Chapter 10: The Rendering Pipeline

This chapter looks at the way a graphics system implements the creation of an image based on the geometry and appearance information we have been discussing in the earlier chapters of the book. We introduced the graphics pipeline at the beginning of the book, and the rendering pipeline picks up the geometry produced by the graphics pipeline and produces the actual image specified by a graphics program. This involves computation of appearance information at each vertex produced by the geometry computation, rasterization of each geometric object into fragments that correspond to scanlines in the output raster, and several computations including color, depth, and texturing that place the fragment on the actual color buffer. We describe some of the techniques for breaking down a line segment into points on separate scanlines that determine the ends of the fragments and both linear and perspective-corrected interpolations of properties on these scanlines. In order to read this chapter, you should have an understanding of graphics primitives and the graphics pipeline that will enable you to see how the primitives are handled by the pipeline operations discussed in this chapter. After finishing this chapter you should have an understanding of the processes that graphics systems use to create images that may let you create more efficient graphics programs.

Introduction

In an earlier chapter we saw the outline of the graphics pipeline at a rather high level, and then we described how the API operations transform information in model coordinates to screen coordinates. With these screen coordinates, it is still necessary to carry out a number of operations to create the actual image you see on your screen or other output device. These operations are carried out in several ways, depending on the graphics system used, but in general they also have a pipeline structure that we will call the rendering pipeline because it creates the rendered image from the geometry of the graphics pipeline.

We should be careful to point out that the rendering pipeline as we will describe it applies mainly to polygon-based graphics systems that render a scene by processing each polygon through operations that develop its appearance as it is rendered. Not all graphics systems work this way. A ray-tracing system will generate a ray (or a set of rays) for each pixel in the display system and will calculate the intersection of the ray with the nearest object in the scene, and will then calculate the visible appearance of that intersection from properties of the object or from operations based on optical properties of the object. The rendering process here is simply the appearance calculation for each pixel, not of a whole polygon, even if the scene should be made up of polygons. Thus ray tracing has no rendering pipeline in the sense we describe in this chapter.

In this chapter we will look at the rendering pipeline for graphics systems based on per-polygon properties in some detail, describing the various operations that must be performed in rendering an image, and eventually focusing on the implementation of the pipeline in the OpenGL system.

The pipeline

When we begin to render the actual scene, we have only a few pieces of information to work from. We have the fundamental structure of the various pieces of the scene (such as triangles, rects, bitmaps, texture maps, lights, or clipping planes). We have the coordinates of each point that describes the geometry along with additional information for each point such as the color of the point, the normal at the point, the texture name and texture coordinates for the point, and the like. We also have the basic information that describes the scene, such as whether or not we have enabled depth buffering, smooth shading, lighting, fog, or other operations. The rendering task is to take all of this data, some of which will change from object to object in the scene, and to create the image that it describes.
This process takes place in several stages. In one, the graphics pipeline takes the original modeling coordinates for each geometry node and applies your modeling and viewing transformation to create 3D eye space vertex data from your original model data, creating a complete representation of the properties of the vertex. In a second, the vertex data for each polygon in your scene is interpolated to raster information to define the endpoints of each scanline in each polygon so that the properties of the pixels in the polygon may then be interpolated. In another, the color of each pixel is defined as the scanline is interpolated, using color data or texture data that is applied to the pixels as they are processed. In yet another, the data for each pixel is modified to apply effects such as depth testing, clipping, fog, or color blending. Overall, these processes provide the computations that make high-quality visual representations of the model you have defined with the image properties you have specified.

We have already seen how the geometric computations of the graphics pipeline are applied. When we discussed it previously, we focused on transformation operations on the vertices of the models that transformed the vertices from model space into 2D screen space. The eventual data structure that holds vertex information, though, holds a complete description of the vertex and is much richer than just the 2D x and y coordinates of a screen point. It also holds the depth of the pixel in the original model space, which is needed for accurate polygon interpolation; color information from simple color definition or a lighting computation, needed to determine the color for a polygon or for simple color interpolation; texture coordinates of the vertex, needed for texture mapping; and other information that depends on the graphics API used. For example, you would want to include the vertex normal in world space if you were using Phong shading for your lighting model.

