Chapter 1: 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.

We set the scene for this process in the last chapter, when we defined the graphics pipeline. If we begin at the point where we have the 3D world coordinates—that is, where we have a complete scene fully defined in a 3D world—then this chapter is about creating a view of that scene in our display space of a computer monitor, a piece of film or video, or a printed page, whatever we want. To remind ourselves of the steps in this process, the pipeline is again shown in Figure 1.1.

![Figure 1.1: the graphics pipeline for creating an image of a scene](image)

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 Yosemite National Park, which is a wonderful example of a 3D world. Certainly there is a basic geometry in the park, made up of stone, wood, and water; this geometry can be seen from a number of points. In Figure 1.2 we see the classic piece of Yosemite geometry, the Half Dome monolith, from below in the valley and from above at Glacier Point. This gives us an excellent example of two views of the same geometry.

If you think about this area shown in these photographs, we can see the essential components of viewing. First, you notice that your view depends first on where you are standing. If you are standing on the valley floor, you see the face of the monolith in the classic view; if you are standing on the rim of Yosemite Valley at about the same height as Half Dome, you get a view that shows the profile of the rock. So your view depends on your position, which we call your eye point. Second, the view also depends on the point you are looking at, which we will call the view reference point. Both photos look generally towards the Half Dome monolith, or more specifically, towards a point in space behind the dome. This makes a difference not only in the view of the dome, but in the view of the region around the dome. In the classic Half Dome view from the valley, if you look off to the right you see the south wall of the valley; in the view from Glacier Point, if you look off to the right you see Vernal and Nevada falls on the Merced River.
and, farther to the right, the high Sierra in the south of the park. The view also depends on the breadth of field of your view, whether you are looking at a wide part of the scene or a narrow part; again, the photograph at the left is a view of just Half Dome, while the one at the right is a panoramic view that includes the dome. While both photos are essentially square, you can visualize the left-hand photo as part of a photo that’s vertical in layout while the right-hand photo looks more like it would come from a horizontal layout; this represents an aspect ratio for the image that can be part of its definition. Finally, although this may not be obvious at first because our minds process images in context, the view depends on your sense of the up direction in the scene: 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 determining factors for the image are where your eye is, the point you are looking toward, the breadth of your view, the aspect ratio of your view, and the way your view is tilted. All these will be accounted for when you define an image in computer graphics.

Once you have determined your view, it must then 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, and it follows the graphics pipeline that we discussed in the last chapter. 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-dimensional 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 is a right-handed coordinate system, so-called because if you orient your right hand with your fingers pointing from the X-axis towards the Y-axis, your thumb will point towards the Z-axis. This orientation is commonly used for modeling in computer graphics 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, because the system must clip away parts of the scene that are not visible in the final image.

*Fundamental model of viewing*

As a physical model, we can think of the viewing process in terms of looking through a rectangular frame that is 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 viewpoint and view reference point. The shape of the frame and the orientation you give it determine the aspect ratio and the up direction for the image. Once you have set your position in the world, you can hold up the frame to your eye and this will set your projection; by changing the distance of the frame from the eye you change the breadth of field for the projection. Between these two operations you define how you see the world in perspective through the hole. And finally, if you put a piece of transparent material that is ruled in very small squares behind the cardboard (instead of your eye) and you fill in each square to match the brightness you see in the square, you will create a copy of the image that you can take away from the original location. Of course, you only have a perspective projection instead of an orthogonal projection, 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, orientation, and field of view of the eye as we noted above. To understand this a little more fully, consider the situation shown in Figure 1.3. Here we have a world coordinate system that is oriented in the usual way, and within this world we have both a (simple) model and an eyepoint. At the eyepoint we have the coordinate system that is defined by the eyepoint-view reference point-up information that is specified for the view, so you may see the eyepoint coordinates in context. From this, you should try to visualize how the model will look once it is displayed with the view.

