Viewing

Prerequisites

An understanding of 2D and 3D geometry and familiarity with simple linear mappings.

Introduction

We have consistently emphasized 3D modeling in these notes, because we believe that computer graphics should be encountered through 3D processes. But all of the viewing technologies that are readily available to us are 2D — monitors, printers, video, and film — and eventually even the visual cortex of our eyes is a 2D environment. So in order to present the images of the scenes we define with our modeling, we must create a 2D representation of the 3D scenes. This is what the viewing process is all about.

The fundamental point of viewing is to define a plane within 3-space, define a mapping that projects the model into that plane, and displays that plane in a given space on the viewing surface (we will usually think of a screen, but it could be a page or a number of other spaces). OpenGL defines this plane as the X-Y plane and projects the model onto this plane along the Z-axis. Further, as you carry out the projection noted above, you must create the 2D view that would be seen from a particular point in space, the point where the viewer’s eye would be. This sounds more difficult than it really is in practice, because a graphics API such as OpenGL has a number of tools that let you do this without working through all the geometry that this definition implies. All OpenGL needs is that you specify a few simple parameters in standard functions. We will describe the general concepts of viewing below and will then tell you how to specify the various parts of this process to OpenGL.

Finally, it is sometimes useful to “cut away” part of an image so you can see things that would otherwise be hidden behind some objects in a scene. We include a brief discussion of clipping planes, a technique for accomplishing this action.

Definitions

There are a small number of things that you must consider when thinking of how you will view your scene. These are independent of the particular API or other graphics tools you are using, but we will couple our discussion of these points with a discussion of how they are handled in OpenGL. The things are:

• Your world must be seen, so you need to say how the view is integrated into your model.
• In general, your world must be seen on a 2D surface such as a screen or a sheet of paper, so you must define how the 3D world is projected into a 2D space
• When your world is seen on the 3D surface, it must be seen at a particular place, so you must define the location where it will be seen.

These three things are called setting up your viewing environment, defining your projection, and defining your viewport, respectively.

Setting up the viewing environment: in order to set up a view, you have to put your eye in the geometric world where you do your modeling. This world is defined by the coordinate space you assumed when you modeled your scene as discussed earlier. Within that world, you define three critical components for your eye setup: where your eye is located, what point your eye is looking towards, and what direction is vertical with respect to your eye. These are defined in an OpenGL function and define the way the OpenGL system modifies the geometry in your modeling to create the view as it would be seen with the environment that you defined.
Projections: when you (or a camera) view something in the real world, everything you see is the result of light that comes to the retina (or the film) through a lens that focuses the light rays onto that viewing surface. This project is a projection, and computer graphics uses similar processes to create images. The projections from the natural world have all the light passing through the lens of the eye (or camera), essentially a single point; these projections have the property that parallel lines going off to infinity seem to converge at the horizon, and things in the distance are seen as smaller than the same things if they are up close. This is accomplished by a process called perspective. So projections where everything is seen by being projected onto a viewing plane through or towards a single point are called perspective projections. Standard graphics references will show diagrams that illustrate objects projected to the viewing plane through the center of view. This is illustrated by the left-hand image in Figure 4.1.

On the other hand, there are sometimes situations where you want to have everything of the same size show up as the same size on the image. This is most common where you need to take careful measurements from the image, as in engineering drawings. Parallel projections accomplish this by projecting all the objects in the scene to the viewing plane by parallel lines. Every graphics text and reference will contain a standard diagram showing how objects are projected by parallel lines to the viewing plane. This is illustrated in the right-hand image of Figure 4.1.

![Figure 4.1: perspective image (left) and orthographic image (right)](image)

Note that the differences between these two images in Figure 4.1 is small, but you should try out the sample code and move around the houses to see how the differences work in different views.

A projection is often thought of in terms of its view volume, the region of space that is visible in the projection. When we talk about these projections below, we will include figures that show what the view volume is for each.

Defining the viewport: We usually think first of a window when we do graphics on a screen. A window is a rectangular region in your viewing space and all your graphics will be seen in that window. If you use the GLUT toolkit, this is usually the kind of window you are accustomed to seeing in a current computing system — a rectangular space that carries a title bar and can be moved around on the screen and reshaped. This is the space in which all your graphical image will be seen.