The rendering pipeline, then, starts with the original modeling data (an original model vertex, data defined for that vertex, and the transformation to be applied to that vertex) and creates the screen representation of each vertex. That screen vertex is part of the definition of a polygon, but to complete the definition of a polygon the pipeline must gather the information on all the polygon’s vertices. Once all the vertices of the polygon are present, the polygon can begin to be rendered, which involves defining the properties of each pixel that makes up the polygon and writing those pixels which are visible to the graphics output buffer.

We will assume that the graphics hardware you will use is scanline-oriented, or creates its image a line at a time as shown in Figure 10.1. A scanline is the set of pixels on your display device that have the same value of y; it is one horizontal row. The set of pixels in a polygon on one scanline is called a fragment. Rendering the polygon requires that you define all the fragments that comprise the polygon and determine the properties of all the pixels in each fragment. Notice that on a convex polygon, each scanline will meet the polygon in a single fragment, while on a non-convex polygon there may be more than one line segment on a scanline. This is why most graphics APIs only work with convex polygons and require you to break up a non-convex polygon into convex parts before it is displayed. In OpenGL, for example, any polygon you define with the

![Figure 10.1: a scanline on a convex polygon (left) and general polygon (right)](image-url)
GL_POLYGON statement will be treated as if it were convex. If it is not, the image that is built will likely look quite strange.

Once you have defined the vertices of the polygon in screen space, the next step in the rendering pipeline is interpolating the polygon vertices to define the points on the edges of the polygon that are the endpoints of the scanline segments so you can process these segments and write them to the frame buffer. Here you will use either linear or perspective-corrected interpolation to determine the coordinates of the original point on the polygon that would be projected to the screen point that lies on the desired scan line. This same interpolation will also be applied to other properties that are defined for each vertex, such as z-depth or texture coordinates. The perspective-corrected interpolation affects the depth, texture coordinates, and possibly other data for the interpolated vertices. This is discussed in more detail in the later chapter on texture mapping.

Once you have the scanline endpoints (and their data) calculated, you can begin to create the pixels on the scanline between the endpoints, filling this interval of the polygon. Again, you must interpolate the data between the endpoints, and again, some perspective correction may be applied. Now, however, you have actual colors or actual texture coordinates of each individual pixel so you may determine the color each pixel is to be given. However, not all pixels are actually written to the output buffer because there may be depth testing or clipping applied to the scene, so there are now several tests that must be applied to each pixel before actually writing it out. If depth testing is being done, then the pixel will only be written if the depth is less than the depth at this pixel in the depth buffer, and in that case the depth buffer will be modified to show this pixel’s depth. If other clipping planes have been enabled, the original coordinates of the pixel will be recalculated and compared with the clipping plane, and the pixel will or will not be written depending on the result. If there is an alpha channel with the pixel’s color and the color is visible, color blending will be applied with the pixel color and the color in the image buffer before the pixel is written. If there is a fog effect, then the fog calculations will be done, depending on the pixel depth and the fog parameters, before the pixel is written. This set of pixel-by-pixel operations can be expensive, so most of it can be enabled or disabled, depending on your needs.

In addition to this set of operations for displaying pixels, there are also operations needed to create the texture information to be applied to the pixels. Your texture map may come from a file, a computation, or a saved piece of screen memory, but it must be made into the internal format needed by your API. This will usually be an array of color values but may be in any of several internal formats. The indices in the array will be the texture coordinates used by your model, and the texture coordinates that are calculated for individual pixels may well not be integers so there will need to be some algorithmic selection process to get the appropriate color from the texture map for each pixel in the pixel pipeline above.

The rendering pipeline for OpenGL

The OpenGL system is defined in terms of the processing described by Figure 10.2, which outlines the overall system structure. In this figure, system input comes from the OpenGL information handled by the processor and the output is finished pixels in the frame buffer. The input information consists of geometric vertex information, transformation information that goes to the evaluator, and texture information that goes through pixel operations into the texture memory. The details of many of these operations are controlled by system parameters that you set with the glEnable function and are retained as state in the system. We will outline the various stages of the system’s operations to understand how your geometry specification is turned into the image in the frame buffer.
Let us begin with a simple polygon in immediate-mode operation. The model-space vertex geometry that you specify is passed from the CPU and is sent to the polynomial evaluator for the full set of transformations and for the clipping operations, and is then forwarded to the per-vertex operations. At this point lighting operations are applied and the color data for each vertex is calculated. The result is the transformed vertex (along with the other information on the vertex that has been retained through this process), ready for primitive assembly and rasterization.

