Viewing and Projection

Prerequisites

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

Introduction

We emphasize 3D computer graphics consistently in these notes, because we believe that computer graphics should be encountered through 3D processes and that 2D graphics can be considered effectively as a special case of 3D graphics. But almost all of the viewing technologies that are readily available to us are 2D — certainly monitors, printers, video, and film — and eventually even the active visual retina of our eyes presents 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. As we saw in the graphics pipeline in the previous chapter, you begin by developing a set of models that make up the elements of your scene and set up the way the models are placed in the scene, resulting in a set of objects in a common world space. You then define the way the scene will be viewed and the way that view is presented on the screen. In this early chapter, we are concerned with the way we move from the world space to a 2D image with the tools of viewing and projection.

Let’s consider an example of a world space and look at just what it means to have a view and a presentation of that space. One of the author’s favorite places is Glacier Point in Yosemite National Park. Standing in an observation building on the point you can see up and down Yosemite Valley, with the Merced River falls and Half Dome in one direction and Yosemite Village and other park facilities in another direction. The photographs in Figure 1.1 below give you an idea of the views from this point.

Figure 1.1: two photographs of Yosemite Valley from Glacier Point

As you look around from this point and think about the view, you will notice that your view depends first on where you are standing. If you were standing in Yosemite Village on the valley floor, or at the top of Nevada Falls, you could not have this view. The view also depends on the direction in which you are looking. The two photographs in the figure above are taken from the same point, but show very different views. Finally, although this may not be obvious at first because our minds process images in context, the view depends on whether you are standing with your head upright or tilted (this might be easier to grasp if you think of the view as being defined by a camera instead of by your vision; it’s clear that if you tilt a camera at a 45° angle you get a very different photo than one that’s taken by a horizontal or vertical camera.) The world is the same in any case, but the three facts of where your eye is, the direction you are facing, and the way your view is tilted, determine the view.

But the view, once determined, must now be translated into an image that can be presented on your computer monitor. You may think of this in terms of recording an image on a digital camera, because the result is the same: each point of the view space (each pixel in the image) must be given a specific color. Doing that with the digital camera involves only capturing the light that comes through the lens to that point in the camera’s sensing device, but doing it with computer graphics requires that we calculate exactly what will be seen at that particular point when the view is presented. We must define the way the scene is transformed into a two-dimensional space, which involves a number of steps: taking into account all the questions of what parts are in front of what other parts, what parts are out of view from the camera’s lens, and how the lens gathers light from
the scene to bring it into the camera. The best way to think about the lens is to compare two very different kinds of lenses: one is a wide-angle lens that gathers light in a very wide cone, and the other is a high-altitude photography lens that gathers light only in a very tight cylinder and processes light rays that are essentially parallel as they are transferred to the sensor. Finally, once the light from the continuous world comes into the camera, it is recorded on a digital sensor that only captures a discrete set of points.

This model of viewing is paralleled quite closely by a computer graphics system. You begin your work by modeling your scene in an overall world space (you may actually start in several modeling spaces, because you may model the geometry of each part of your scene in its own modeling space where it can be defined easily, then place each part within a single consistent world space to define the scene). This is very different from the viewing we discuss here but is covered in detail in the next chapter. The fundamental operation of viewing is to define an eye within your world space that represents the view you want to take of your modeling space. Defining the eye implies that you are defining a coordinate system relative to that eye position, and you must then transform your modeling space into a standard form relative to this coordinate system by defining, and applying, a viewing transformation. The fundamental operation of projection, in turn, 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, a video frame, or a number of other spaces).

We will think of the 3D space we work in as the traditional X-Y-Z Cartesian coordinate space, usually with the X- and Y-axes in their familiar positions and with the Z-axis coming toward the viewer from the X-Y plane. This orientation is used because most graphics APIs define the plane onto which the image is projected for viewing as the X-Y plane, and project the model onto this plane in some fashion along the Z-axis. The mechanics of the modeling transformations, viewing transformation, and projection are managed by the graphics API, and the task of the graphics programmer is to provide the API with the correct information and call the API functionality in the correct order to make these operations work. We will describe the general concepts of viewing and projection 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.