![Figure 1.3: the eye coordinate system within the world coordinate system](image)

In effect, you have defined a coordinate system within the world space relative to the eye. There are many ways to create this definition, but basically they all involve specifying three pieces of data in 3D space. Once this eye coordinate system is defined, we can apply an operation that changes the coordinates of everything in the world into equivalent representations in the eye coordinate system. This change of coordinates is a straightforward mathematical operation, performed by creating a change-of-basis matrix for the new system and then applying it to the world-space geometry. The transformation places the eye at the origin, looking along the Z-axis, and with the Y-axis pointed upwards; this view is similar to that shown in Figure 1.4. The specifications allow us to define the viewing transformation needed to move from the world coordinate system to the eye coordinate system. 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.

In the next chapter we will discuss modeling, and part of that process is using transformations to place objects that are defined in one position into a different position and orientations in world space. This can be applied to defining the eye point, and we can think of starting with the eye in standard position and applying transformations to place the eye where you want it. If we do that, then the viewing transformation is defined 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 you have organized the viewing information as we have described, you must organize the information you send to the graphics system to define the way your scene is projected to the screen. The graphics system provides ways to define the projection and, once the projection is defined, the system will carry out the manipulations needed to map the scene to the display space. These operations will be discussed later 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 including the eye position, view direction, field of view, and orientation. This defines the viewing transformation that will be used to move from 3D world space to 3D eye space.
- 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. This defines the 3D clipping and projection that will take the view from 3D eye space to 2D eye space.
- The region of the viewing device where the image is to be visible must be defined. This is the window, which should not be confused with the concept of window on your screen, though they often will both refer to the same space.
- When your world is seen in the window on the 2D surface, it must be seen at a particular place, so you must define the location where it will be seen. This defines the location of the viewport within the overall 2D viewing space and the window-to-viewport mapping that takes the 2D eye space to screen space.

We will call these three things setting up your viewing environment, defining your projection, and defining your window and viewport, respectively, and they are discussed in that order in the sections below.

**Setting up the viewing environment**

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. Objects defined in their individual model spaces are then placed in the world space with modeling transformations, as described in the next chapter on modeling. This world space is then transformed by the viewing transformation into a 3D space with the eye in standard position. To define the viewing transformation, you must set up a view by putting your eyepoint in the world space. This world is defined by the coordinate space you assumed when you modeled your scene as discussed earlier. Within that world, you define four critical components for your eye setup: where your eye is located, what point your eye is looking towards, how wide your field of view is, and what direction is vertical with respect to your eye. When these are defined to your graphics API, the geometry in your modeling is transformed with the viewing transformation to create the view as it would be seen with the environment that you defined.

A graphics API defines the computations that transform your geometric model as if it were defined in a standard position so it could be projected in a standard way onto the viewing plane. Each graphics API defines this standard position and has 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 right-handed coordinate system and transforms all the geometry in your scene (and we do mean all the geometry, including lights and directions, 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.

![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 the viewing transformation that is defined from your view definition as we discussed earlier.

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 standard view that would be used unless you define another one 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 probably will not want to define your whole viewable world as lying somewhere behind the \(X-Y\) plane. 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 very important in the modeling chapter below, and is discussed in some depth later in this chapter in terms of defining the view environment for the OpenGL API.

Defining the projection

The viewing transformation above defines the 3D eye space, but that cannot be viewed on our standard devices. In order to view the scene, it must be mapped to a 2D space that has some correspondence to your display device, such as a computer monitor, a video screen, or a sheet of paper. The technique for moving from the three-dimensional world to a two-dimensional world uses a projection operation that you define based on some straightforward fundamental principles.
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. An *orthographic projection* accomplishes this by projecting all the objects in the scene to the viewing plane by parallel lines. For orthographic 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 1.5 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 an 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 modest, but you should look for the differences noted above. It could be useful to use both projections on some of your scenes and compare the results to see how each of the projections works in different situations.

These two projections operate on points in 3D space in rather straightforward ways. For the orthographic projection, all points are projected onto the \((X, Y)\)-plane in 3D eye space by simply omitting the \(Z\)-coordinate. Each point in 2D eye space is the image of a line parallel to the \(Z\)-axis, so the orthographic projection is sometimes called a *parallel projection*. For the perspective projection, any point is projected onto the plane \(Z=1\) in 3D eye space at the point where the line from the point to the origin in 3D eye space meets that plane. Because of similar triangles, if the
point \((x,y,z)\) is projected to the point \((x',y')\), we must have \(x' = x/z\) and \((y' = y/z)\). Here each point in 2D eye space is the image of a line through that point and the origin in 3D eye space.

