, there are no problems; they can be rendered in any order. If they do overlap in all extents (ranges), you can feed the vertices of one polygon into the plane equation of the other. The sign of the result will tell you what side of the polygon you are on. If all vertices agree in sign, your decision is made. The ugliest cases require you to clip one polygon against another (we'll talk about clipping later), but hopefully this won't arise. If your scene is that complex, you should probably stick with the z-buffer algorithm.

Reading Topics: Polygonal boundary representations, hidden surface removal, Hill Chapters 3.1-3.2, 6.1-6.2, 8.4, and 13.

Project 4: Create a boundary model for a simple, non-convex object (i.e. it has some surfaces which may be partially blocked by other surfaces) using a vertex-edge-polygon representation. The object should be specified in 3-D screen coordinates. Create 3 orthogonal views of the object as described above (top, front, side), and display all edges (this is called a *wire-frame* view). Now remove all back faces from each view (compare surface normals to the axis the camera is looking down). Finally, implement a hidden surface removal algorithm (scan-line, z-buffer, or depth sort are fine, and each can use the scan-line fill algorithm described in Module 1) and verify that it works correctly around any concavity in your object. If there is a chance that you've got penetrating polygons in your model, you'll have to do the z-buffer algorithm.

Chapter 5

The Object, The World, and the Eye

Thus far we've been describing objects in what I call a 3-D screen coordinate system. This has greatly simplified many of the operations we've performed, but is too limiting for a general-purpose rendering pipeline involving arbitrary views. We start by differentiating between *world coordinates* and *screen coordinates*. World coordinates tend to be 3-D floating point values, with some boundaries within which all objects are defined. Screen coordinates, on the other hand, are 2-D, generally positive integers bounded by the resolution of your output device. Assuming we wish to display all objects in our world, we can convert world coordinates to screen coordinates by translating and scaling values in the range of the world coordinates to the range of the screen coordinates. You simply make use of the fact that the relative position of any point between the minimum and maximum values for that coordinate is kept constant. Thus coordinate transforms of this type reduce to an offset and a scale factor.

Specifically, if we are looking along the z-axis of a world bounded by the points $(wx_{min}, wy_{min}, wz_{min})$ and $(wx_{max}, wy_{max}, wz_{max})$, with screen coordinate boundaries (sx_{min}, sy_{min}) and (sx_{max}, sy_{max}), we can convert any arbitrary vertex (wx, wy, wz) to its screen location (sx, sy) using the following equations:

$$sx = sx_{min} + (wx - wx_{min}) * (sx_{max} - sx_{min})/(wx_{max} - wx_{min}) \tag{5.1}$$

$$sy = sy_{min} + (wy - wy_{min}) * (sy_{max} - sy_{min})/(wy_{max} - wy_{min}) \tag{5.2}$$

This is easy to derive by noting the position of the point relative to the boundaries must remain constant between the different coordinate systems. To generate the other orthogonal views one simply switches which world coordinate will map to which screen coordinate and update the above equations accordingly.

One tricky problem with this mapping is that of maintaining a consistent *aspect ratio* between the two sets of boundaries. The aspect ratio of a rectangular area is the ratio of the height to the width.

If the world boundaries form a square, for example, and the screen boundaries form a horizontally oriented rectangle (width greater than height), all objects mapped between the two coordinate systems will appear wider than original (a circle would map to an ellipse with a horizontal major axis). To display the entire scene from the world on a given area of the screen, the ideal scenario is to have identical aspect ratios between the boundaries. Failing that, we would need to draw into a subregion of the screen area, with blank space either along the top/bottom or left/right side, depending on how the aspect ratios differ.

Other coordinate systems come in handy in computer graphics as well. One can use a master coordinate system to define a master copy of a particular primitive and then transform instances of this to populate a world. For example, by creating a tree with its own origin and then displacing the x and z coordinates of each vertex of the tree by the same amount, we can place this tree anywhere on the x-z plane. Multiple copies could be used to create a forest. Another useful coordinate system is the eye or camera coordinate system. We can obtain arbitrary views of the world by defining an eye coordinate system and transforming all vertices in the world into this coordinate system (more on this in Module 7).

The key to "hopping" between coordinate systems is the transformation matrix. If we represent a 3-D point as a 4-D vector (the use of the fourth component, which we call w, will be apparent shortly), we can represent an arbitrary transformation of this point by a 4 x 4 matrix. This is known as a *homogeneous coordinate transformation*. We preform the transformation by multiplying the 4 x 4 transformation matrix by the 4 x 1 vector, giving another 4 x 1 vector. A *Translation* moves a point by a certain displacement in one or more coordinates, and is indicated in the transformation matrix if any of the first three entries in the bottom row are non-zero. For example, $x' = x + 0y + 0z + \Delta x$, where Δx is the value in the lower left of the 4 x 4 matrix, x is the original x-coordinate, and x' is the new x-coordinate. Note that to get this result, the diagonal values must be set to 1. Other transformations can be specified similarly:

Scaling of any dimension means that one of the diagonal values of the matrix must be equal to a value other than 1. Values greater than 1 will enlarge objects, while those between 0 and 1 will shrink objects. Negative numbers will rotate objects around their axes as well as change the size.

Rotation about any axis is accomplished by placing sines and cosines at appropriate locations in the matrix. In general, rotation is performed about a single coordinate axis (see matrices and their derivation at the end of this section), and arbitrary rotations are done via combining translations and one or more axis rotations.

Shearing modifies the value of a coordinate by an amount proportional to one or both of the other coordinates. For example, one might want to offset x and/or y proportional to z to give a sort of false perspective. Entries in the matrix which are not along the diagonal or in either the last row or column have a shearing effect. Thus we can see by analyzing the matrices for rotation that this consists of both a scaling and a shearing component. Shearing by itself is not as commonly performed as the other transformations.

Translation, scaling, rotation, and shearing are all examples of what are called *affine* transformations. Most graphics books will give you examples of all of these transformations (be wary: some

books differ as to whether the vector is multiplied by the matrix or vice versa, which causes some variation in which elements are set in the matrix). Basically, all cells of the matrix perform a certain operation. The beauty of this formulation is that you can "compose" a complex transformation by multiplying a sequence of simple transformations, so that each vertex of your object or world only needs to be multiplied a single time once you have the complete matrix. In this way you can create a transformation which has rotations, scales, and translates in it without having to figure out the matrix components by yourself.

But what is the fourth column for? First, it allows us a clean mechanism for integrating translation into a transformation matrix. Certainly, we could do all translations without matrix multiplication by simply offsetting the resulting coordinates in an appropriate manner. However, in compound transformations which might include multiple translations it is much more efficient to consolidate all transformations into a single matrix. Also, remember that w is initialized to 1. If any transformation causes w' to be not equal to 1, you need to divide the all components (x', y', z', w') by w'. In effect, this allows you to scale all coordinates by either a constant (the lower right element is not 1) or by a factor proportional to x, y, or z (which we'll see in perspective projections).

In summary, we can view the 4 by 4 homogeneous transformation matrix as affecting each of the original coordinates in the following manner:

$$\begin{pmatrix} \begin{matrix} scale\ x \\ by\ constant \end{matrix} & \begin{matrix} offset\ y \\ proportional\ to\ x \end{matrix} & \begin{matrix} offset\ z \\ proportional\ to\ x \end{matrix} & \begin{matrix} scale\ everything \\ proportional\ to\ x \end{matrix} \\ \\ \begin{matrix} offset\ x \\ proportional\ to\ y \end{matrix} & \begin{matrix} scale\ y \\ by\ constant \end{matrix} & \begin{matrix} offset\ z \\ proportional\ to\ y \end{matrix} & \begin{matrix} scale\ everything \\ proportional\ to\ y \end{matrix} \\ \\ \begin{matrix} offset\ x \\ proportional\ to\ z \end{matrix} & \begin{matrix} offset\ y \\ proportional\ to\ z \end{matrix} & \begin{matrix} scale\ z \\ by\ constant \end{matrix} & \begin{matrix} scale\ everything \\ proportional\ to\ z \end{matrix} \\ \\ \begin{matrix} offset\ x \\ by\ constant \end{matrix} & \begin{matrix} offset\ y \\ by\ constant \end{matrix} & \begin{matrix} offset\ z \\ by\ constant \end{matrix} & \begin{matrix} scale\ everything \\ by\ constant \end{matrix} \end{pmatrix} \tag{5.3}$$

Reading Topics: Coordinate systems, affine transformations, Hill Chapter 5.