Fundamental model of viewing

As a physical model, we can think of the viewing process in terms of looking through 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, setting your eye into whatever position and orientation from you wish to see the scene. This defines your view. Once you have set your position in the world, you can hold up the cardboard to your eye and this will set your projection; by changing the distance of the cardboard from the eye you change the viewing angle for the projection. Between these two operations you define how you see the world in perspective through the hole. Of course, you only have a perspective projection instead of an orthogonal projection, and you see through the entire hole instead of through only a part of it, but this model of viewing is a good place to start in understanding how viewing and projection work.

As we noted above, the goal of the viewing process is to rearrange the world so it looks as it would if the viewer’s eye were in a standard position, depending on the API’s basic model. When we define the eye location, we give the API the information it needs to do this rearrangement. In the next chapter on modeling, we will introduce the important concept of the scene graph, which will integrate viewing and modeling. Here we give an overview of the viewing part of the scene graph.
The key point is that your view is defined by the location, direction, and orientation of the eye as we noted above. There are many ways to create this definition, but the effect of each is to give the transformation needed to place the eye at its desired location and orientation, which we will assume to be at the origin, looking in the negative direction down the Z-axis. To put the eye into this standard position we compute a new coordinate system for the world by applying what is called the viewing transformation. The viewing transformation is created by computing the inverse of the transformation that placed the eye into the world. (If the concept of computing the inverse seems difficult, simply think of undoing each of the pieces of the transformation; we will discuss this more in the chapter on modeling). Once the eye is in standard position, and all your geometry is adjusted in the same way, the system can easily move on to project the geometry onto the viewing plane so the view can be presented to the user.

Once you have organized the view in this way, you must organize the information you send to the graphics system to create your scene. The graphics system provides some assistance with this by providing tools that determine just what will be visible in your scene and that allow you to develop a scene but only present it to your viewer when it is completed. These will also be discussed in this chapter.

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 later in the chapter 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 defined in 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 window and 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. When these are defined to your graphics API, the geometry in your modeling is adjusted to create the view as it would be seen with the environment that you defined. This is discussed in the section below on the fundamental model of viewing.

Projections: When you define a scene, you will want to do your work in the most natural world that would contain the scene, which we called the model space in the graphics pipeline discussion of the previous chapter. For most of these notes, that will mean a three-dimensional world that fits the objects you are developing. But you will probably want to display that world on a two-dimensional space such as a computer monitor, a video screen, or a sheet of paper. In order to move from the three-dimensional world to a two-dimensional world we use a projection operation.

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 process is a projection of the natural (3D) world onto a two-dimensional space. These projections in the natural world operate when light passes through the lens of the eye (or camera), essentially a single point, and have the property that parallel lines going off to infinity...
seem to converge at the horizon so things in the distance are seen as smaller than the same things when they are close to the viewer. This kind of projection, where everything is seen by being projected onto a viewing plane through or towards a single point, is called a perspective projection. Standard graphics references show diagrams that illustrate objects projected to the viewing plane through the center of view; the effect is that an object farther from the eye are seen as smaller in the projection than the same object closer to the eye.

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. For parallel projections, objects that are the same size are seen in the projection with the same size, no matter how far they are from the eye. Standard graphics texts contain diagrams showing how objects are projected by parallel lines to the viewing plane.

In Figure 1.2 below we show two images of a wireframe house from the same viewpoint. The left-hand image of the figure is presented with a perspective projection, as shown by the difference in the apparent sizes of the front and back ends of the building, and by the way that the lines outlining the sides and roof of the building get closer as they recede from the viewer. The right-hand image of the figure is shown with a parallel or orthogonal projection, as shown by the equal sizes of the front and back ends of the building and the parallel lines outlining the sides and roof of the building. The differences between these two images is admittedly small, but you should use both projections on some of your scenes and compare the results to see how the differences work in different views.

