Chapter 7

Coordinate Transformations

This chapter describes the subroutines that you use to set up a graphics scene, for example, how much of the screen to use, where to locate the viewing point, where shapes in the scene are located in space, how much of the scene is visible, and what type of view you are looking at.

- Section 7.1 describes the coordinate systems that the GL uses and explains how they are derived.
- Section 7.2, “Projection Transformations,” describes how coordinates are mapped to the screen and tells you how to set up your scene projection.
- Section 7.3, “Viewing Transformations,” tells you how to set up the view of your scene.
- Section 7.4, “Modeling Transformations,” tells you how to locate a geometry in space and how to change its size and orientation.
- Section 7.5, “Controlling the Order of Transformations,” explains how to put the items in your scene where you want them to go and how to save a scene setup for later use.
- Section 7.6, “Hierarchical Drawing with the Matrix Stack,” tells you how to draw items that are grouped into hierarchies.
- Section 7.7, “Viewports, Screenmasks, and Scrboxes,” tell you how to define the visible limits of your scene.
- Section 7.8, “User-Defined Transformations,” tells you how to define your own way of looking at the scene and manipulating items in it.
- Section 7.9, “Additional Clipping Planes,” tells you how to define the visible limits of your scene using boundaries that you create.

Changing how your graphics scene is viewed and changing where items are placed in it are called *transformations*. Three-dimensional transformations are difficult to describe, and even harder to visualize. You may need to read the chapter more than once, study the illustrations, experiment with the sample code, and write your own programs in order to gain an understanding of these topics.

7.1 Coordinate Systems

You use many different coordinate systems in the process of drawing a graphics scene. Figure 7-1 illustrates the coordinate systems used at different stages of the drawing process.

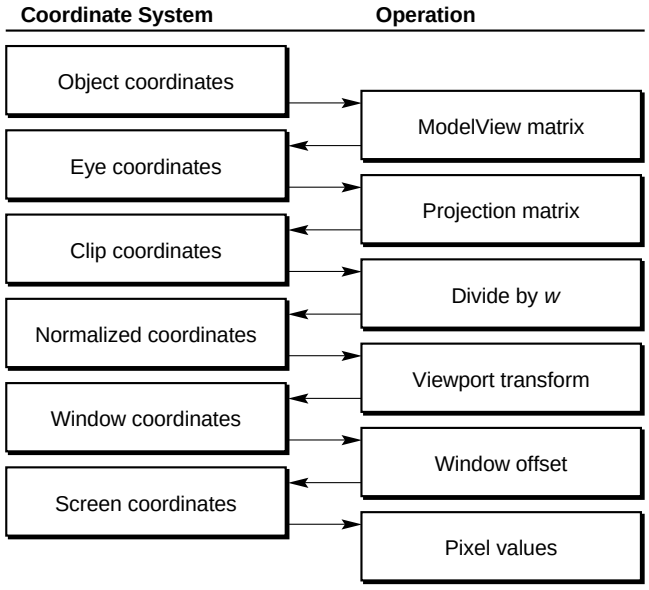

Figure 7-1 Coordinate Systems

The coordinate systems on the IRIS begin with a 3-D system defined in *right-handed cartesian floating point* coordinates, called the *object coordinate system*. Geometry vertices that you specify in *(x, y, z)* triplets are in this

coordinate system. There are no limits to the size of coordinates in this system (other than the largest legal floating point value).

The *eye coordinate system* is the result of *transforming* (through matrix multiplication) the geometry coordinates by the contents of the modeling and viewing (*ModelView*) matrix. The eye coordinate system is the system in which lighting calculations are performed internally.

When the IRIS transforms points expressed in eye coordinates by the *Projection* matrix, the output is expressed in *clip coordinates*. Values returned by a call to `getgpos()` are expressed in this coordinate system.

The next system is called the *normalized coordinate system*. Clip coordinates are converted to normalized coordinates by first limiting *x*, *y*, and *z* to the range $-w \leq x,y,z \leq w$ (clipping), then dividing *x*, *y*, and *z* by *w*. The result is normalized coordinates in the range $-1 \leq x,y,z \leq 1$. The space of normalized coordinates is called the *3-D unit cube*.