Project 5: Modify your modeling program (Module 4) to allow the definition of objects in floating point world coordinates. Create a scene of multiple simple objects by applying affine transformations to copies of primitives (e.g. you could build a house of bricks). Make sure that you include at least one example of each type of transformation. Incorporate a world-to-screen coordinate transformation to display your scene in the same three views you used in Module 4. Note that by changing this coordinate transformation you can zoom in or out of each view (don't get too close, though, until we've implemented clipping).

Common 3-D Homogeneous Coordinate Transformation Matrices

$$General\ formulation:\ (x', y', z', w') = (x, y, z, w) * \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix} \tag{5.4}$$

$$Translation\ by\ (d_x, d_y, d_z):\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix} \tag{5.5}$$

$$Scaling\ by\ (s_x, s_y, s_z):\ \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

$$Rotate\ about\ z-axis\ by\ \alpha\ degrees:\ \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

$$Rotate\ about\ x-axis\ by\ \alpha\ degrees:\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.8}$$

$$Rotate\ about\ y-axis\ by\ \alpha\ degrees:\ \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.9}$$

$$Shear\ y\ proportional\ to\ x:\ \begin{pmatrix} 1 & sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

Rules for Composing Transformations

1. Transformations of the same type are commutative. In general, transformations of different types are not (e.g. a rotation followed by a translation is not the same as the translation followed by the rotation), though rotation is commutative with uniform scaling.

2. Successive translations or rotations are additive. Successive scalings are multiplicative.

3. Compose transformations left to right.

4. Scaling is relative to the origin. To scale a vertex about an arbitrary point (a, b, c), translate the vertex by (-a, -b, -c), perform the scaling, and then translate the result by (a, b, c). Alternatively, multiply the 3 transformation matrices together, and then multiply the homogeneous coordinate representation of the vertex by the resulting matrix ($P' = P(Tr_{(-a,-b,-c)}Sc_{(sx,sy,sz)}Tr_{(a,b,c)})$).

5. Rotation is relative to the origin. To rotate a vertex about an arbitrary point (a, b, c), translate the vertex by (-a, -b, -c), perform the rotation, and then translate the result by (a, b, c).

Derivation of Rotation Transformation Matrix

Assume we are rotating about the z-axis by α degrees. Given a point $(x,\ y)$, to compute $(x',\ y')$ it is easiest to convert the problem to polar coordinates. Thus our initial position would be $(r\ *\ cos\beta,\ r\ *\ sin\beta)$ and the final position would be $(r\ *\ cos(\alpha+\beta),\ r\ *\ sin(\alpha+\beta))$. But what is β? Well, we don't even have to compute it if we remember a bit of trigonometry.

$$x' = r\ *\ cos(\alpha+\beta) = r\ *\ cos\alpha\ *\ cos\beta\ -\ r\ *\ sin\alpha\ *\ sin\beta \tag{5.11}$$

$$y' = r\ *\ sin(\alpha+\beta) = r\ *\ sin\alpha\ *\ cos\beta\ +\ r\ *\ cos\alpha\ *\ sin\beta \tag{5.12}$$

If $x = r\ *\ cos\beta\ and\ y = r\ *\ sin\beta$, we get

$$x' = x\ *\ cos\alpha\ -\ y\ *\ sin\alpha \tag{5.13}$$

$$y' = x\ *\ sin\alpha\ +\ y\ *\ cos\alpha \tag{5.14}$$

These are the equations to combine to form the rotation transformation matrix. The derivation for rotation about the x and y axis are similar, although the signs of the *sin* components are reversed in rotating about the y-axis.

Chapter 6

Clipping to the Field of View

Often there will be components of our scene which are not visible to the viewer because of the viewer's position and direction of viewing. It is common, for instance, to want to zoom in on a scene or do a fly-over. We could avoid drawing objects which shouldn't be mapped to the screen by simply setting boundaries for the view and ignoring any point during hidden surface removal which falls outside this boundary. This is OK for simple scenes or scenes where most objects are visible, but is a tremendous waste of computations otherwise. If possible, objects which are not going to be visible should be eliminated from consideration as soon as possible. One way of doing this is by examining the *extents* (bounding boxes) of objects against the region of the world which will project to the screen. Any object whose bounding box is totally outside this region doesn't need to be processed. But what about objects which are partially inside and partially outside the region?

If we assume that our region of interest is bounded by 6 planes, we can decompose the problem into a sequence of steps which compares each polygon of the model against a plane. Each plane can either eliminate the polygon (it is entirely outside), preserve the entire polygon (totally inside), or break the polygon into one or more components. Any polygon that survives the clipping process of one plane is passed on to the next one until all 6 planes have had a go at it or the polygon is completely eliminated. Anything left is within the region and can be passed on to the hidden surface removal process.

There are many algorithms for clipping a line against another line (some which extend easily to clipping against a plane). Some algorithms assume that edges or polygons are being clipped against lines or planes which are parallel to one of the axes (i.e. the equation is something simple, such as $y = 0$). This can greatly simplify the formulas necessary to perform the clipping. It also makes the process of eliminating or accepting entire line segments very straightforward. One popular method is called *Cohen-Sutherland Clipping*. This assumes a clipping rectangle in 2-D and a clipping box in 3-D. Each boundary of the clipping shape divides the space such that points are either inside or outside the boundary, and vertices are classified by a bit pattern (4 bits in 2-D, 6 bits in 3-D), where a bit is set to 1 if the vertex is outside the corresponding boundary. If both vertices of line segment have a classification of 0000 (or 000000), this means both are within the clipping rectangle (or box) and the segment should be drawn (trivially accepted). If the logical AND of the two bit

patterns is non-zero, that means both vertices are outside of a common boundary and the segment can be eliminated from consideration (trivially rejected). All remaining segments may intersect the boundaries at 0, 1, or 2 places.

To find the potential intersection point(s), there is a trade-off between algorithm complexity and efficiency. One simple method is a binary search, which divides an edge in half and computes the Cohen-Sutherland classification for the mid-point. The algorithm determines if either half can then be trivially accepted or rejected, and if not, repeats the division process. This can be a slow process, but for situations where there are only a small number of segments, it is good enough.

Another method simply computes the intersection point between the line (extend the segment infinitely in each direction) and each boundary (also extended). The intersection point is then checked to see whether it falls between the endpoints of the line segment. If the segment goes from (x_1, y_1) to (x_2, y_2) and we have a vertical boundary with equation $x = x_{bound}$, the intersection point is given by the equation

$$y = y_1 + m(x_{bound} - x_1), \ where \ m = (y_2 - y_1)/(x_2 - x_1) \tag{6.1}$$

For a horizontal boundary with equation $y = y_{bound}$, the intersection is at

$$x = x_1 + (y_{bound} - y_1)/m \tag{6.2}$$

The main disadvantage of this technique is its reliance on floating point multiplication and division, which can be costly in execution time for large numbers of segments.

There are several very efficient algorithms which use parametric equations of lines and point-normal forms of planes to compute the parameter value t for which the line hits the plane. The idea is that for each line segment (defined by 2 points), its direction is either taking it from inside the boundary to outside, outside the boundary to inside, or it is parallel to the boundary. By finding the range of t values for which the line is within all boundaries, we get the coordinates of the intersection points by sticking those t values into the parametric line equation. Now, since we are interested in just the line segment between the original 2 points, we only need to consider t values bounded by 0 and 1. Anything outside that range is not part of the segment.