If you are compiling display lists instead of working in immediate mode, then the vertex and transformation operation data is sent to the polygon evaluator, but when the computations are finished the results do not go to the vertex operations for rasterization. Instead, the vertex and operation information goes into display list memory for later use. When the display list is executed, the operations are passed into the rendering process just the same as they would had they been passed in immediate mode, except that there is no computational overhead to apply the transformations and there may be some optimization done on the data as it is put into the display list.

From this point, then, the vertices of the completed primitives go into the rasterization stage. This stage applies the interpolation and scanline processing described above. Some aspects of this process, such as whether a perspective-corrected interpolation is performed as described in the chapter on texture mapping, may be controlled at this point. As the visibility and colors of the individual pixels are computed, appropriate color or texture data is computed for each depending on your specifications, and the resulting scan-line data is ready to go on the per fragment operations.

You may not have noticed the feedback line from the per-pixel operations to the CPU, but it is very important. This is the mechanism that supports pick and selection operations that we will discuss in a later chapter. Here the connection allows the system to note that a given pixel is involved in creating a graphics primitive object and that fact can be noted in a data structure that can be returned to the CPU and thus to the application program. By processing the data structure, the application can see what primitives lie at the given point and operate accordingly.

While we have been discussing the actions on vertex points, there are other operations in the OpenGL system. We briefly mentioned the use of the polynomial evaluator; this comes into play when we are dealing with splines and define evaluators based on a set of control points. These
evaluators may be used for geometry or for a number of other graphic components, and here is where the polynomial produced by the evaluator is handled and its results are made available to the system for use. This is discussed in the chapter on spline modeling below.

**Texture mapping in the rendering pipeline**

Texture mapping involves other parts of the rendering system. Here a texture map is created from reading a file or from applying pixel operations to data from the frame buffer or other sources. This texture map is the source of the texture data for rasterization. Information in the texture map, which is simply an array in memory, is translated into information in texture memory that can be used for texture mapping. The arrow from the frame buffer back to the pixel operations indicates that we can take information from the frame buffer and write it into another part of the frame buffer; the arrow from the frame buffer to texture memory indicates that we can even make it into a texture map itself. This is described by Figure 10.3.

Here we see that the contents of texture memory can come from the CPU, where they are translated into array form after being read from a file. Because OpenGL does not know about file formats, it can be necessary to decode data from the usual graphics file formats (see [Murray & vanRyper], for example) into array form. And, as we see in the texture mapping chapter, we can also fill the array as a result of computations. However, the texture memory can also be filled by copying contents from the frame buffer with the `glCopyTexImage(...)` function or by performing other pixel-level operations. This can allow you to create interesting textures, even if your version of OpenGL does not support multitexturing.

![Figure 10.3: processing for texture maps](image)

It is rare for an individual pixel to have a texture coordinate that exactly matches the indices of a texture point. Instead of integer texture coordinates, the pixel will have real-valued texture coordinates and a calculation will need to be done to determine how to treat the texture data for the pixel. This can involve choosing the nearest texture point or creating a linear combination of the adjoining points, with the choice defined by the `glTexParameter` function as described in the texture mapping chapter.

**Per-fragment operations**

Much of the power of the OpenGL system lies in its treatment of the fragments, or small sets of scanline data, that are computed by the rasterization process. The fragment operations follow a sub-pipeline that is described in Figure 10.4. Some of the fragment operations probably fall under
the heading of advanced OpenGL programming and we will not cover them in depth, although
many of them will be covered in later chapters. Most of them are operations that must be enabled
(for example, with glEnable(GL_SCISSOR_TEST), glEnable(GL_STENCIL_TEST), or
the like) and some require particular capabilities of your graphics system that may not always be
present. If you are interested in any details that aren’t covered adequately here, we would direct
you to the OpenGL manuals or an advanced tutorial for more information.