After a projection is applied, your scene is mapped to 2D eye space, as we discussed in the last chapter. However, the \(z\)-values in your scene are not lost. As each point is changed by the projection transformation, its \(z\)-value is retained for later computations such as depth tests or perspective-corrected textures. In some APIs such as OpenGL, the \(z\)-value is not merely retained but its sign is changed so that positive \(z\)-values will go away from the origin in a left-handed way. This convention allows the use of positive numbers in depth operations, which makes them more efficient.

**View Volumes**

A projection is often thought of in terms of its *view volume*, the region of space that is to be visible in the scene after the projection. With any projection, the fact that the projection creates an image on a rectangular viewing device implicitly defines a set of boundaries for the left, right, top, and bottom sides of the scene; these correspond to the left, right, top, and bottom of the viewing space. In addition, the conventions of creating images include not including objects that are too close to or too far from the eye point, and these give us the idea of front and back sides of the region of the scene that can be viewed. Overall, then, the projection defines a region in three-dimensional space that will contain all the parts of the scene that can be viewed. This region is called the *viewing volume* for the projection. The viewing volumes for the perspective and orthogonal projections are shown in Figure 1.6, with the eye point at the origin; this region is the space within the rectangular volume (left, for the orthogonal projection) or the pyramid frustum (right, for the perspective transformation). Note how these view volumes match the definitions of the regions of 3D eye space that map to points in 2D eye space, and note that each is presented in the left-handed viewing coordinate system described in Figure 1.4.

![Figure 1.6: the viewing volumes for the orthogonal (left) and perspective (right) projections](image)

While the orthographic 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. In fact, this freedom to place your viewing volume for the orthographic projection is not particularly important because you could always use simple translations to center the region you will see.

One of the reasons we pay attention to the view volume is that only objects that are inside the view volume for your projection will be displayed; anything else in the scene will be clipped, that is, be identified in the projection process as invisible, and thus will not be handled further by the graphics system. Any object that is partly within and partly outside the viewing volume will be clipped so
that precisely those parts inside the volume are seen, and we discuss the general concept and process of clipping later in this chapter. The sides of the viewing volume correspond to the projections of the sides of the rectangular space that is to be visible, but the front and back of the volume are less obvious—they correspond to the nearest and farthest space that is to be visible in the projection. These allow you to ensure that your image presents only the part of space that you want, and prevent things that might lie behind your eye from being projected into the visible space.

Calculating the perspective projection

The perspective projection is quite straightforward to compute, and although you do not need to carry this out yourself we will find it very useful later on to understand how it works. Given the general setup for the perspective viewing volume, let’s look at a 2D version in Figure 1.7. Here we see that \( \frac{Y}{Y'} = \frac{Z}{1} \), or simplifying, \( Y' = Y/Z \). Thus with the conventions we have defined, the perspective projection defined on 3D eye space simply divides the original \( X \) and \( Y \) values by \( Z \).

![Figure 1.7: the perspective projection calculation](image)

If we were to write this as a matrix, we would have the following:

\[
\begin{bmatrix}
\frac{1}{Z} & 1 & 1 \\
1 & \frac{1}{Z} & 1 \\
1 & 1 & 1
\end{bmatrix}
\]

While this is a matrix, it is not a linear mapping, and that will have some significance later on when we realize that we must perform perspective corrections on some interpolations of object properties. Note here that we do not make any change in the value of \( Z \), so that if we have the transformed values of \( X' \) and \( Y' \) and keep the original value of \( Z \), we can reconstruct the original values as \( X = X' \times Z \) and \( Y = Y' \times Z \).

Defining the window and viewport

The scene as presented by the projection is still in 2D eye space, and the objects are all defined by real numbers. However, the display space is discrete, so the next step is a conversion of the geometry in 2D eye coordinates to discrete coordinates. This required identifying discrete screen points to replace the real-number eye geometry points, and introduces some sampling issues that must be handled carefully, but graphics APIs do this for you. The actual display space used depends on the window and the viewport you have defined for your image.