The *Liang-Barsky* clipping algorithm is based on this notion. If (in 2-D) we assume the boundaries of the clipping rectangle are specified by (x_{min}, y_{min}) and (x_{max}, y_{max}), and the parametric equations for the line are $x = x_1 + d_x t$ and $y = y_1 + d_y t$ (where $d_x = x_2 - x_1$ and $d_y = y_2 - y_1$), we can write the following inequalities:

$$x_{min} \leq x_1 + d_x t \leq x_{max}, \ and \ y_{min} \leq y_1 + d_y t \leq y_{max} \tag{6.3}$$

We rewrite this as $p_k t \leq q_k$, for k = 1, 2, 3, 4. If $p_k = 0$, the segment is parallel with the boundary; if $q_k \geq 0$ the segment is inside the boundary, and otherwise it is outside. If $p_k < 0$ the extended

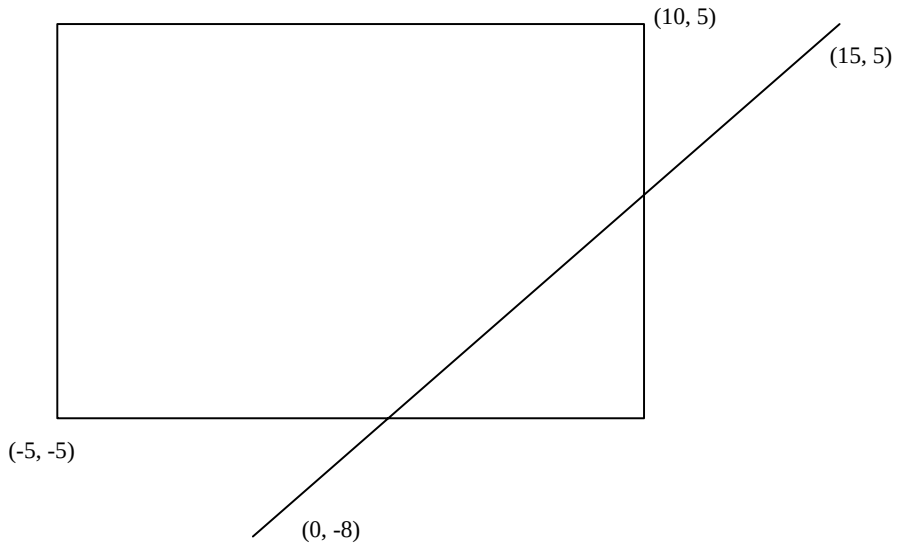
segment goes from outside the boundary to inside, and otherwise it goes from inside to outside. We now solve for t for each inequality, which gives us the intersection point. This is simply q_k/p_k, which we shall call r_k. We now separate the cases for which the segment is entering or exiting the boundary. We define $t_1 = max(0, r_k)$ *when* $p_k < 0$ and $t_2 = min(1, r_k)$ *when* $p_k > 0$. If $t_1 > t_2$, the segment is outside all boundaries. Otherwise we stick t_1 and t_2 into the parametric equations to get the intersection points. Note that this works in all cases, whether the segment starts within all boundaries, ends within all boundaries, or starts and ends outside all boundaries. The following figure shows the algorithm in action.

To clip a segment against an arbitrary boundary, we can follow a similar strategy. In this case, we represent the boundary in its point-normal form, $N \cdot P = D$, where N is the outward-pointing normal vector, P is a point on the boundary (if we put (x, y) or (x, y, z) into the equation, it will look familiar), and D is the solution given any point on the boundary. Let the segment in question go from P_1 to P_2, and $C = P_2 - P_1$ is the directional vector of our line segment (the same as d_x and d_y above). If $N \cdot C = 0$, the segment is parallel to the boundary, and if $N \cdot P_1 < D$ the segment is entirely inside the boundary. If the segment is not parallel to the boundary, the intersection point is given as $T = (D - N \cdot P_1)/(N \cdot C)$. If the denominator is negative, the segment is entering the boundary and all values of $t < T$ are invisible. Otherwise the segment is exiting and all values of $t > T$ are invisible. Note this works in 2-D or 3-D.

Once we can clip a line against a boundary, we can clip a polygon by keeping track, for each vertex, whether we are inside or outside the boundaries. All intersection points from clipping edges against the boundaries are used as part of the resulting polygon(s). Note that polygons with concavities can cause multiple polygons to result, which is why many people in graphics advocate sticking with simple, convex shapes. The main thing to remember is that you always want the result to be one or more closed polygons (or no points at all). Thus segments coinciding with parts of one or more boundaries may need to be added. Let us assume we are clipping a convex polygon (such as a triangle) against a set of boundaries. If we trace around the edges of the polygon we are clipping (choose either clockwise or counterclockwise), each edge will be either totally inside, totally outside, or partially inside each clipping boundary. All inside vertices of the original polygon will be included as vertices in the clipped polygon, while outside vertices may be eliminated (if both its adjacent vertices are also outside the boundaries) or replaced by the point of intersection between an edge shared by that vertex and the boundary crossed by that edge.

An algorithm based on this notion is the Sutherland-Hodgman polygon clipping technique. A convex clipping region is defined (a rectangle will do) with outward facing normals. If we start with a vertex list for the polygon to be clipped (with the first vertex repeated as last). we compare each pair of consecutive vertices with a single infinite clipping boundary (thus we will make four passes for a rectangular clipping region). For each pair of vertices, we may generate 0, 1, or 2 vertices for the newly clipped polygon based on the following algorithm:

1. if both vertices are inside the boundary, output the second one to the new polygon vertex list.
2. if the first vertex in the pair is inside and the second is outside, compute the intersection point of the edge and boundary and output it to the new polygon vertex list.



xmin = -5 xmax = 10 dx = 15 -5 <= 0 + 15t <= 10
ymin = -5 ymax = 5 dy = 13 -5 <= -8 + 13t <= 5

left: -15t <= 5 p0 = -15 q0 = 5 r0 = -1/3

right: 15t <= 10 p1 = 15 q1 = 10 r1 = 2/3

bot: -13t <= -3 p2 = -13 q2 = -3 r2 = 3/13

top: 13t <= 13 p3 = 13 q3 = 13 r3 = 1

t1 = max(0, ri) [r values where pi < 0] = max(0, -1/3, 3/13) = 3/13

t2 = min(1, rj) [r values where pj > 0] = min(1, 2/3, 1) = 2/3

Figure 6.1: An example of Liang-Barstky clipping.

3. if both vertices are outside the boundary, don't output anything.

4. if the first vertex in the pair is outside and the second is inside, compute the intersection point of the edge and boundary and output it AND the second vertex to the new polygon vertex list.

This algorithm works fine for convex clipping regions, though if the polygon to be clipped is complex and results in more than one isolated polygons, this algorithm will leave edges connecting the individual polygons. This may look fine, since these extra edges would be along the boundary and thus not very visible. However, it may wreak havoc with some polygon filling algorithms. A more powerful (and a bit more complicated) algorithm known as the Weiler-Atherton clipping technique is capable of clipping two arbitrarily shaped polygons against each other, without leaving extra edges between disjoint regions of the resulting clipped polygon.

We refer to the polygon to be clipped as the *subject* polygon, and the polygon which defines the clipping boundaries as the *clipping* polygon. We store the vertices of each polygon in lists which define each in clockwise order (thus the *inside* of the polygon is to the right as we move through the vertex list). We now compute the intersection points for each edge of one polygon against the other, storing the ones which fall between the original vertices in the corresponding position in each of the two vertex lists (i.e. each intersection point will be included in each list). Note that each intersection vertex can be classified as either *entering* the clipping polygon as we traverse the subject polygon or *exiting* the clipping polygon. Once the lists are complete and intersections are classified, we proceed as follows:

1. find the first intersection point in the subject list which is an *entering* point. This is the first point of the result polygon list.