![Figure 10.4: details of fragment processing](image)

The first fragment operation is a scissor test that lets you apply additional clipping to a rectangular
bounding box defined by glScissor(...), and the second operation allows you to use a test
against a pixel’s alpha value to create a mask for textures, defined by glAlphaFunc(...). The
next operation applies a stencil test, which is much like the alpha test except that it creates a mask
based on values in a stencil buffer. The stencil operations are based on a stencil mask that you can
draw to with normal OpenGL operations and that is used to choose whether or not to eliminate a
pixel from a fragment when it is drawn. The stencil test is based on a comparison of the value in
the stencil buffer and a reference value, and each pixel in the fragment is either kept or replaced by
a value that you can set. The key functions for stencil testing are glStencilFunc(...), glStencilMask(...), and
glStencilOp(...) to set the test function, control writing to the stencil buffer, and
specify the actions for the stencil test.

The next set of operations are more familiar. They begin with the depth test that compares the
depth of a pixel with the depth of the corresponding point in the depth buffer and accepts or rejects
the pixel, updating the depth buffer if the pixel is accepted. Following this is the blending
operation that blends the color of the pixel with the color of the frame buffer as specified in the
blending function and as determined by the alpha value of the pixel. This operation also supports
fog, because fog is primarily a blending of the pixel color with the fog color. The dithering
operation allows you to create the appearance of more colors than your graphics system has by
using a combination of nearby pixels of different colors that average out to the desired color.
Finally, the logical operations allow you to specify how the pixels in the fragment are combined
with pixels in the frame buffer. This series of tests determines whether a fragment will be visible
and, if it is, how it will be treated as it moves to determine a pixel in the frame buffer.

Some extensions to OpenGL

In this section we’ll discuss programmable vertex and fragment operations and the idea of a
shading language, with the goal of giving you some background on these ideas. We expect that
future versions of OpenGL, or at least generally accepted extensions, will allow this kind of
programmable operations.

In standard OpenGL, when a vertex comes into the rendering pipeline we know much more about
it than just its coordinates. We also know its color (whether determined by a lighting model or by
simply setting the color), and perhaps its texture coordinate. There is no reason why we cannot define much more than this about a vertex, however. We can also store displacement vectors, up to eight multitexture coordinates, and particular transformations. Vertices can even store addresses of programs that can compute shape, color, anisotropic shading by computing lighting-oriented normals instead of geometric normals, or bump maps. Graphics cards are beginning to include quite a bit of per-vertex programmability with 16 or more 4D real vectors per vertex to hold additional data, although each card will have a distinct instruction set that is oriented to its particular architecture.

Besides this kind of program that can be attached to each vertex, however, we can also apply other techniques to per-fragment operations than are included in the fragment processing described above. There are programmable dot product operations in some graphics cards, modeled on the idea of texture combining operations, that will allow you to apply additional kinds of operations for processing fragments. The end result can be thought of as a programmable rendering pipeline, with three programmable stages: group processing, vertex processing, and fragment processing. Two of these are familiar from the OpenGL rendering pipeline in Figure 10.2, with group processing representing operations on collections of vertices instead of individual vertices for efficiency. This programmable pipeline is shown in Figure 10.5.

![Programmable pipeline with three programmable stages](image)

These give us a good idea of some kinds of developments we should expect to see in future versions (or extensions) of OpenGL or other graphics APIs. We should expect to be able to provide a program with each vertex to compute vertex properties such as we described above. For compatibility with a wide range of hardware, the language of such a program will probably be independent of the particular graphics card, and the graphics API will provide a way to either compile or interpret the language into the specific operations needed for the card. We expect that the language for per-vertex shaders will probably look a lot like RenderMan™, because this language is already designed for writing shader operations and is rather familiar to many in the computer graphics community. This language should provide at least surface shaders (programs that compute an RGBA color for the frame buffer) and light shaders (programs that compute light information for use with the surface shader), and likely other kinds of shader operations as well. It will be interesting to see what this does for advanced programming with graphics APIs.