![Figure 1.2: perspective image (left) and orthographic image (right)](image)

A projection is often thought of in terms of its view volume, the region of space that is visible in the projection. With either perspective or parallel projection, the definition of the projection implicitly defines a set of boundaries for the left and right sides, top and bottom sides, and front and back sides of a region in three-dimensional space that is called the *viewing volume* for the projection. The viewing volumes for the perspective and orthogonal projections are shown in Figure 1.3 below. Only objects that are inside this space will be displayed; anything else in the scene will be clipped and be invisible.
While the parallel view volume is defined only in a specified place in your model space, the orthogonal view volume may be defined wherever you need it because, being independent of the calculation that makes the world appear from a particular point of view, an orthogonal view can take in any part of space. This allows you to set up an orthogonal view of any part of your space, or to move your view volume around to view any part of your model.

Defining the window and viewport: We usually think first of a window when we do graphics on a screen. A window in the graphics sense is a rectangular region in your viewing space in which all of the drawing from your program will be done, usually defined in terms of the physical units of the drawing space. The space in which you define and manage your graphics windows will be called screen space here for convenience, and is identified with integer coordinates. The smallest displayed unit in this space will be called a pixel, a shorthand for picture element. Note that the window for drawing is a distinct concept from the window in a desktop display window system, although the drawing window may in fact occupy a window on the desktop; we will be consistently careful to reserve the term window for the graphic display.

A single program can manage several different windows at once, drawing to each as needed for the task at hand. Window management can be a significant problem, but most graphics APIs have tools to manage this with little effort on the programmer’s part, producing 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. Of course, other graphical outputs such as video will handle windows differently, usually treating the entire output frame as a single window without any title or border.

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. In any window or viewport, the ratio of its width to its height is called its aspect ratio. A window can have many viewports, even overlapping if needed to manage the effect you need, and each viewport can have its own image. The default behavior of most graphics systems is to use the entire window for the viewport. A viewport is usually defined in the same terms as the window it occupies, so if the window is specified in terms of physical units, the viewport probably will be also. However, a viewport can also be defined in terms of its size relative to the window.

If your graphics window is presented in a windowed desktop system, you may want to be able to manipulate your graphics window in the same way you would any other window on the desktop. You may want to move it, change its size, and click on it to bring it to the front if another window
has been previously chosen as the top window. This kind of window management is provided by the graphics API in order to make the graphics window compatible with all the other kinds of windows available.

When you manipulate the desktop window containing the graphics window, the contents of the window need to be managed to maintain a consistent view. The graphics API tools will give you the ability to manage the aspect ratio of your viewports and to place your viewports appropriately within your window when that window is changed. If you allow the aspect ratio of a new viewport to be different than it was when defined, you will see that the image in the viewport seems distorted, because the program is trying to draw to the originally-defined viewport.

What this means: Any graphics system will have its approach to defining the computations that transform your geometric model as if it were defined in a standard position and then project it to compute the points to set on the viewing plane to make your image. Each graphics API has its basic concept of this standard position and its tools to create the transformation of your geometry so it can be viewed correctly. For example, OpenGL defines its viewing to take place in a left-handed coordinate system (while all its modeling is done in a right-handed system) and transforms 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. The eye-space orientation is illustrated in Figure 1.4. The projection then determines how the transformed geometry will be mapped to the X-Y 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 have the image you defined.

![Figure 1.4: the standard OpenGL viewing model](image)

Of course, no graphics API assumes that you can only look at your scenes with this standard view definition. Instead, you are given a way to specify your view very generally, and the API will convert the geometry of the scene so it is presented with your eyepoint in this standard position. This conversion is accomplished through a viewing transformation that is defined from your view definition.