2. traverse the subject polygon until another intersection point is found, adding each to the result polygon list. The point you are now examining is an *exiting* point.

3. remember where you left off in the subject polygon vertex list, and find the corresponding intersection point in the clipping polygon vertex list.

4. now traverse the clipping polygon until another intersection point is found, adding each to the result polygon list. The point you are now examining is an *entering* point.

5. if this point is NOT the first point of the result polygon, continue traversing as in step 2.

6. if this point IS the first point of the result polygon, you now have a complete, closed polygon. If all *entering* points in the subject polygon have not been included, find the next unused entering point and continue the process at step 2 with a new result polygon list.

Note the importance of the ordering of the vertices in each list. If they were not both listed in clockwise order the traversal would be much more complicated to implement. As it is, once the lists are created, the rest of the processing is trivial.

Reading Topics: Line and polygon clipping, Hill Chapters 3.3, 4.7 and 4.8.

Project 6: Write a program which clips an arbitrary (planar is OK) 3-D polygon against an arbitrary plane and displays it in 3 orthogonal views. The result should be 0, 1, or multiple polygons, based on the shape of the original polygon and position of the clipping plane. The polygon is obviously specified as a sequence of points, where the first and last points are the same. The plane can either be specified by 3 non-colinear points or by the 4 components of a plane equation. In the first case you will have to specify which side of the plane is to be considered inside (the direction of the normal).

Chapter 7

Perspective Projection and Arbitrary Viewing

Projection is the process where by data of dimension N is reduced to dimension M, where M is less than N. Our primary interest is projecting 3-D data to 2-D. Thus far we've done this by simply ignoring one of the dimensions. We now look at this process in more detail.

The act of projecting in graphics amounts to placing a plane of projection (PoP) between the eye/camera and the scene and having the points of the scene map to a location on this plane. What we've done to date has been a simple form of *parallel* projection, which assumes that if you were to draw lines between vertices in the scene and the points they map to on the plane of projection, the lines would all be parallel. In effect, we are assuming the eye/camera is at an infinite distance from the PoP. There are two primary categories of parallel projections: orthographic and oblique.

Orthographic parallel projection assumes that the plane of projection is perpendicular to the direction of the lines between the viewer and the scene (alternately, we could say the viewer is along the normal of the plane of projection). This is the most common form of parallel projection. The views down each of the three coordinate axes all fall into this category. We are not restricted to viewing our world down one of the coordinate axes, however. *Axonometric* views set the normal of the PoP so it is not aligned with any axis. *Isometric* projections look onto the world from a 45 degree angle in each dimension (a diagonal view), while *dimetric* projections position the normal of the PoP at an equal angle between two axes while allowing variation in the third. Finally, *trimetric* allows arbitrary off-axis positioning in all three directions.

Oblique parallel projection assumes that the plane of projection is not perpendicular to the lines between the viewer and the scene. This may be hard to envision until you realize that the transformation is simply a shearing in one or more dimensions. Thus we can look down one of the axes at a box which has been aligned with all three axes and see two or three faces, depending on whether the shear is in one direction or two. A shearing amount of 1 would make the side and top of a cube project to the same size as the front (called a *cavalier* view). A less dramatic view can be obtained with a shearing amount of .5 (called a *cabinet* view). Anyone who has taken a technical drawing

course has probably been exposed to this form of projection.

Alternatively, we could have the lines through the PoP converge to a camera position which is at a finite distance from the PoP. This is known as a *perspective* projection, which mimics the way humans view the world. If we map an edge from the scene which is parallel to the plane of projection, we note that its length on the screen is proportional to the relative positions of the edge, the camera, and the plane of projection. By using similar triangles we see that if the camera is at z=0, the plane of projection is at z=d, and a vertex is at z=z', the resulting coordinates on the plane of projection (px, py) are simply scaled versions of the original 3-D vertices (x', y'). This formula is simply px = x' * (d/z') and py = y' * (d/z'). This can be incorporated into our transformation matrix by making the third row, fourth column entry be 1/d and the fourth row, fourth column be 0 (remember, if w' does not equal 1 we divide all components of the resulting vector by w'). Some text books have slightly different variations on this matrix formulation, usually because some place the plane of projection at z=0 instead of z=d. You should be able to work out the math to prove to yourself that it works correctly. There are many other variants on projections which you should read about, though they won't be crucial for your projects.

The last of the "required" transformations we are interested in is to allow viewing of our world from arbitrary locations and orientations. What we'd like to do is create a camera coordinate system and place all objects into this system prior to clipping, projection, hidden surface elimination, and so on. Up until now, we have had a very simple camera coordinate system which was aligned with one of the three world coordinate axes (with translation along the axis). We can envision this process in two ways; in the first, the camera is moved and all objects remain fixed. In the second (the one we will explore), the camera remains fixed and all objects are moved. It should be clear that both interpretations can be applied to get a given scene. Thus the viewing transformation can be defined as the transformation which maps objects from the world coordinates into the camera coordinates. We assume the origin of the camera coordinate system is at the center of the plane of projection (the view reference point - VRP). We then need 3 orthogonal vectors to correspond to the axes of the coordinate system (often referred to (U, V, N) to avoid confusion with the world coordinates (X, Y, Z)). The N axis can be defined in a few ways. I like to use a "look at" point, L, in the scene, in conjunction with the VRP, to form this vector. Thus N = L - VRP. You then need an "up vector", V, which one could envision as a head or camera tilt. This must be perpendicular to the N vector to create a valid set of axes. To insure this, we will use an approximation which isn't necessarily perpendicular and refine it to obtain the correct orientation. If V' is an approximate up vector, we can generate the U vector which is perpendicular to both V' and N by taking the cross product of V' and N. We can then get the final version of V by taking the cross product of U and N. It is critical that V' is not colinear with N, or else this won't work.

We now can use U, V, and N to create a 3 x 3 transform which we can apply to each point to put it into the new coordinate system. The only thing we have to do is subtract the VRP from each point before multiplying the matrix. Thus we can transform each point as follows:

$$P' = (P - VRP)M, \; where \; M = \begin{pmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{pmatrix} \tag{7.1}$$

Some texts provide a 4 x 4 matrix that incorporates both of these transformations. If we pre-multiply the VRP with the U, V, and N vectors we get $R = (r_x, r_y, r_z) = (-VRP \cdot U, -VRP \cdot V, -VRP \cdot N)$. Thus the final matrix is

$$P' = PA, \ where \ A = \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ r_x & r_y & r_z & 1 \end{pmatrix} \tag{7.2}$$

Another common way to derive the transformation matrix is by starting with an N axis and VRP and perform the transformations necessary to align this with the world Z axis and origin, after which we apply a rotation about the z axis to account for tilt (let us say α degrees). This method appears in some graphics texts and is a bit more intuitive to some students. We create the compound transformation in the following manner.