An implementation of the rendering pipeline in a graphics card

The system described above is very general and describes the behavior required to implement the OpenGL processes. In practice, the system is implemented in many ways, and the diagram in Figure 10.6 shows the implementation in a typical fairly simple OpenGL-compatible graphics card.
The pipeline processor carries out the geometry processing and produces the fully-developed screen pixels from the original modeling space vertices as described above. The texture memory is self-describing and holds the texture map after it has been decoded by the CPU. The rasterizer handles both the rasterization operations and the per-fragment operations above, and the Z-buffer and double-buffered framebuffer hold the input and output data for some of these operations. The cursor is handled separately because it needs to move independently of the frame buffer contents, and the video driver converts both the frame buffer content and the other inputs (cursor, video) to the format that drives the actual monitor display mechanism. This kind of straightforward mapping of API functionality to hardware functionality is one of the reasons that OpenGL has become an important part of current graphics applications—it provides good performance and good price points to the marketplace.

The rasterization process

In our discussion of the graphics pipeline, we saw that the rasterization process plays a key role. Polygons come into the process in terms of their vertices, and the vertex geometry is translated into scanline-oriented fragments for further detailed processing, whether by texture mapping or by the various per-fragment operations. The end result is the polygon as it is meant to be displayed by your program. Rasterization is carried out by the OpenGL system so it is not necessary to understand it in order to do graphics programming, but there are details of the process that will help you understand some of the basic concepts of computer graphics. Here we will describe the rasterization process in some detail.

First, let’s recall the information that is present at each vertex in the geometry as it gets to the rendering pipeline. We have the 2D screen coordinates of the vertex, calculated by projecting the vertex from 3D eye space to 2D eye space and then mapping that space to screen space. We have the z-value of the vertex in 3D eye space, because we do not need to change it for the screen display but we do need it for some computations; this z-value may be in its original terms, or it may have been converted to a more convenient form such as the integer value having 0 at the front of the view volume and the largest integer at the back of the view volume, as OpenGL stores it. We have the color of the vertex, usually as an RGB triple, either given by the model or calculated from the lighting model. (If we were using Phong shading, instead of the color of the vertex we might have the normal at the vertex for later shading computations.) And we have the texture coordinates of the vertex. So each vertex carries a great deal more information than just its screen geometry.

As we go through the rasterization process, we must take the vertices from the geometry and scan-convert them—interpolate them and find the pixels the polygon will use for each scan line—to determine the total set of pixels that are displayed for the polygon. The scan conversion process operates first on each edge of the polygon to get the set of pixels that represent that edge. When this is done for all the edges, you will have all the pixels that bound the polygon. For a convex polygon, only two edges can intersect any one scan line, so you can organize the pixels into a set...
of pairs, one for each scan line. Each pair then determines a fragment by interpolating the pixels between them.

There are many algorithms for performing the rasterization itself, so we will choose the one that is probably the simplest: the DDA (Digital Differential Analyzer) algorithm. This algorithm takes the screen-space coordinates of the two endpoints of a line segment and uses the straightforward line equation and roundoff to calculate the pixels of the line segment on each scanline. Because each pixel’s coordinates are integers and a line segment is continuous, we must realize that we will create an approximation of the line segment, not the exact segment; we will make this a best approximation by calculating the pixel on each scanline that is the closest to the real-valued point on that scanline. So we will create an aliased version of the line segment, but it will be the best alias we can create.

To begin, let’s assume that our line segment has endpoint vertices \((X_1, Y_1)\) and \((X_2, Y_2)\) and let’s label \(\Delta X = X_2 - X_1\) and \(\Delta Y = Y_2 - Y_1\). For the sake of convenience, we will assume that \(\Delta X\) and \(\Delta Y\) are both positive; if they are not then you can adjust the algebraic sign in what we do below. Notice also that we can translate our line segment however we want, because once we calculate all the pixels for the line we can translate the entire line segment by translating each pixel of the segment. So we can assume that our line segment lies in the first quadrant.

Now the nature of the pixels for the line segment differs if \(\Delta Y > \Delta X\) or \(\Delta X > \Delta Y\). If \(\Delta Y > \Delta X\), then there will be only one pixel on each scanline, while if \(\Delta X > \Delta Y\), there will be only one pixel lying on any vertical line in screen space. We will develop our algorithm for the case \(\Delta Y > \Delta X\) because this makes us think of \(X\) as a function of \(Y\), an unfamiliar approach, but you can exchange the \(X\) and \(Y\) terms in the algorithm to deal with the other case.