To a graphics system, a window is a rectangular region in your viewing space in which all of the drawing from your program will be done. It is usually defined in terms of the physical units of the drawing space. The window will be placed in your overall display device in terms of the device’s coordinate system, which will vary between devices and systems. The window itself will have its own coordinate system, and the window space in which you define and manage your graphics
content will be called *screen space*, 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. While the window is placed in screen space, within the window itself—we will do all our graphics work—we have a separate coordinate system that also has integer coordinates that represent pixel coordinates within the window itself. We will consistently think of the display space in terms of window points and window coordinates because they are all that matter to our image.

You will recall that we have a final transformation in the graphics pipeline from the 2D eye coordinate system to the 2D screen coordinate system. In order to understand that transformation, you need to understand the relation between points in two corresponding rectangular spaces. In this case, the rectangle that describes the scene to the eye is viewed as one space, and the rectangle on the screen where the scene is to be viewed is presented as another. The same processes apply to other situations that are particular cases of corresponding points in two rectangular spaces that we will see later, such as the relation between the position on the screen where the cursor is when a mouse button is pressed, and the point that corresponds to this in the viewing space, or points in the world space and points in a texture space.

![Figure 1.8: correspondences between points in two rectangles](image)

In Figure 1.8, we show two rectangles with boundaries and points named as shown. In this example, we assume that the lower left corner of each rectangle has the smallest coordinate values in the rectangle. So the right-hand rectangle has a smallest $X$-value of $L$ and a largest $X$-value of $R$, and a smallest $Y$-value of $B$ and a largest $Y$-value of $T$, for example (think *left*, *right*, *top*, and *bottom* in this case).

With the names that are used in the figures, we have the proportions

- $X : XMIN:: XMAX: XMIN = u : L :: R : L$
- $Y : YMIN:: YMAX: YMIN = v : B :: T : B$

from which we can derive the equations:

- $(x - XMIN)/(XMAX - XMIN) = (u - L)/(R - L)$
- $(y - YMIN)/(YMAX - YMIN) = (v - B)/(T - B)$

and finally these two equations can be solved for the variables of either point in terms of the other, giving $x$ and $y$ in terms of $u$ and $v$ as:

- $x = XMIN + (u - L)(XMAX - XMIN)/(R - L)$
- $y = YMIN + (v - B)(YMAX - YMIN)/(T - B)$

or the dual equations that solve for $(u,v)$ in terms of $(x,y)$.

This discussion was framed in very general terms with the assumption that all our values are real numbers, because we were taking arbitrary ratios and treating them as exact values. This would
hold if we were talking about 2D eye space, but a moment’s thought will show that these relations cannot hold in general for 2D screen space because integer ratios are only rarely exact. In the case of interest to us, one of these is 2D eye space and one is 2D screen space, so we must stop to ask how to modify our work for that case. For this case, to use the equations above for $x$ and $y$ we would regard the ratios for the right-hand rectangle in terms of real numbers and those for the left-hand rectangle as integers, we can get exact values for the ratios $(u - L)/(R - L)$ and $(v - B)/(T - B)$ and calculate real values for $x$ and $y$, which we then truncate to get the desired integer values. This means that we take the view that an integer coordinate pair represents the unit square with that pair at the lower left of the square.

We noted that the window has a separate coordinate system, but we were not more specific about it. Your graphics API may use either of two conventions for window coordinates. The window may have its origin, or $(0,0)$ value, at either the upper left or lower left corner. In the discussion above, we assumed that the origin was at the lower left because that is the standard mathematical convention, but graphics hardware often puts the origin at the top left because that corresponds to the lowest address of the graphics memory. If your API puts the origin at the upper left, you can make a simple change of variable as $Y' = Y_{MAX} - Y$ and using the $Y'$ values instead of $Y$ will put you back into the situation described in the figure.

When you create your image, you can choose to present it in a distinct sub-rectangle of the window instead of the entire window, 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. Mapping an image to a viewport is done with exactly the same calculations we described above, except that the boundaries of the drawing area are the viewport’s boundaries instead of the window’s. 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, in which case its boundary points will be calculated from the window’s.

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 behavior on your system compatible with the behavior on 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.