1. Translate to place the VRP on the origin: $T_{VRP} = (-VRP_x, -VRP_y, -VRP_z)$
2. Compute the unit vector for N: $N = (L_x - VRP_x, L_y - VRP_y, L_z - VRP_z)$, where L is the lookat point. The unit vector $= N/|N| = (a, b, c)$ where, $a = N_x/|N|$, $b = N_y/|N|$, $c = N_z/|N|$, and $|N| = \sqrt{N_x^2 + N_y^2 + N_z^2}$
3. Rotate about the x-axis until in xz plane (use projection onto yz plane (0 b c)): the disired angle β has a hypotenus $d = \sqrt{b^2 + c^2}$, and thus $\cos\beta = c/d$ and $\sin\beta = b/d$. This gives us the transformation matrix:

$$R_{x,\beta} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c/d & b/d & 0 \\ 0 & -b/d & c/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.3}$$

4. Rotate about the y-axis until onto the positive z axis (start with (a 0 d)): the desired angle γ has components $\cos\gamma = d$ and $\sin\gamma = -a$. This gives the transformation matrix:

$$R_{y,\gamma} = \begin{pmatrix} d & 0 & a & 0 \\ 0 & 1 & 0 & 0 \\ -a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.4}$$

5. Rotate about the z-axis by desired head tilt angle α. This gives the transformation matrix:

$$R_{z,\alpha} = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.5}$$

6. The entire transformation for each point would thus be: $P' = PT_{VRP}R_{x,\beta}R_{y,\gamma}R_{z,\alpha}$

Once the vertices of our world have been converted into the viewing coordinate system, we can apply what is termed a *prewarping* transform, which distorts the positions of our vertices to include perspective foreshortening. However, instead of mapping all depth values to the plane of projection, we conserve the relative depth of each point (called *pseudo-depth* while modifying the u and v coordinates as in the perspective projection transformation. If e_n is the position of the eye along the N axis (this is a negative number if you assume the plane of projection is at $N = 0$), the necessary transformation matrix is as follows:

$$P'' = P'W, \; where \; W = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/e_n \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.6}$$

Finally, we need to perform a *normalization* process, which converts vertices in prewarped eye coordinates into 3-D screen coordinates. To accomplish this, we need to specify a number of parameters. Let W_{left}, W_{right}, W_{top}, and W_{bottom} define the range of u and v on the plane of projection which will map to the screen. Similarly, let V_{left}, V_{right}, V_{top}, and V_{bottom} define the range of pixel coordinates in our viewport which will contain the resulting image. Finally, let F and B define the front and back planes along the N axis (both should be in front of the eye, with B set to allow easy clipping of distant objects). The normalization matrix will convert depth values so that points between F and B will fall between 0 and 1 and thus positions points to allow simple clipping to a box defined by (V_{left}, V_{bottom}, 0) and (V_{right}, V_{top}, 1). The transformation is as follows:

$$P''' = P''R, \; where \; R = \begin{pmatrix} S_u & 0 & 0 & 0 \\ 0 & S_v & 0 & 0 \\ 0 & 0 & S_n & 0 \\ r_u & r_v & r_n & 1 \end{pmatrix} \tag{7.7}$$

Where
$S_u = (V_{left} - V_{right})/(W_{left} - W_{right})$
$S_v = (V_{top} - V_{bottom})/(W_{top} - W_{bottom})$
$S_n = [(e_n - B)(e_n - F)]/e_n^2(B - F)$
$r_u = (V_{right}W_{left} - V_{left}W_{right})/(W_{left} - W_{right})$
$r_v = (V_{bottom}W_{top} - V_{top}W_{bottom})/(W_{top} - W_{bottom})$
$r_n = F(e_n - B)/e_n(F - B)$

We can now compose these three transformations and apply them to all vertices with the following equation:

$$P' = PAWR \tag{7.8}$$

The objects are now in position to be clipped and rendered with hidden surfaces removed. This completes the graphics rendering pipeline!

Reading Topics: Projections and arbitrary viewing, Hill Chapter 7.

Project 7: This final "required" project has several steps. I strongly recommend that you finish them one at a time rather than trying to incorporate all of the functionality in it at once. First, expand your clipping algorithm from Module 6 to clip the 6 sides of a user-specified view volume and test it on your model from Module 5. You should be able to move your camera along the axis so that it is in among your objects and generate correct views. Now implement a perspective projection; this can be specified by simply entering the distance to the plane of projection. The x and y bounds (in camera coordinates) on the plane of projection can be used to generate 4 of the plane equations for clipping. The user can then specify near and far clipping planes for the camera's z axis. Finally, integrate a viewing transformation which allows views to be generated from arbitrary locations and orientations. This can be done by having the user enter a view reference point (the center of the plane of projection), a look-at point, and an up vector.

Chapter 8

Introduction to Ray Tracing

The fundamental concept of ray tracing is quite simple: given a discrete array of points on the plane of projection, send a ray from the eye through each point and see if the ray hits anything. If it does, compute the surface normal at this point and use this (in conjunction with the light source location) to compute the value of the pixel. If the surface the ray hits is shiny, a new ray can be sent out on the angle of reflection, and any other object hit can contribute to the color of the original pixel. Likewise, if the surface is translucent, another ray can be sent through the surface via the angle of refraction, again contributing to the final pixel value. Finally, you can easily determine if the original hit location is in a shadow by sending a ray to each of the light sources and see if any of them hit another object prior to arriving at the light source. Thus there are only three basic capabilities we need: creating rays (which are simply parametric lines which start at the eye and go to infinity), computing intersections of a ray with each of the objects in our scene, and computing the surface normal at the hit point. For rays which propagate off of a surface we also need some way of combining the contributions of each additional ray, but we won't go into this here.

Creating the rays are the easiest part. You just determine the region of the plane of projection that will map to the screen, decide how many pixels to have in the horizontal and vertical directions, divide up the continuous values on the plane into discrete points, and create a point-vector form of a parametric line going from the eye though any of the points. If R is the view reference point in world coordinates and M is the viewing transformation matrix (UVN components), we can set the eye location E to be a distance of d behind the plane of projection along the N axis with the following: $E = (0\ 0\ -d)M + R$. We then need the range of values for u and v on the plane of projection and the number of pixels which will be rendered. If (u_{min}, v_{min}) and (u_{max}, v_{max}) represent the rectangle at the plane of projection (in eye coordinates) and *width* and *height* are the desired dimensions of the resulting image, we compute the coordinates of the discrete pixels on the plane of projection by simply dividing up the rectangle according to the desired size. If (u_i, v_j) represents a particular point in this rectangle, the equation for the ray through this point from the eye is $r_{ij}(t) = E + (u_i, v_j, d)Mt$, which we'll refer to as $r_{ij}(t) = s + ct$ for simplicity.

You now want to determine if there is any value of t, the parameter, for which you hit an object in your scene. Unfortunately, each ray has to be compared against each object (and maybe each surface patch) to see if they intersect, which can be very computationally intensive. This is even

more demanding if you have reflective and/or refractive rays as well as the ones going through the plane of projection. The basic strategy used in ray tracing is to find closed form solutions for intersecting a ray with various primitives, such as spheres (the easiest), cylinders, planes, and so on. Obviously if you have an analytic surface you can simply replace the x, y, and z of the surface with the parametric representations for x, y, and z of the ray and solve for t. For spheres this gives you a quadratic formula which has 0 (no hit), 1 (a glancing blow), or 2 (penetrating) solutions, and you just use the smaller of the t values for the hit point. For surface patches you would first figure if and where the ray hits the plane containing the patch and then determine if this is inside the patch.

As an example, assume you have a unit sphere at the origin, which has an equation $|P| = 1$ (in other words, $x^2 + y^2 + z^2 = 1$). We can square both sides to give $|P|^2 = 1$ and use the formula

$$|a + b|^2 = |a|^2 + 2(a \cdot b) + |b|^2, \ \textit{where } a \textit{ and } b \textit{ are vectors} \tag{8.1}$$

to give

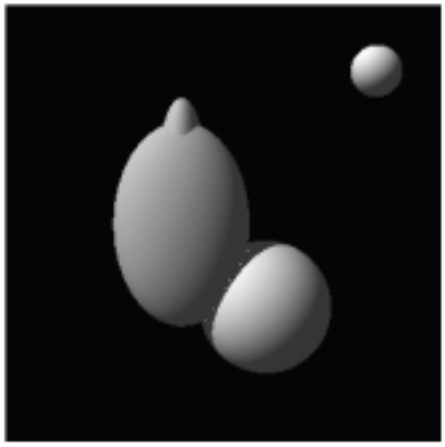
$$|s|^2 + 2(s \cdot ct) + |c|^2 t^2 = 1 \tag{8.2}$$

after substituting s for a and ct for b. This is a quadratic formula in t of the form $At^2 + 2Bt + C = 0$, where $A = |c|^2$, $B = s \cdot c$, and $C = |s|^2 - 1$. The solution is $-B \pm \sqrt{B^2 - AC}/A$, which gives 0, 1, or 2 solutions for the value of the parameter t where the ray intersects the sphere.