So for each scanline between \(Y_1\) and \(Y_2\), we will want to compute the pixel on the scanline that best represents the exact point on the line segment. We will begin with the equation of a line that calculates \(X\) as a function of \(Y\):

\[
X = X_1 + ((Y - Y_1)/(Y_2 - Y_1)) \cdot (X_2 - X_1)
\]

Here the term \((X_2 - X_1)/(Y_2 - Y_1)\) represents the slope of the line in terms of \(\Delta X/\Delta Y\) instead of the more usual \(\Delta Y/\Delta X\) because we are calculating the value of \(X\) as a function of \(Y\). Once we have calculated the value of \(X\) for each (integer) scanline \(Y\), then, we simply round that value of \(X\) to the nearest integer to calculate the pixel nearest the actual line. As pseudocode, this becomes the algorithm:

```plaintext
Input:  two screen points \((X_1,Y_1)\) and \((X_2,Y_2)\) with \(Y_2 > Y_1\) and with \((Y_2-Y_1)/(X_2-X_1) > 1\)
Output: set of pixels that represents the line segment between these points in screen space
for (int Y = Y1; Y < Y2; Y++) {
    // we do not include Y2 as discussed below
    float P = (Y-Y1)/(Y2-Y1);
    float X0 = X1 + P*(X2-X1);
    int X = round(X0);
    output point (X,Y);
}
```

The operation of this algorithm is illustrated in Figure 10.7, where we show a raster with two endpoints and then show how the DDA algorithm populates the scanlines between the endpoints. Note that a roundoff that rounds an \(X\)-value upwards will give you a pixel to the right of the actual line, but this is consistent with the relation between the pixels and the line given by the endpoints.
Now when you scan-convert a complete polygon, you begin by scan-converting all the line
segments that make up its boundaries. The boundary pixels are not written immediately to the
frame buffer, however. You must save these pixels in some sort of pixel array from which you
can later get them for further processing into the fragments that make up the polygon. A good
mental model is to think of a 2D array, with one index being the scanline and the other representing
the two pixels you might have for each scanline (recalling that for a convex polygon, each scanline
will meet the polygon in zero or two points). You will write each pixel to the appropriate array,
and as you do so you will sort each scanline’s 2D array by the $X$-value of the pixels. Each of these
2D arrays, then, represents a fragment—a line segment within the polygon having a constant
scanline value—for the polygon.

There are some details of handling these scanline fragments that you need to understand, because
the process as we have defined it so far includes some ambiguous points. For example, we have
not talked about the “fragment” you would get at the highest or lowest vertex in the polygon,
where you would have the same pixel included twice. We have also not talked about the relation
between this polygon and others with which it might share an edge; that edge should be part of
one, but not both, of the polygons. If we would include it in both polygons, then the image we get
would depend on the order in which the polygons are drawn, which would be a problem. In order
to address these problems, we need to introduce a couple of conventions to creating fragments.
First, we will assume that we will include a horizontal boundary fragment only if it represents the
bottom of the polygon instead of the top. This is easily handled by including every pixel except the
topmost pixel for any non-horizontal boundary. Second, we will include any left-hand boundary
in a polygon but no right-hand boundary. This is also easily handled by defining the fragment for
each scanline to include all pixels from, and including, the left-hand pixel up to, but not including,
the right-hand pixel. Finally, we are handling all scanlines as fragments, so we do not process any
horizontal edge for any polygon.
With the algorithm and conventions defined above, we are able to take any convex polygon and produce the set of fragments that present that polygon in the frame buffer. As we interpolate the vertices across scanlines, however, there are other things defined for each pixel that must also be interpolated, including color, depth, and texture coordinates. Some of these properties, such as color, are independent of the pixel depth, but others, such as texture coordinates and depth itself, are not. Any depth-independent property can be treated simply by interpolating it along each edge to get values for the endpoints of each fragment, and then interpolating it along the fragment when that is processed. This interpolation can be computed in exactly the same way as the DDA or Bresenham algorithms interpolate geometry. But for depth-dependent properties, we do not have a linear relationship between the property and the pixel in the plane and ordinary linear interpolation can fail, as shown in Figure 10.8 (taken from the chapter on texture mapping, because textures can be very depth-dependent).