A single program can manage several different windows at once, drawing to each as needed for the task at hand. Individual windows will have different identifiers, probably returned when the window is defined, and these identifiers are used to specify which window will get the drawing commands as they are given. 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.

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.

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 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. This makes each integer value represent a smaller real number and so there is less likelihood of two real depths getting the same integer representation.

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. The painter’s algorithm requires that you know the depth of each object in 3D eye space, however, and this can be difficult if your image includes moving parts or a moving eyepoint. Getting depths in eye space is discussed in the modeling chapter in the discussion of scene graphs.

Double buffering

A buffer is a set of memory that is used to store the result of computations, and most graphics APIs allow you to use two image buffers to store the results of your work. These are called the color buffer and the back buffer; the contents of the color buffer are what you see on your graphics screen. If you use only a single buffer, it is the color buffer, and as you generate your image, is written into the color buffer. Thus all the processes of clearing the buffer and writing new content to the buffer—new parts of your image—will all be visible to your audience.

Because it can take time to create an image, and it can be distracting for your audience to watch an image being built, it is unusual to use a single image buffer unless you are only creating one image. Most of the time you would use both buffers, and write your graphics to the back buffer instead of the color buffer. When your image is completed, you tell the system to switch the buffers so that the back buffer (with the new image) becomes the color buffer and the viewer sees the new image. When graphics is done this way, we say that we are using double buffering.

Because it can be disconcerting to actually watch the pixels changing as the image is created, particularly if you were creating an animated image by drawing one image after another, double buffering is essential to animated images. 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: those points \((x,y,z)\) for which \(Ax + By + Cz + D\) is positive and those points for which it is negative. When you define the clipping plane for your graphics API with the functions it provides, you will probably specify it to the API by giving 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 would then be able to use these tests to drive a set of clipping operations on the line segment. We could then extend the work of clipping on line segments to clipping on the segments that are the boundaries of a polygon in order to clip parts of a polygon against one or more planes. 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 viewport 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. It is also quite important to keep the overall display small enough so that the distance between the centers of the images in the display is not larger than the distance between the viewer’s eyes so that he or she can focus each eye on a separate 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 1.9 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.

![Figure 1.9: A stereo pair, including a clipping plane](image)

Many people have physical limitations that do not allow their eyes to perform the kind of 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.

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 nearer parts and more blue in the more distant parts, as shown in Figure 1.10. 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 an 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 1.10: A ChromaDepth™ display, courtesy of Michael J. Bailey, SDSC](image)
Another technique involves preparing images in a way that will allow them to be picked up and displayed by special hardware such as the CrystalEyes® glasses with alternating eye vision from StereoGraphics Corp. 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.

![Figure 1.11: side-by-side images and above-below images, respectively.](image)

**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);
glutInitWindowPosition(topleftX,topleftY);
thisWindow = glutCreateWindow("Your window name here");
The integer value thisWindow that is returned by the glutCreateWindow can be used later to set
the window you just created as the active window to which you will draw. This is done with the
glutSetWindow function, as in

```c
  glutSetWindow(thisWindow);
```
which sets the window identified with thisWindow as the current window. If you are need to
check which window is active, you can use the glutGetWindow() function that returns the
window’s value. In any case, no window is active until the main event loop is entered, as
described in the previous chapter.

A 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.

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 reshapes 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

```c
  glutReshapeFunc(reshape)
```
that 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 and updating the definition of the viewport(s) in the window, or can delegate
some of these to the display function. The reshape callback gets the dimensions of the window as
it has been reshaped, and you can use these to control the way the image is presented in the
reshaped window. For example, if you are using a perspective projection, the second parameter of
the projection definition is the aspect ratio, and you can set this with the ratio of the width and
height you get from the callback, as

```c
  gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);
```
This will let the projection compensate for the new window shape and retain the proportions of the
original scene. On the other hand, if you really only want to present the scene in a given aspect
ratio, then you can simply take the width and height and define a viewport in the window that has
the aspect ratio you want. If you want a square presentation, for example, then simply take the
smaller of the two values and define a square in the middle of the window as your viewport, and
then do all your drawing to that viewport.