Within the window, you can choose the part where your image is presented, and this part is called a viewport. A viewport is a rectangular region within that window to which you can restrict your image drawing. So a window can have many viewports, and each viewport can have its own image. The default behavior of OpenGL and of most graphics systems is to use the entire window for the viewport.
What this means: Any graphics system will have its conventions for carrying out the computations that are required to take your model and transform it to decide what pixels to set on the screen to make your image. OpenGL takes the model of taking your viewing environment information and transforming all the geometry in your scene (and we do mean all the geometry, as we will see in later chapters) to place your eye point at the origin, looking in the negative direction along the Z-axis, as is shown in Figure 4.2. The projection then determines how the transformed geometry will be mapped to a viewing plane, and these processes are illustrated later in this chapter. Finally, the viewing plane is mapped to the viewport you have defined in your window, and you will have the image you defined.

Figure 4.2: the standard OpenGL viewing model

In the end, we can think of the viewing process in terms of a rectangular hole cut out of a piece of cardboard and held in front of your eye. You can move yourself around in the world and can place the cardboard in any orientation you wish, and this will set your viewing environment; you can only see the world as it is projected in perspective through the hole, so you only have a perspective projection instead of an orthogonal projection; and you will hold the cardboard nearer or farther from your eye, and this will define your viewport and affect how much of the world is visible through the hole. You could probably put some kind of string over the hole and create a smaller viewport in the window presented by the hole, but this would be a little clumsy. Still, this model of viewing is a good place to start in understanding how this works.

Once you have defined the basic features for viewing your model, there are a number of other things you can consider that affect how the image is created and presented. We will talk about many of these over the next few chapters, but here we talk about hidden surfaces, clipping planes, and double buffering.

Hidden surfaces: another important question in creating an image is whether each new object should be fully displayed when it is called up by the imaging functions, or whether the geometry of the scene should be used to decide what objects are in front of other objects and to only draw the part of the objects that are in front. This is managed by the depth buffer in OpenGL, which uses a technique called Z-buffering. This buffer holds the z-value of the current value each pixel from the eye point in standard position. After the modelview transformation is applied, the entire geometry of the scene is transformed into a standard position with the eye at the origin and looking in the negative z-direction. As part of this, each vertex of the scene is also transformed; in particular, each vertex of each polygon takes on a new set of coordinates with a new z-value that represents the distance of the point from the eye in the z-direction.
When the polygon is processed by the graphics pipeline, an interpolation process is applied as described in the interpolation discussion in the chapter on the pipeline. This process will define a z-value, which is also the distance of that point from the eye in the z-direction, for each pixel in the polygon as it is processed. This allows a comparison of the z-value of the pixel to be plotted with the z-value that is currently held in the depth buffer. When a new point is to be plotted, the system first makes this comparison to check whether the new pixel is closer to the viewer than the current pixel in the image buffer and if it is, replaces the current point by the new point. This is a straightforward technique that can be managed in hardware by a graphics board or in software by simple data structures. There is a subtlety in this process that should be understood, however. Because it is more efficient to compare integers than floating-point numbers, the depth values in the buffer are kept as unsigned integers, scaled to fit the range between the near and far planes of the viewing volume with 0 as the front plane. If the near and far planes are far apart you may experience a phenomenon called “Z-fighting” in which roundoff errors when floating-point numbers are converted to integers causes the depth buffer shows inconsistent values for things that are supposed to be at equal distances from the eye. This problem is best controlled by trying to fit the near and far planes of the view as closely as possible to the actual items being displayed.

**Double buffering:** As you specify geometry in your program, it is entered into the viewing process and after all the viewing operations are finished, it causes pixels to be changed on the screen. However, it can take a moment to do all the work to create an image, and during this time you can be actually watching the pixels changing. If you were trying to create an animated image, drawing one image and then another, this would be disconcerting because you would constantly see your image being drawn once and then destroyed and re-drawn over and over again. Most graphics systems provide a facility called double buffering, in which you can draw to a background frame buffer and then swap buffers when it is completed to display the finished image to your viewer.

**Clipping planes:** Another part of defining a view is, in some cases, defining parts of the scene that you do not want to display — parts that are to be left out for some reason or another. (Note that OpenGL already leaves out the space to the left, right, above, below, in front, and behind the viewing volume, so some clipping is always being done. See the discussion on defining the projections below.) One important reason might be to see what is inside an object instead of just seeing the object’s surface. OpenGL allows you to define clipping planes in order to choose to display only the part of an object on one side or another of the plane. It’s fairly straightforward to define and use clipping planes, but you must define the plane in terms of its equation: \( Ax + By + Cz + D = 0 \), as defined in the first chapter above. The part of the space where \( Ax + By + Cz + D > 0 \) is actually displayed; the part where this expression is negative is clipped away. When you define the clipping plane to OpenGL, you will use the four coefficients of the equation above.