As we interpolate linearly across the pixel coordinates in screen space, the actual points on the line segment that correspond to these pixels are not themselves distributed linearly, as we see in Figure 10.9, where the space between points at the top of the actual line segment is much larger than the space between points at the bottom. We must use a perspective correction in order to reconstruct the actual point in 3D eye space. Recalling from Chapter 2 that the perspective projection gets the
2D eye space coordinates of a vertex by dividing by the vertex’s \( z \)-value, we need the actual depth value \( z \) to compute the original vertex coordinates. Once we calculate—or estimate, which is all we can do because of the aliased nature of pixel coordinates—the original depth value, we can estimate the original vertex coordinates and then use simple geometric principles to estimate the actual texture coordinates.

To interpolate the \( z \)-values, we must recognize that we are interpolating points that have been transformed by a perspective transformation. If the original point in 3D eye space is \((x, y, z)\) and the point in 2D eye space is \((X, Y)\), that transformation is given by \(X = x/z\) and \(Y = y/z\). Now if we are interpolating, say, \(X1\) and \(X2\), then we are interpolating \(x1/z1\) and \(x2/z2\), so that to interpolate the key values, we must interpolate \(1/z1\) and \(1/z2\) to get our estimate of \(z\) for our interpolated point. If we consider the point \(X = (1–t)*X1+t*X2\), then the corresponding \(z\) would be \((1–t)/z1 + t/z2 = ( (1–t)*z2+t*z1 )/(z1*z2)\). We would then reconstruct the original point in 3D eye space by multiplying the \(X\) and \(Y\) values of the interpolated point by this estimated \(z\) value.

We have discussed scanline interpolation in terms of a very simple, but not very fast, algorithm. In most cases, the scanline interpolation is done by a somewhat more sophisticated process. One such process is the Bresenham algorithm. This depends on managing an error value and deciding what pixels to light based on that value. For the usual basic case, we assume that the line we are interpolating has a slope no larger than one, and that the left-hand vertex is the \((0, 0)\) pixel. Such lines are said to lie in the first octant of the plane in standard position. For such a line, we will set a single pixel for each value of \(X\) in screen coordinates, and the question for any other pixel is simply whether the new pixel will be alongside the previous pixel (have the same \(Y\) value) or one unit higher than the previous pixel, and this is what the Bresenham algorithm decides.

The algorithm takes as input two vertices, \((X0, Y0)\) and \((X1, Y1)\) with \(X0 < X1\) and \(Y0 < Y1\), and we compute the two total distance terms \(DX = (X1 – X0)\) and \(DY = (Y1 – Y0)\). We want to set up a simple way to decide for any value of \(X\) whether the value of \(Y\) for that \(X\) is the same as the value of \(Y\) for \(X–1\) or one larger than the value of \(Y\) at \(X–1\).

We begin with the first vertex, which we be at the lower left of the space the line segment will live in, and we ask where the pixel will be for \(X0+1\). In the leftmost part of Figure 10.9 we see the setup for this question, which really asks whether the actual line will have a \(Y\)-value larger than \(Y0+.5\) for \(X=X0+1\). This can be rephrased as asking whether \((DY/DX) > 1/2\), or whether \(2*DY>DX\). This gives us an initial decision term \(P = 2*DY–DX\), along with the decision logic that says that \(Y\) increases by one if \(P>0\) and does not increase if \(P<0\).

![Figure 10.9: moving from one pixel to the next and considering the decision term](image)

With the first new vertex of the line settled, let’s now look at the second vertex. If we did not change \(Y\) for the first vertex, we find ourselves in the situation of the middle part of Figure 10.9. In this case, the decision for the second vertex is whether \(2*(DY/DX) > 1/2\). We calculate this out quickly as \(4*DY > DX\), or \(4*DY – DX > 0\). But this decision term can be written in terms of the previous decision term \(P\) as \(P + 2*DY > 0\). This case is, in fact, general and so we name the update term \(C1 = 2*DY\) and write the general operation: if \(P < 0\), then we create a new value of the decision variable \(P\) by \(P = P + C1\).
But if we did change $Y$ for the first vertex, we find ourselves in the situation of the right-hand part of Figure 10.9. Here the decision for the second vertex is whether $2*(DY/DX) - 1 > 1/2$. We again calculate this and get $2*DY > 3*DX/2$, or $4*DY - 3*DX > 0$. But again using the previous value of the decision term, we see that we now have $P + 2*(DY - DX) > 0$. Again, this case is general and we name the update term $C_2 = 2*(DY - DX)$ and write the general operation: if $P > 0$, then we create a new value of the decision variable by $P = P + C_2$.