One interesting trick to use is that if you have defined your scene as a set of transformed unit primitives, all you need is the equation of a ray intersecting the unit primitives (usually much simpler than arbitrarily located and oriented shapes). You then would apply the inverse of the primitive transformation to the ray before doing the intersection calculation. This is similar to the difference between moving the camera and moving the scene that we covered in the discussion on viewing transformations. Thus if a unit sphere is translated by a displacement d and rotated and scaled via a transformation matrix M, we simply translate the ray by -d and then transform it by M^{-1}. The inverse can be composed by noting that the inverse of a scaling transform S is simply 1/S, and the inverse of a rotation about an axis is achieved by reversing the signs of the sin components. Each ray would thus need to be transformed for each object it is to intersect, but the savings in calculating the intersections is worth it. We simply need to store the inverse transformation for each object in the scene.

An advantage of using this trick is that, for several primitives, computing the surface normal at the hit point is greatly simplified. For example, the surface normal at a hit point on a unit sphere centered at the origin is simply the coordinates of the hit point. You can then get the transformed normal by applying the non-translational component of the object's original transformation. This complexity of normal calculation is not necessary for planar patches, but is useful for parametric or analytic surfaces. The first example below is generated only with unit spheres which have been transformed.

Most text books which cover ray tracing provide you with closed form solutions to ray intersection with various primitives, along with ways of computing the surface normal at the hit point. One of

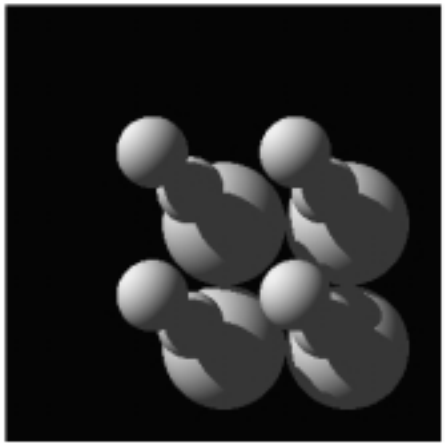


Ray tracing with penetrating ellipsoids

the nice things about ray tracing is you don't have to worry about objects penetrating each other - this is perfectly legal and will give you the correct surface each time. One thing to be careful of is, if you plan to reflect rays off of shiny surface, don't do any form of clipping before hand (another nice aspect of ray tracing is that clipping is not necessary). Just be wary that ray tracing grows in computational requirements proportional to the image size, number of objects, and amount of recursion you apply for reflective and refractive surfaces. Any recursion should be bounded in depth to avoid the possibility that the first ray never completes! The second example below casts a second ray at each intersection point to determine if the hit point is in a shadow. Thus with a few lines of code and approximately twice the computations we can add this feature.

Reading Topics: Introduction to ray tracing, Hill Chapter 14.

Project 8: Write a simple ray tracer based on transformed spheres. Your input file should contain



Ray tracing with shadows

a set of sphere descriptions (location, scaling [not necessarily uniform], rotation, reflectivity values), light parameters, and viewing parameters. With minor modifications you should be able to add shadows. Other possible extensions include reflections, refractions, and environment mapping, though you'd need to have significant free time to tackle some of these capabilities.

Other Useful Primitives and Their Intersections:

unit plane: if we have a unit plane at z = 0, the intersection point for the ray is $t = -s_z/c_z$, which are the z components of the point and vector. For planar patches, you then determine if the intersection point is within the patch.

unit cylinder: the equation for the cylinder is $x^2 + y^2 = 1$. We follow the same procedure as with spheres and get a quadratic formula in t with $A = c_x^2 + c_y^2$, $B = s_x c_x + s_y c_y$, and $C = s_x^2 + s_y^2 - 1$.

infinite cone: the equation for an infinite cone is $x^2 + y^2 - z^2 = 0$. The quadratic formula for the intersection points would be $A = c_x^2 + c_y^2 - c_z^2$, $B = s_x c_x + s_y c_y - s_z c_z$, and $C = s_x^2 + s_y^2 - s_z^2$.

The book *An Introduction to Ray Tracing*, edited by Andrew Glassner (Academic Press, 1989) gives equations for a number of other useful shapes.

Chapter 9

Curved and Fractal Surfaces

In general, it would be quite tedious to manually enter a boundary representation for complex surfaces, so we look for methods for algorithmic generation. We can consider two classes of such methods: fractals (for rough surfaces) and parametric curves (for smooth surfaces).

Fractal surfaces are usually generated by taking an existing surface (which may simply be a triangular patch), decompose it into a number of smaller shapes, perturb the vertices, and repeat this process on each of the smaller shapes. This can be done for any number of levels of recursion. What makes the process fractal, to the mathematical purists, is that the type and amount of perturbation is consistent over different scales (the self-similarity principle). Thus if you look at the decomposition of a patch and the resulting subpatches as one level of detail it will look very similar to what you'd see if you zoomed in on a sub-sub-patch, for example. Common examples of fractal surfaces seen in computer graphics are mountains and clouds. Fractals are also used in 2-D graphics to generate rough boundaries, as in a coastline.

Fractal methods can also be applied to generate objects such as plants and trees (these are sometimes referred to as *graftals*. We can think of a branch of a tree as being a scaled down, translated, and rotated version of an entire tree, and branchlets have a similar relationship to the branch. The idea is to find a way to represent an object as a combination of smaller versions of itself, and determine constraints on perturbing the transformations which still preserve the integrity of the desired objects. Thus angles and lengths of components may shift somewhat, but there are limits on the variations. Another method used to specify fractals is in terms of grammars (also known as *L-systems*), where legal sentences of the grammar describe the components of the fractal objects. Production rules are used to compose the allowable objects, and random numbers are used to impart some variability in scale, position, and orientation. This formalism greatly enhances the extensibility of a fractal generation process, as modifying and adding production rules and primitives can generally be done with great ease.

In contrast to fractals, parametric curves are useful for creating surfaces which are smooth in nature. If we start with a short curved line, we can try to find a polynomial equation in t which specifies how x, y, and z (assuming we are in 3-D) are formed. Thus for a quadratic polynomial we might have $x(t) = At^2 + Bt + C$, where t goes from 0 to 1 (with similar equations for y and z). This

means x(0) = C and x(1) = A + B + C. To draw the curve we just decide how many values of t we will use (say 20) and step t through its range, generating coordinates for t=0, t=.05, t=.1, and so on. These are then connected. The smaller the step in t value, the smoother the appearance of the curve. The order of the polynomial dictates how many parameters control the curve and how complex the curve can get. In much of graphics people work with cubic (3rd order) polynomials. The question is how do we specify a particular curve? We may know where we want it to start and stop, but how is the interior specified?

The answer involves 2 concepts: *control points* and *blending functions*. The idea is that control points "attract" a curve to them, with the level of attraction varying over the length of the curve. For example, the first control point may force the curve to start at its location (maximum attraction) and then let up on it as t increases. Likewise, the last control point should have no influence on the curve at the beginning, but maximize its attraction at the end. Interior control points will have different influences which will peak for different values of t. We use the notion of a *blending function* to define the influence of a control point over the duration of a curve. Each control point will have a different blending function, and the entire curve is defined as the sum of the control points, each weighted by their blending function for that value of t. Each blending function should be a polynomial of the same order as the curve you are trying to generate, as the sum of 2 order-N polynomials is itself an order-N polynomial.