**Some examples**

The function below captures much of the code needed in the discussion that follows in this section.

```c
void reshape(int w, int h) {
    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,1.0,1.0,30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //           eye point     center of view     up
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```
Defining a window and viewport: The window was defined in the Getting Started module by the functions that initialize the window size and location and create the window.

```c
glutInitWindowSize(500,500);
glutInitWindowPosition(70,70);
glutCreateWindow("Your window name here");
```

The viewport is defined by the `glViewport` function that specifies the lower left coordinates and the upper right coordinates for the portion of the window that will be used by the display. This function will normally be used in your initialization function for the program.

```c
glViewport(VPLowerLeftX,VPLowerLeftY,VPUpperRightX,VPUpperRightY);
```

You can see the use of the viewport in the stereo viewing example below to create two separate images within one window.

Defining a viewing environment: To define what is usually called the viewing projection, you must first ensure that you are working with the `GL_MODELVIEW` matrix, then setting that matrix to be the identity, and finally define the viewing environment by specifying two points and one vector. The points are the eye point and the center of view (the point you are looking at), and the vector is the up vector — a vector that will be projected to define the vertical direction in your image. The only restrictions are that the eye point and center of view must be different, and the up vector must not be parallel to the vector from the eye point to the center of view.

```c
glmMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// eye point    center of view    up
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

Note that the `gluLookAt` function need not be invoked from the `reshape` function; it may be put inside the `display` function and variables may be used as needed to define the environment. See the stereo view discussion below for an idea of what that can do.

Defining perspective projection: a perspective projection is defined by first specifying that you want to work on the `GL_PROJECTION` matrix, and then setting that matrix to be the identity. You then specify the properties that will define the perspective transformation. In order, these are the field of view (an angle, in degrees, that defines the width of your viewing area), the aspect ratio (a ratio of width to height in the view; if the window is square this will probably be 1.0 but if it is not square, the aspect ratio will probably be the same as the ratio of the window width to height), the zNear value (the distance from the viewer to the plane that will contain the nearest points that can be displayed), and the zFar value (the distance from the viewer to the plane that will contain the farthest points that can be displayed). This sounds a little complicated, but once you’ve set it up a couple of times you’ll find that it’s very simple. It can be interesting to vary the field of view, though, to see the effect on the image.

```c
glmMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,30.0);
```

It is also possible to define your perspective projection by using the `glFrustum` function that defines the projection in terms of the volume containing the visible items, as shown in Figure 4.3. However, the `gluPerspective` function is so natural that we’ll leave the other approach to the student who wants it.

Defining an orthogonal projection: an orthogonal projection is defined much like a perspective projection except that the parameters of the projection itself are different. As you can see in the illustration of a parallel projection in Figure 4.3, the visible objects lie in a box whose sides are parallel to the X-, Y-, and Z-axes in the viewing space. Thus to define the viewing box for an orthogonal projection, we simply define the boundaries of the box as shown in Figure 4.3 and the OpenGL system does the rest.

```c
glOrtho(xLow,xHigh,yLow,yHigh,zNear,zFar);
```

There is no alternate to this function in the way that the `glFrustum(...)` is an alternative to the `gluLookAt(...)` function for parallel projections.
With either perspective or parallel projection, the definition of the projection includes a set of boundaries for the left and right sides, top and bottom sides, and front and back sides of a space. Only objects that are in this space, called the view volume, will be displayed; anything else will be clipped and be invisible.

![View volumes for perspective (left) and orthogonal (right) projections](image)

**Figure 4.3**: the view volumes for the perspective (left) and orthogonal (right) projections

Managing hidden surface viewing: in the Getting Started module when we introduced the structure of a program that uses OpenGL, we saw the glutInitDisplayMode function, called from main, as a way to define properties of the display. This function also allows the use of hidden surfaces if you specify GLUT_DEPTH as one of its parameters.

```c
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

You must also enable the depth test. Enabling is a standard property of OpenGL; many capabilities of the system are only available after they are enabled through the glEnable function, as shown below.