The process of defining an initial value of the decision variable, making a decision about the next pixel, and then updating the decision variable depending on the last decision, is then carried out from the first pixel in the line to the last. Written as a code fragment, this first-octant version of the Bresenham algorithm implements the discussion above and looks like this:

```plaintext
bx = x0;
by = y0;
dx = (x1 - x0);
dy = (y1 - y0);
p = 2 * dy - dx;
c1 = 2 * dy;
c2 = 2 * (dy - dx);
while ( bx != x1 ){  
    bx = bx + 1;
    if ( p < 0 ) p = p + c1; // no change in by
    else {
        p = p + c2;
        by = by + 1;
    }
    setpixel( bx, by);
}
```

This algorithm can readily be made to interpolate not only lines in the first octant but lines in any direction, and can be readily adapted to interpolate also any property that is not depth dependent or, if you choose the property to be the reciprocal of the depth, can interpolate these values so you can approximate the actual depth of the pixel.

Some 3D Viewing Operations with Graphics Cards

Besides the techniques for creating 3D views presented earlier, there is a technique that involves preparing images so that they can be picked up and displayed by special hardware such as the CrystalEyes glasses from StereoGraphics Corp. This technology presents images alternately to the left and right eyes so the viewer sees the two images as though they were two views of the same scene. There are a variety of ways that a left-eye and right-eye image can be combined so they can be picked up by special display hardware; these include side-by-side images, above-and-below images, or interlaced images. These combinations may require some distortion of the images that will have to be handled by the display hardware, as suggested by the distortions in the images in Figure 1.11 below, and the video stream may have to be manipulated in other ways to accommodate these approaches. In these cases, the display hardware manipulates the video output stream, separating the stream into two images that are displayed in synchronization with alternating polarized blanking of one eye, allowing the two eyes to see two distinct images and thus see the stereo pair naturally.

These images are rasterized differently than the usual process described above. Each of the two images is sent to a different color buffer by doubling each pixel. In the case of the side-by-side images, the pixels are doubled horizontally and the buffers are changed when the middle band is reached. In the case of the above-below images, the pixels are doubled vertically and the buffers are again changed when the middle band is reached. This allows the system to maintain two
separate images from one original, creating the dual images needed by the alternating presentation hardware.

Figure 1.11: side-by-side images and above-below images, respectively.

Summary

This chapter has described the processing needed to create an image in an output color buffer (a raster of pixels) from the geometry and appearance information you would have in an ordinary graphics program with most graphics APIs. With this you can understand the amount and kind of processing needed to create a graphics image so you will understand why some kinds of images take longer to create than others, and are prepared to consider some of the techniques described in a later chapter to create higher-performance graphics programs. You will also have some background in case you want to undertake a more detailed study of computer graphics later in your education.

Questions

1. Draw an image containing only a small triangle with different colors at each vertex and smooth shading. Use a screen-capture utility for your computer to grab the screen with this image showing, and open the screen-grabbed file with a program such as Photoshop that allows you to greatly enlarge an image. Enlarge the image and look at the triangle; identify the scanlines on the triangle and note the fragments that make up the triangle across the scanlines. How do the fragments change as you move from the top to the bottom of the triangle?

2. Manually create a short fragments (perhaps 10 pixels long) with depth and color (including alpha) information for each pixel, and manually create a scanline with its own set of depth and color information. Manually walk through the operation of placing the fragment on the
scanline, using both the depth and color buffers. Show how the depth buffer is used to choose whether a fragment pixel will be used, and if it is, show how the color buffer and the pixel’s alpha value determine the color the scanline pixel will become. Assume whatever blending function you like.

**Exercises**

X.

**Experiments**

X. ... a set of experiments on timing that will point out how certain operations take more time than others ...