Any viewport you may have defined in your window probably 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

```c
    (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);
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);
```

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 `gluLookAt` function with the parameters

```c
  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 eye-point 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 functions have the property that the inverse of a product is the product of the inverses in reverse order, as in

\[(f \ast g)^{-1} = g^{-1} \ast f^{-1}\]

for any \(f\) and \(g\), 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. Because of this we will usually see the `gluLookAt(...)` function as one of the first things to appear in the `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 later 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 \texttt{zNear} value (the distance from the viewer to the plane that will contain the nearest points that can be displayed), and the \texttt{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}
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  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 1.4 above. This call is

\begin{verbatim}
  glFrustum( left, right, bottom, top, near, far );
\end{verbatim}

Perhaps the \texttt{gluPerspective} function is more natural, so we will not discuss the \texttt{glFrustum} function further and leave it to the student who wants to explore 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 \texttt{X}, \texttt{Y}, and \texttt{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.

\begin{verbatim}
  glOrtho(xLow,xHigh,yLow,yHigh,zNear,zFar);
\end{verbatim}

The viewing space is still the same left-handed space as noted earlier, so the \texttt{zNear} and \texttt{zFar} values are the distance from the \texttt{X-Y} plane in the negative direction, so that negative values of \texttt{zNear} and \texttt{zFar} refer to positions behind the eye (that is, in positive \texttt{Z}-space). There is no alternate to this function in the way that the \texttt{glFrustum(...)} is an alternative to the \texttt{gluLookAt(...)} function for parallel projections.

**Managing hidden surface viewing**

In the Getting Started chapter, we introduced the structure of a program that uses OpenGL and saw the \texttt{glutInitDisplayMode} function, called from \texttt{main}, is a way to define properties of the display. This function also allows the use of hidden surfaces if you specify \texttt{GLUT_DEPTH} as one of its parameters.

\begin{verbatim}
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
\end{verbatim}

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 \texttt{glEnable} function, as shown below.

\begin{verbatim}
  glEnable(GL_DEPTH_TEST);
\end{verbatim}
From that point the depth buffer is in use and you need not be concerned about hidden surfaces. OpenGL uses integer values for the depth test and so is vulnerable to Z-fighting. The default behavior of the depth test is that a point passes the depth test (and so is recorded in the scene) if its \( z \)-value is less than the \( z \)-value stored in the depth buffer, but this can be changed by using the \texttt{glDepthFunc(value)} function, where value is a symbolic constant. We will only use the depth test in its default form, but you can see OpenGL reference sources for more details.

If you want to turn off the depth test, there is a \texttt{glDisable} function as well as the \texttt{glEnable} function. Note the use of the enable and disable functions in enabling and disabling the clipping plane in the example code for stereo viewing.

**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 \texttt{GLUT_DOUBLE} to set up double buffering. This indicates that you will use two buffers, called the \textit{back buffer} and the \textit{front buffer}, in your drawing. The content of the front buffer is displayed, and all drawing will take place to the back buffer. So in your \texttt{display()} function, you will need to call \texttt{glutSwapBuffers()} when you have finished creating the image; that will cause the back buffer to be exchanged with the front buffer and your new image will be displayed. An added advantage of double buffering is that there are a few techniques that use drawing to the back buffer and examination of that buffer’s contents without swapping the buffers, so the work done in the back buffer will not be seen.

**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 \texttt{GL_CLIP_PLANE0} through \texttt{GL_CLIP_PLANE5}. The clipping planes are defined by the function \texttt{glClipPlane(plane, equation)} where \textit{plane} is one of the pre-defined clipping planes above and \textit{equation} is a vector of four \texttt{GLfloat} values. Once you have defined a clipping plane, it is enabled or disabled by a \texttt{glEnable(GL_CLIP_PLANEn)} function or equivalent \texttt{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);  
...  
gldisble(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 $ep$ 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 reshapes 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.

**Questions**

This set of questions covers your recognition of issues in viewing and projections as you see them