```c
    glEnable(GL_DEPTH_TEST);
```

From that point the depth buffer is in use and you need not be concerned about hidden surfaces. If you want to turn off the depth test, there is a glDisable function as well. Note the use of these two functions in enabling and disabling the clipping plane in the stereoView.c example code.

Setting double buffering: double buffering is a standard facility, and you will note that the function above that initializes the display mode includes a parameter GLUT_DOUBLE to set up double buffering. In your display() function, you will call glutSwapBuffers() when you have finished creating the image, and that will cause the background buffer to be swapped with the foreground buffer and your new image will be displayed.

Defining clipping planes: OpenGL allows you to define up to six clipping planes of your own. These are named GL_CLIP_PLANE0 through GL_CLIP_PLANE5, and are defined by an OpenGL function that takes a clipping plane name and a vector <A,B,C,D> as parameters. They are then enabled or disabled as needed to take effect in the scene. Specifically, some example code looks like

```c
    GLfloat myClipPlane[] = { 1.0, 1.0, 0.0, -1.0 };
    glClipPlane(GL_CLIP_PLANE0, myClipPlane);
    glEnable(GL_CLIP_PLANE0);
    ...
    glDisable(GL_CLIP_PLANE0);
```

and the stereo viewing example with this module includes the definition and use of clipping planes.

**Stereo viewing**

Stereo viewing gives us an opportunity to see some of these viewing processes in action. Let us say quickly that this should not be your first goal in creating images; it requires a bit of experience.
with the basics of viewing before it makes sense. Here we describe binocular viewing — viewing that requires you to converge your eyes beyond the computer screen or printed image, but that gives you the full effect of 3D when the images are converged. Other techniques are described in later chapters.

Stereo viewing is a matter of developing two views of a model from two viewpoints that represent the positions of a person’s eyes, and then presenting those views in a way that the eyes can see individually and resolve into a single image. This may be done in many ways, including creating two individual printed or photographed images that are assembled into a single image for a viewing system such as a stereopticon or a stereo slide viewer. (If you have a stereopticon, it can be very interesting to use modern technology to create the images for this antique viewing system!) However, it is also possible to present these as two viewports in a single window on the screen so the two images may be manipulated together while viewer resolves these into a single image by focusing each eye on a separate image.

This latter process is fairly simple. First, create a window that is twice as wide as it is high, and whose overall width is twice the distance between your eyes. Then when you display your model, do so twice, with two different viewports that occupy the left and right half of the window. Each display is identical except that the eye points in the left and right halves represent the position of the left and right eyes, respectively.

In the example code `stereoView.c`, you will find the window initialization function

```c
    glutInitWindowSize(600,300);
```

and the left- and right-hand viewports as

```c
    // left-hand viewport
    glViewport(0,0,300,300);
    ...
    //   eye point      center of view       up
    gluLookAt(ep-offset, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    ...
    // right-hand viewport
    glViewport(300,0,300,300);
    ...
    //   eye point      center of view       up
    gluLookAt(ep+offset, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
    ...
```

When you do this, you need to create a suitable value of offset that is about half the distance between the eyes of the observer (or perhaps a bit less, to help the viewer’s eyes converge). This makes it easier for each eye to focus on its individual image and let the brain’s convergence create the merged stereo image. The result can be quite startling if the model exaggerates the front-to-back differences in the view, or it can be more subtle if your goal is to represent realistic views. Figure 4.4 the effect of such stereo viewing as produced by the example code.

Many people cannot do the kind of eye convergence that stereo viewing requires. Some people have general convergence problems which do not allow the eyes to focus together on a image to create a merged image, and some simply cannot see beyond the screen to the point where convergence would occur. In addition, if you do not get the spacing of the stereo pair right, or have the sides misaligned, or allow the two sides to refresh at different times, or ... well, it can be difficult to get this to work well for users. If some of your users can see the converged image and some cannot, that’s probably as good as it’s going to be.

There are other techniques for doing 3D viewing. When we discuss texture maps later, we will describe a technique that colors 3D images more red in the near part and more blue in the distant part. This makes the images self-converge when you view them through a pair of ChromaDepth™ glasses, as we will describe there, so more people can see the spatial properties of the image, and it
can be seen from anywhere in a room. There are also more specialized techniques such as creating
alternating-eye views of the image on a screen with a overscreen that can be given alternating
polarization and viewing them through polarized glasses that allow each eye to see only one screen
at a time, or using dual-screen technologies such as head-mounted displays. The extension of the
techniques above to these more specialized technologies is straightforward and is left to your
instructor if such technologies are available.

Figure 4.4: A stereo pair, including a clipping plane

*Code examples*

The source codes for these examples may be found in the two files

```c
projections.c
stereoView.c
```

that are available online with this note.