The information needed to define your view includes your eye position (its \((x, y, z)\) coordinates), the direction your eye is facing or the coordinates of a point toward which it is facing, and the direction your eye perceives as “up” in the world space. For example, the default view that we mention above has the position at the origin, or \((0, 0, 0)\), the view direction or the “look-at” point coordinates as \((0, 0, -1)\), and the up direction as \((0, 1, 0)\). You will probably want to identify a different eye position for most of your viewing, because this is very restrictive and you aren’t likely to want to define your whole viewable world as lying somewhere behind the X-Y plane, and so your graphics API will give you a function that allows you to set your eye point as you desire.
The viewing transformation, then, is the transformation that takes the scene as you define it in world space and aligns the eye position with the standard model, giving you the eye space we discussed in the previous chapter. The key actions that the viewing transformation accomplishes are to rotate the world to align your personal up direction with the direction of the Y-axis, to rotate it again to put the look-at direction in the direction of the negative Z-axis (or to put the look-at point in space so it has the same X- and Y-coordinates as the eye point and a Z-coordinate less than the Z-coordinate of the eye point), to translate the world so that the eye point lies at the origin, and finally to scale the world so that the look-at point or look-at vector has the value \((0, 0, -1)\). This is a very interesting transformation because what it \textit{really} does is to invert the set of transformations that would move the eye point from its standard position to the position you define with your API function as above. This is discussed in some depth later in this chapter in terms of defining the view environment for the OpenGL API.

\textit{Some aspects of managing the view}

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.

\textbf{Hidden surfaces:} Most of the things in our world are opaque, so we only see the things that are nearest to us as we look in any direction. This obvious observation can prove challenging for computer-generated images, however, because a graphics system simply draws what we tell it to draw in the order we tell it to draw them. In order to create images that have the simple “only show me what is nearest” property we must use appropriate tools in viewing our scene.

Most graphics systems have a technique that uses the geometry of the scene in order to decide what objects are in front of other objects, and can use this to draw only the part of the objects that are in front as the scene is developed. This technique is generally called Z-buffering because it uses information on the z-coordinates in the scene, as shown in Figure 1.4. In some systems it goes by other names; for example, in OpenGL this is called the \textit{depth buffer}. This buffer holds the z-value of the nearest item in the scene for each pixel in the scene, where the z-values are computed from the eye point in eye coordinates. This z-value is the depth value after the viewing transformation has been applied to the original model geometry.

This depth value is not merely computed for each vertex defined in the geometry of a scene. When a 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.
There are other techniques for ensuring that only the genuinely visible parts of a scene are presented to the viewer, however. If you can determine the depth (the distance from the eye) of each object in your model, then you may be able to sort a list of the objects so that you can draw them from back to front — that is, draw the farthest first and the nearest last. In doing this, you will replace anything that is hidden by other objects that are nearer, resulting in a scene that shows just the visible content. This is a classical technique called the painter’s algorithm (because it mimics the way a painter could create an image using opaque paints) that was widely used in more limited graphics systems, but it sometimes has real advantages over Z-buffering because it is faster (it doesn’t require the pixel depth comparison for every pixel that is drawn) and because sometimes Z-buffering will give incorrect images, as we discuss when we discuss modeling transparency with blending in the color chapter.

Double buffering: As you specify geometry in your program, the geometry is modified by the modeling and projection transformations and the piece of the image as you specified it is written into the color buffer. It is the color buffer that actually is written to the screen to create the image seen by the viewer. Most graphics systems offer you the capability of having two color buffers — one that is being displayed (called the front buffer) and one into which current graphics content is being written (called the back buffer). Using these two buffers is called double buffering.

Because it can take some time to do all the work to create an image, if you are using only the front buffer you may end up actually watching the pixels changing as the image is created. If you were trying to create an animated image by drawing one image and then another, it would be disconcerting to use only one buffer because you would constantly see your image being drawn and then destroyed and re-drawn. Thus double buffering is essential to animated images and, in fact, is used quite frequently for other graphics because it is more satisfactory to present a completed image instead of a developing image to a user. You must remember, however, that when an image is completed you must specify that the buffers are to be swapped, or the user will never see the new image!