in your personal environment. They will help you see the effects of defining views and applying
projections and the other topics in this chapter

1. Find a comfortable environment and examine the ways your view of that environment depend
   on your eyepoint and your viewing direction. Note how objects seem to move in front of and
   behind other objects as you move your eyepoint, and notice how objects move into the view
   from one side and out of the view on the other side as you rotate your viewing direction. (It
   may help if you make a paper or cardboard rectangle to look through as you do this.)

2. Because of the way our eyes work, we cannot see an orthogonal view of a scene. However, if
   we keep our eyes oriented in a fixed direction and move around in a scene, the view directly
   ahead of us will approximate a piece of an orthogonal view. For your familiar environment as
   above, try this and see if you can sketch what you see at each point and put them together into a
   single image.

3. Consider a painter’s algorithm approach to viewing your environment; write down the objects
   you see in the order of farthest to nearest to your eye. Now move to another position in the
   environment and imagine drawing the things you see in the same order you wrote them down
   from the other viewpoint. What things are out of order and so would have the farther thing
drawn on top of the nearer thing? What conclusions can you draw about the calculations you would need to do for the painter’s algorithm?

4. Imagine defining a plane through the middle of your environment so that everything on one side of the plane is not drawn. Make this plane go through some of the objects you would see, so that one part of the object would be visible and another part invisible. What would the view of the environment look like? What would happen to the view if you switched the visibility of the two sides of the plane?

Exercises

These exercises ask you to carry out some calculations that are involved in creating a view of a scene and in doing the projection of the scene to the screen.

5. Take a standard perspective viewing definition with, say, a 45° field of view, an aspect ratio of 1.0, a distance to the front plane of the viewing frustum of 1.0, and a distance to the back plane of the viewing frustum of 20.0. For a point \(P=(x,y,1.)\) in the front plane, derive the parametric equation for the line segment within the frustum that all projects to \(P\). Hint: the line segment goes through both the origin and \(P\), and these two points can serve to define the segment’s equation.

6. Create an \(X\)-\(Y\)-\(Z\) grid on a piece of paper using the convention that \(X\) is to the right, \(Y\) is up, and \(Z\) is into the page; an example is shown at the right.
   (a) In your familiar environment from question 1, place everything (or a number of things) from that environment into the grid with measured coordinates, to get familiar with coordinate systems to define positions. For this example, put everything into the space with non-negative coordinates (the first octant in 3D Cartesian coordinates) to make it easier to deal with the coordinates.
   (b) Define a position and direction for your eye in that same space and visualize what will be seen with that viewing definition, and then go to your space and see if your visualization was accurate. If not, then work out what was causing the inaccuracy.

7. In the numerically-modeled environment above, place your eyepoint in the \((X,Z)\)-center of the space (the middle of the space left to right and front to back), and have your eye face the origin at floor height. Calculate the coordinates of each point in the space relative to the eye coordinate system, and try to identify a common process for each of these calculations.

Experiments

In these experiments, we will work with the very simple model of the house in Figure 1.5, though you are encouraged to replace that with a more interesting model of your own. The code for a function to create the house centered around the origin that you can call from your \(display()\) is given below to help you get started.
void drawHouse(void)
{
  point3 myHouse[10]=
  { {-1.0, -1.0,  2.0 }, { -1.0,  1.0,  2.0 },
     {  0.0,  2.0,  2.0 }, {  1.0,  1.0,  2.0 },
     {  1.0, -1.0,  2.0 }, { -1.0, -1.0, -2.0 },
     { -1.0,  1.0, -2.0 }, {  0.0,  2.0, -2.0 },
     {  1.0,  1.0, -2.0 }, {  1.0, -1.0, -2.0 } },

  int i;

  glBegin(GL_LINE_STRIP);
  for ( i=0; i<5; i++)
    glVertex3fv(myHouse[i]);
  glEnd();

  glBegin(GL_LINE_STRIP);
  for ( i=0; i<5; i++)
    glVertex3fv(myHouse[i+5]);
  glEnd();

  for ( i=0; i<5; i++) {
    glBegin(GL_LINE_STRIP);
    glVertex3fv(myHouse[i]);
    glVertex3fv(myHouse[i+5]);
    glEnd();
  }
}

8. Create a program to draw the house with this function or to draw your own scene, and note what happens to the view as you move your eyepoint around the scene, always looking at the origin (0,0,0).

9. With the same program as above and a fixed eye point, experiment with the other parameters of the perspective view: the front and back view planes, the aspect ratio of the view, and the field of view of the projection. For each, note the effect so you can control these when you create more sophisticated images later.