Most forms of parametric curves are defined by the form of their blending functions and how the control points influence things. Hermite curves, for example, define curves as a start and end point along with the first derivative of the curve at the start and end. Bezier curves have start and end points along with intermediate points (for order-N polynomial, there are N-1 intermediate control points); the curve goes through the end points, but in general doesn't touch the intermediate control points. B-splines allow an arbitrary number of control points, but only a fixed number are used in defining the curve at any given point (this gives you smooth transitions along complex curves, where the other forms of curves can't guarantee high levels of smoothness). In addition to control points and blending functions, B-Splines also require the specification of *knot points*. These indicate the spacing between points along the curve where one control point drops out and another starts influencing the curve. Different forms of B-Splines have different characteristics for knot points and blending functions. A well-known form is known as NURBS, which stands for Non-Uniform, Rational B-Splines, indicate that knots are not uniformly spaced, and the blending functions have a rational (divisor) component.

Each of the equations for x, y, and z for cubic parametric curves can be stated in matrix form; a vector holding t^3, t^2, t, and 1, a vector holding the control points, and a 4 x 4 matrix derived from the blending functions which is fixed for a particular class of curve.

One formulation of the Bezier curve is as follows: Assume we are using polynomials of order N, which means we have N+1 control points, each with a blending function of order N. We compute each of the parametric equations (for x, y, and z) using the following equations.

$$x(t) = \sum_{k=0}^{k=N} x_k * B_{k,n,t} \tag{9.1}$$

$$B_{k,N,t} = C_{N,k} * t^k * (1-t)^{N-k} \tag{9.2}$$

$$C_{N,k} = N!/(k! * (N-k)!) \tag{9.3}$$

Thus for a cubic curve we get

$$x(t) = x_0(1-t)^3 + 3x_1t(1-t)^2 + 3x_2t^2(1-t) + x_3t^3 \tag{9.4}$$

When multiplied out, this can be formulated as a matrix multiplication:

$$x(t) = TM_{bezier}P_x = (t^3\ t^2\ t\ 1) * \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{9.5}$$

This matrix is the same for the y and z parametric equations.

The formulation of the cubic Hermite curve is as follows (I give the parametric equation for x: the ones for y and z are the same). The generic form for a cubic polynomial is given by

$$x(t) = at^3 + bt^2 + ct + d \tag{9.6}$$

We have 4 unknowns (a, b, c, d) and thus need 4 constraints to solve for the unknowns. The constraints consist of the two end points for the curve as well as the slope of the curve at these end points, which we specify by $x(0) = x_0,\ x(1) = x_3,\ x'(0) = sx_0,\ x'(1) = sx_3$, where $x' = dx/dt$ and sx_0 and sx_3 are the x-components of the slope at the ends of the curve. We thus get

$$x(t) = \left[t^3\ t^2\ t\ 1\right] * [a\ b\ c\ d]^T \tag{9.7}$$

Let $C = [a\ b\ c\ d]^T$. Our constraints can now be specified as

$$x(0) = [0\ 0\ 0\ 1] * C \tag{9.8}$$

$$x(1) = [1\ 1\ 1\ 1] * C \tag{9.9}$$

$$x'(0) = [0\ 0\ 1\ 0] * C \tag{9.10}$$

$$x'(1) = [3\ 2\ 1\ 0] * C \tag{9.11}$$

We combine this into a matrix formulation to give

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix}^{-1} * \begin{pmatrix} x_0 \\ x_3 \\ sx_0 \\ sx_3 \end{pmatrix} = M_{hermite} P_x = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_3 \\ sx_0 \\ sx_3 \end{pmatrix} \tag{9.12}$$

This can be equated to a comparable Bezier curve if we define the slope components by the interior control points, so that $x'(0) = 3(x_1 - x_0)$ and $x'(1) = 3(x_2 - x_3)$. This can be obtained by multiplying C by the matrix

$$M_{herm \to bez} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{pmatrix} \tag{9.13}$$

One formulation for a segment of a B-Spline is as follows (this is only the briefest of introductions to this complex topic):

$$P(t) = T M_{bspline} G \tag{9.14}$$

$$T = \begin{bmatrix} t^3\ t^2\ t\ 1 \end{bmatrix} \tag{9.15}$$

$$G = [P_{1-1}\ P_i\ P_{i+1}\ P_{i+2}] \tag{9.16}$$

$$M_{bspline} = 1/6 * \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix} \tag{9.17}$$

Extending curved lines to curved surfaces is trivial; you just have 2 parameters (s and t), a grid of control points (for cubics this would be a 4 by 4 grid), and blending functions (in s and t). Thus, for example, x(s,t) would be the sum of all control points, each of which is scaled by the product of its vertical(s) and horizontal(t) blending functions. By making one of the parameters (say s) fixed and varying the other (t), you generate a series of points along a horizontal or vertical curve. You

then move the fixed parameter (s) and repeat. By stepping s through the range of possible values (0 to 1), you end up with a grid of points which can be linked together in a mesh, resulting in a surface.

A Bezier surface can be generated as follows:

$$P(s,t) = \sum_{j=0}^{j=m} \sum_{k=0}^{k=n} p_{j,k}\ B_{j,m,s}\ B_{k,n,t} \tag{9.18}$$

where $p_{j,k}$ specifies (m+1)*(n+1) control points and $P(s,t)$ is the point on the surface corresponding to the values of the parameters s and t.

Efficiency is an issue in computing these equations. We can use Horner's Rule to reduce calculating the cubic polynomial to a smaller number of computations.

$$f(t) = At^3 + Bt^2 + Ct + D \textit{ becomes } ((At + B) * t + C) * t + D \tag{9.19}$$

We can also use a process known as *forward differencing*, which computes the value of a function at location $(t + \Delta t)$ using the value computed at (t), along with a term corresponding to how the function is changing. Thus

$$f(t + \Delta t) = f(t) + \Delta f(t) \tag{9.20}$$

If Q is the step size for our parameter t, we get

$$\Delta f(t) = f(t + Q) - f(t) \tag{9.21}$$

$$\Delta f(t) = 3AQt^2 + (3AQ^2 + 2BQ)t + AQ^3 + BQ^2 + CQ \tag{9.22}$$

$$\Delta f(t + Q) = \Delta f(t) + \Delta^2 f(t) \tag{9.23}$$

$$\Delta^2 f(t) = \Delta f(t + Q) - \Delta f(t) = 6AQ^2 t + 6AQ^3 + 2BQ^2 \tag{9.24}$$

$$\Delta^2 f(t + Q) = \Delta^2 f(t) + \Delta^3 f(t) \tag{9.25}$$

$$\Delta^3 f(t) = \Delta^2 f(t + Q) - \Delta^2 f(t) = 6AQ^3 \tag{9.26}$$

Initially $f(0) = D$, $\Delta f(0) = AQ^3 + BQ^2 + CQ$, and $\Delta^2 f(0) = 6AQ^3 + 2BQ^2$.

Reading Topics: Parametric surfaces, fractals, Hill Chapters 9 and 11.

Project 9: Write a program which allows the user to enter a set of control points for a Bezier surface and generate and render a polygon mesh. If you are really ambitious, you could find the set of Bezier control points used to generate the Utah teapot. They can be found in some textbooks, and are probably on the net somewhere.

Chapter 10

Solid Modeling

Most of the 3-D modeling we've done so far (excluding some of the ray tracing stuff) has been focussed on surfaces. One of the problems with surfaces is that it is very easy to make mistakes (e.g. creating internal representations which are physically not possible or complete). The alternative to boundary representation is solid modeling, where all primitives are guaranteed to occupy 3-D space. Solid modeling is good for a variety of purposes beyond guaranteeing physically realizable objects. It is easy to derive properties such as length and volume from solids. It is also useful in Finite Element Analysis to have a space-filling representation. One problem with solid models is the rendering algorithms are often difficult or produce results of less quality than boundary models. This is often resolved by transforming the solid model into a boundary model prior to rendering.