Clipping planes: Clipping is the process of drawing with the portion of an image on one side of a plane drawn and the portion on the other side omitted. Recall from the discussion of geometric fundamentals that a plane is defined by a linear equation

\[ Ax + By + Cz + D = 0 \]

so it can be represented by the 4-tuple of real numbers \((A, B, C, D)\). The plane divides the space into two parts: that for which \(Ax+By+Cz+D\) is positive and that for which it is negative. When you define the clipping plane for your graphics API with the functions it provides, you will probably use the four coefficients of the equation above. The operation of the clipping process is that any points for which this value is negative will not be displayed; any points for which it is positive or zero will be displayed.

Clipping defines parts of the scene that you do not want to display — parts that are to be left out for any reason. Any projection operation automatically includes clipping, because it must leave out objects in the space to the left, right, above, below, in front, and behind the viewing volume. In effect, each of the planes bounding the viewing volume for the projection is also a clipping plane for the image. You may also want to define other clipping planes for an image. One important reason to include clipping might be to see what is inside an object instead of just seeing the object’s surface; you can define clipping planes that go through the object and display only the part of the object on one side or another of the plane. Your graphics API will probably allow you to define other clipping planes as well.

While the clipping process is handled for you by the graphics API, you should know something of the processes it uses. Because we generally think of graphics objects as built of polygons, the key point in clipping is to clip line segments (the boundaries of polygons) against the clipping plane.
As we noted above, you can tell what side of a plane contains a point \((x, y, z)\) by testing the algebraic sign of the expression \(Ax+By+Cz+D\). If this expression is negative for both endpoints of a line segment, the entire line must lie on the “wrong” side of the clipping plane and so is simply not drawn at all. If the expression is positive for both endpoints, the entire line must lie on the “right” side and is drawn. If the expression is positive for one endpoint and negative for the other, then you must find the point for which the equation \(Ax+By+Cz+D=0\) is satisfied and then draw the line segment from that point to the point whose value in the expression is positive. If the line segment is defined by a linear parametric equation, the equation becomes a linear equation in one variable and so is easy to solve.

In actual practice, there are often techniques for handling clipping that are even simpler than that described above. For example, you might make only one set of comparisons to establish the relationship between a vertex of an object and a set of clipping planes such as the boundaries of a standard viewing volume. You can then use these tests to drive a set of clipping operations. We leave the details to the standard literature on graphics techniques.

**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!) Later in this chapter we describe how to present these as two viewports in a single window on the screen with OpenGL.

When you set up two viewpoints in this fashion, you need to identify two eye points that are offset by a suitable value in a plane perpendicular to the up direction of your view. It is probably simplest is you define your up direction to be one axis (perhaps the z-axis) and your overall view to be aligned with one of the axes perpendicular to that (perhaps the x-axis). You can then define an offset that is about the distance between the eyes of the observer (or perhaps a bit less, to help the viewer’s eyes converge), and move each eyepoint from the overall viewpoint by half that offset. 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 eye offset is large so the pair exaggerates the front-to-back differences in the view, or it can be more subtle if you use modest offsets to represent realistic views. Figure 1.5 shows the effect of such stereo viewing with a full-color shaded model. Later we will consider how to set the stereo eyepoints in a more systematic fashion.

Many people have physical limitations to their eyes and cannot perform the kind of eye convergence that this kind of stereo viewing requires. Some people have general convergence problems which do not allow the eyes to focus together to create a merged image, and some simply cannot seem to 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.

![A stereo pair, including a clipping plane](image)

Figure 1.5: A stereo pair, including a clipping plane

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.

**Implementation of viewing and projection in OpenGL**

The OpenGL code below captures much of the code needed in the discussion that follows in this section. It could be taken from a single function or could be assembled from several functions; in the sample structure of an OpenGL program in the previous chapter we suggested that the viewing and projection operations be separated, with the first part being at the top of the `display()` function and the latter part being at the end of the `init()` and `reshape()` functions.

```c
// Define the projection for the scene
glViewport(0, 0, (GLsizei)w, (GLsizei)h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, (GLsizei)w/(GLsizei)h, 1.0, 30.0);

// Define the viewing environment for the scene
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 previous chapter by a set of functions that initialize the window size and location and create the window. The details of window management are intentionally hidden from the programmer so that an API can work across many different platforms. In OpenGL, it is easiest to delegate the window setup to the GLUT toolkit where much of the system-dependent parts of OpenGL are defined; the functions to do this are:
glutInitWindowSize(width, height);
grInitWindowPosition(topleftX, topleftY);
grCreateWindow("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.

`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.

Reshaping the window: The window is reshaped when it initially created or whenever is moved it to another place or made larger or smaller in any of its dimensions. These reshape operations are handled easily by OpenGL because the computer generates an event whenever any of these window reshapings happens, and there is an event callback for window reshaping. We will discuss events and event callbacks in more detail later, but the reshape callback is registered by the function `glutReshapeFunc(reshape)` which identifies a function `reshape(GLint w, GLint h)` that is to be executed whenever the window reshape event occurs and that is to do whatever is necessary to regenerate the image in the window.

The work that is done when a window is reshaped can involve defining the projection and the viewing environment, updating the definition of the viewport(s) in the window, or can delegate some of these to the display function. Any viewport needs either to be defined inside the reshape callback function so it can be redefined for resized windows or to be defined in the display function where the changed window dimensions can be taken into account when it is defined. The viewport probably should be designed directly in terms relative to the size or dimensions of the window, so the parameters of the reshape function should be used. For example, if the window is defined to have dimensions `(width, height)` as in the definition above, and if the viewport is to comprise the right-hand side of the window, then the viewport’s coordinates are

`(width/2, 0, width, height)`

and the aspect ratio of the window is `width/(2*height)`. If the window is resized, you will probably want to make the width of the viewport no larger than the larger of half the new window width (to preserve the concept of occupying only half of the window) or the new window height times the original aspect ratio. This kind of calculation will preserve the basic look of your images, even when the window is resized in ways that distort it far from its original shape.

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, 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. As we saw earlier, sample code to do this is:

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

The `gluLookAt(...)` function may be invoked from the reshape function, or it may be put inside the display function and variables may be used as needed to define the environment. In general, we will lean towards including the `gluLookAt` operation at the start of the `display` function, as we will discuss below. See the stereo view discussion below for an idea of what that can do.

The effect of the `gluLookAt(...)` function is to define a transformation that moves the eye point from its default position and orientation. That default position and orientation has the eye at the
origin and looking in the negative z-direction, and oriented with the y-axis pointing upwards. This is the same as if we invoked the \texttt{gluLookAt} function with the parameters 
\texttt{gluLookAt(0., 0., 0., 0., 0., -1., 0., 1., 0.).}

When we change from the default value to the general eye position and orientation, we define a set of transformations that give the eye point the position and orientation we define. The overall set of transformations supported by graphics APIs will be discussed in the modeling chapter, but those used for defining the eyepoint are:

1. a rotation about the Z-axis that aligns the Y-axis with the up vector,
2. a scaling to place the center of view at the correct distance along the negative Z-axis,
3. a translation that moves the center of view to the origin,
4. two rotations, about the X- and Y-axes, that position the eye point at the right point relative to the center of view, and
5. a translation that puts the center of view at the right position.

In order to get the effect you want on your overall scene, then, the viewing transformation must be the inverse of the transformation that placed the eye at the position you define, because it must act on all the geometry in your scene to return the eye to the default position and orientation. Because function inverses act by

\[(F*G)^{-1} = G^{-1}*F^{-1}\]

the viewing transformation is built by inverting each of these five transformations in the reverse order. And because this must be done on all the geometry in the scene, it must be applied last — so it must be specified before any of the geometry is defined. We will thus usually see the \texttt{gluLookAt(...) function} as one of the first things to appear in the \texttt{display()} function, and its operation is the same as applying the transformations

1. translate the center of view to the origin,
2. rotate about the X- and Y-axes to put the eye point on the positive Z-axis,
3. translate to put the eye point at the origin,
4. scale to put the center of view at the point (0.,0.,-1.), and
5. rotate around the Z-axis to restore the up vector to the Y-axis.

You may wonder why we are discussing at this point how the \texttt{gluLookAt(...)} function defines the viewing transformation that goes into the modelview matrix, but we will need to know about this when we need to control the eye point as part of our modeling in more advanced kinds of scenes.

Defining perspective projection: a perspective projection is defined by first specifying that you want to work on the \texttt{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.

\begin{verbatim}
gMatrixMode(GL_PROJECTION);
gLoadIdentity();
gluPerspective(60.0,1.0,1.0,30.0);
\end{verbatim}

It is also possible to define your perspective projection by using the \texttt{glFrustum} function that defines the projection in terms of the viewing volume containing the visible items, as was shown in Figure 1.2 above. However, the \texttt{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 1.3, 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 1.3 and the OpenGL system does the rest.

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

The viewing space is still the same left-handed space as noted earlier, so the zNear and zFar values are the distance from the X-Y plane in the negative direction, so that negative values of zNear and zFar refer to positions behind the eye (that is, in positive Z-space). There is no alternate to this function in the way that the glFrustum(...) is an alternative to the gluLookAt(...) function for parallel 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: In addition to the clipping OpenGL performs on the standard view volume in the projection operation, OpenGL allows you to define at least six clipping planes of your own, named GL_CLIP_PLANE0 through GL_CLIP_PLANE5. The clipping planes are defined by the function glClipPlane(plane, equation) where plane is one of the predefined clipping planes above and equation is a vector of four GLfloat values. Once you have defined a clipping plane, it is enabled or disabled by a glEnable(GL_CLIP_PLANEEn) function or equivalent glDisable(...) function. Clipping is performed when any modeling primitive is called when a clip plane is enabled; it is not performed when the clip plane is disabled. 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);
```