The simplest form of solid model is called *spatial enumeration*, which just means that your 3-D world is represented as a 3-D array of equal sized cubes called *voxels* or volume elements. This is quite common in 3-D medical models formed by CAT scans or similar technology. It is obviously very wasteful in space, and the courseness of the surface normals one can compute makes resulting images a bit blocky. There are several strategies for rendering volumes represented by spatial spatial enumeration. Some are based on ray tracing, where rays proceed through the 3-D array until a voxel with sufficient value stops the ray. A normal is generated by examining the neighborhood of the hit point to approximate the orientation of a surface of this density. Another strategy, called *Marching Cubes*, scans the volume and creates small triangular patches to approximate the surface defined by a particular value. In effect, each set of eight neighboring voxels define the values at the corners of a cube, and the surface defined by the value will have 0 or more corners inside and 0 or more outside. By enumerating all possibilities and creating a configuration of triangles for each case, we can convert the volume into a boundary description and render it in the normal fashion.

The next several figures show different configurations of corner labelings and the triangles which would be used to represent the surface. The last figure contains an image of a hydrogen molecule potential field. It appears blocky because the vertices along each edge of the cube were taken as the midpoint of the edge rather than an interpolated position.

We can reduce some of the space problems by allowing cubes of different sizes, where a cube is either empty or full. A common method used for this is *octrees*, where space is divided into 8

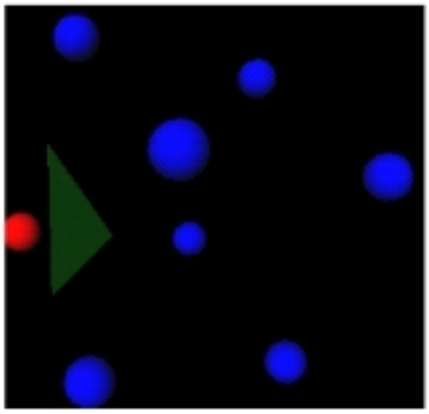


Figure 10.1: Configuration 2.

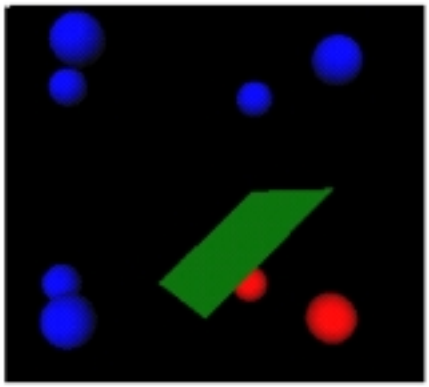


Figure 10.2: Configuration 12.

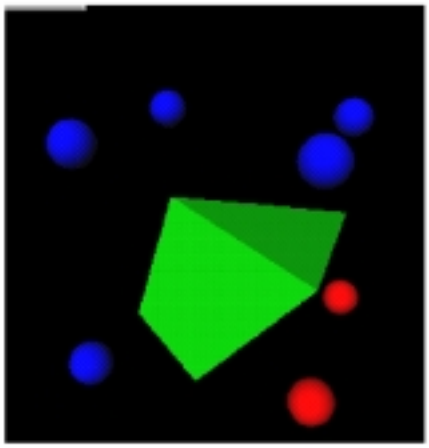


Figure 10.3: Configuration 14.

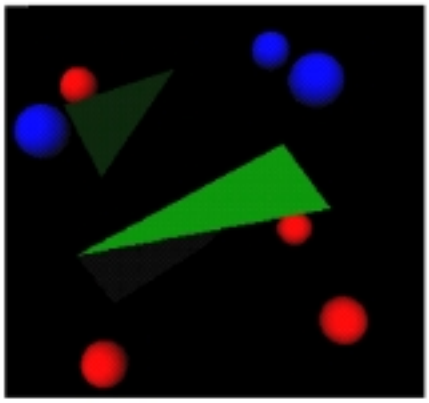


Figure 10.4: Configuration 30.

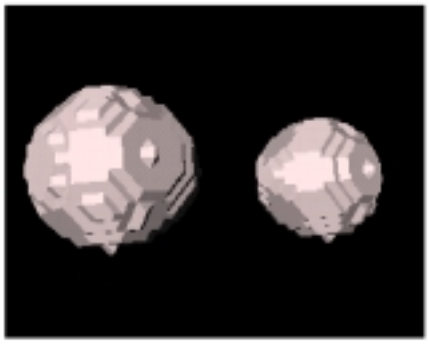


Figure 10.5: Hydrogen molecule potential field (data courtesy of AVS).

octants, and each octant which is not either empty or full is further subdivided. Octrees are very powerful representations; logical operations between objects represented by octrees can be done via tree traversal, with no arithmetic calculations at all. Hidden surface removal is performed simply by rendering the cubes in an order based on the viewer location, sort of like a painter's algorithm. However, other operations, such as rotating an object prior to merging it into another object, can be quite difficult, and can often introduce significant inaccuracies based on the depth of the octree. One difficulty in rendering octrees is determining an appropriate surface normal, since if we just render the cubes we get a very blocky output. Several extensions to octrees have been devised which augment the data structure with surface orientation information to ease this problem.

By allowing shapes other than cubes to be used, we arrive at a representation known as *primitive instancing*. Moderately complex scenes can be devised of parameterized instances of cubes, cylinders, spheres, and so on. There are, of course, limitations on the types of objects you can represent. There is no graceful mechanism, for example, to have objects with holes in them. An extension to this method which is quite popular in the CAD community is *Constructive Solid Geometry*, or CSG. By creating shapes out of logical combinations (AND, OR, SUBTRACT, XOR) of primitive shapes, and then using these shapes in combination, fairly sophisticated objects can be created. There are two (at least) common strategies for rendering objects or scenes represented by CSG or primitive instancing. One involves ray tracing, which gracefully supports the logical combinations of CSG by keeping track, for each ray, as to whether you are inside or outside of subtractive objects when you hit other surfaces. The tricky part is computing the correct surface normal. A second strategy involves converting to a boundary representation and rendering surfaces as in earlier chapters. The key problems involve handling penetrating and subtractive objects, both of which can add significant complexity. Much work has gone into determining closed form solutions for the boundary of intersection between various forms of primitive objects.

A representation which can be viewed as either a boundary or a solid modeling technique is called

a *sweep representation*. If we define a 2-D closed shape (cross-section) and a spine (a path in 3-space), we can extrude the shape along this path, always keeping the cross-section orthogonal to the spine. By connecting corresponding points on adjacent cross-sections and defining triangles or quadrilaterals between adjacent points on adjacent cross-sections, we can define a wide class of axially symmetric objects, and even more variety can be introduced if we allow the cross-section to be scaled as it moves along the spine (e.g. a chess piece). Another variant on this method, sometimes called *lathing*, allows you to spin a shape around an axis, again connecting corresponding locations on each instance of the shape. The angular steps between instances of the shape determines how smooth the resulting object is. Rendering objects defined in this manner is most readily performed by the methods described in the early chapters, although research has been performed on ray tracing certain forms of objects in this category.

Reading Topics: Solid modeling, Hill Chapter 6.3-6.6.

Project 10: Modify your ray tracer to allow CSG-like combinations (intersections, subtractions, unions) on pairs of subobjects. The key is to keep track of "subtractive" objects, so that when a ray hits one of these objects you don't render it at the first hit point. The tricky part is that if you are within a subtractive object and you enter an "additive" object, this object will only be seen if you exit the subtractive object first. In addition, the normal that you should use in shading is the inverse of the normal of the subtractive object at the location where you depart it. Intersections are easier, since you'll be rendering a point which is on the surface of one of the objects involved in the intersection. You will need to develop your own input file format for specifying these CSG operations (a binary tree representation is useful for internal storage).