The stereo viewing example at the end of this chapter includes the definition and use of clipping planes.
**Implementing a stereo view**

In this section we describe the implementation of binocular viewing as described earlier in this chapter. The technique we will use is to generate two views of a single model as if they were seen from the viewer’s separate eyes, and present these in two viewports in a single window on the screen. These two images are then manipulated together by manipulating the model as a whole, 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. This can be done by creating a window with space for both viewports with the window initialization function

```c
#define W 600
#define H 300
width = W; height = H;
glutInitWindowSize(width,height);
```

Here the initial values set the width to twice the height, allowing each of the two viewports to be initially square. We set up the view with the overall view at a distance of $e_p$ from the origin in the $x$-direction and looking at the origin with the $z$-axis pointing up, and set the eyes to be at a given offset distance from the overall viewpoint in the $y$-direction. We then define the left- and right-hand viewports in the `display()` function as follows

```c
// left-hand viewport
glViewport(0,0,width/2,height);
...
//                eye point      center of view       up
gluLookAt(ep, -offset, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
... code for the actual image goes here
...
// right-hand viewport
glViewport(width/2,0,width/2,height);
...
//                eye point      center of view       up
gluLookAt(ep,  offset, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
... the same code as above for the actual image goes here
...
```

This particular code example responds to a `reshape(width,height)` operation because it uses the window dimensions to set the viewport sizes, but it is susceptible to distortion problems if the user does not maintain the 2:1 aspect ratio as he or she resizes the window. It is left to the student to work out how to create square viewports within the window if the window aspect ratio is changed.