The *x* and *y* coordinates of this 3D unit cube are scaled directly into the next coordinate system, usually called the *window coordinate system*. The pixel at the lower-left corner of a window has window coordinates (0,0).

Window coordinates, modified by a window offset that represents the window's location on the screen, represent the *screen coordinate system*, which corresponds to pixel values. Screen coordinates are typically thought of as 2-D, but in fact all three dimensions of the normalized coordinates are scaled, and there is a screen *z* coordinate that can be used for hidden surface removal, described in Chapter 8, or depth cueing, described in Chapter 11.

Note: If the ModelView matrix were separated into a Model matrix and a View matrix, the coordinate system between these matrices would be correctly referred to as the *world coordinate system*. Because the GL concatenates modeling and viewing transformations into a single matrix (ModelView), there are no world coordinates in the GL.

To get from geometry vertices to a scene displayed on your screen, you have to specify a *projection transformation*, to define how images in your scene are projected to the screen and a *viewing transformation*, to define what type of view you have and where you are viewing the scene from. This is sort of like setting up a camera to look through at the scene. You can move the camera around and change lenses to get different views with these transformations.

7.2 Projection Transformations

You can project an image onto the screen in one of three ways:

- Perspective - Items far away are smaller than items close to you, and parallel lines appear to recede into the distance toward a vanishing point.

  This is how you see the real world through your eye's ability to perceive depth. Consequently, scenes with a perspective projection will look and feel more natural to you, unless you have exaggerated some parameter that causes distortion.

- Window - Items are seen in perspective, but it is possible to create an asymmetric view of the scene.

- Orthographic - Items are projected through a rectangular viewing volume, but are not seen in perspective.

All the projection transformations work basically the same way. A viewing volume is mapped into the unit cube, the geometry outside the cube is clipped out, and the remaining data is linearly scaled to fill the window (actually the *viewport*, which is discussed in Section 7.7, "Viewports, Screenmasks, and Scrboxes"). The viewpoint is where your eye is with respect to the viewing volume. This is called the *eye position*, or simply the *eye*.

Projection transformations are either perspective or orthographic in nature. The difference between the three projection transformations is the definitions of their *viewing volumes*. Perspective projections create a volume that has the shape of a pyramid with the top cut off. Orthographic projections create a viewing volume that has parallel sides.

7.2.1 Perspective Projection

Viewing items in perspective on the computer screen is like looking through a rectangular piece of perfectly transparent glass. Imagine drawing a line from your eye through the glass until it hits the item, coloring a dot on the glass where the line passes through the same color on the item. If this were done for all possible lines through the glass, if the coloring were perfect, and if the eye not allowed to move, the picture painted on the glass would be indistinguishable from the true scene.

The collection of all the lines leaving your eye and passing through the glass would form an infinite four-sided pyramid with its apex at your eye. Anything

outside the pyramid would not appear on the glass, so the four planes passing through your eye and the edges of the glass would block your view of the portions of the items outside the glass. These are called the left, right, bottom, and top *clipping planes*.

The geometry hardware also provides two other clipping planes that eliminate anything too far or too near to be seen clearly with the eye, just like your eye cannot focus on objects that are very far away or very close to you. These are called the *near* and *far* clipping planes. Near and far clipping is always turned on, but it is possible to set the near plane very close to the eye and / or the far plane very far from the eye so that all the geometries of interest are visible.

Because floating point calculations are not exact, it is a good idea to move the near plane as far as possible from the eye, and to bring in the far plane as close as possible. This gives optimal resolution for distance-based operations such as hidden surface removal and depth-cueing, as discussed in Chapters 8 and 11.

For a perspective view, the visible region of the *world* (your graphics scene) looks like a pyramid with the top sliced off. The technical name for this is a *frustum*, or *rectangular viewing frustum*.

In a perspective projection, the Projection matrix maps a frustum of eye coordinates so that it exactly fills the unit cube (after *x*, *y*, and *z* are each divided by w). This frustum is part of a pyramid whose apex is at the origin (0.0, 0.0, 0.0). The base of the pyramid is parallel to the x-y plane, and it extends along the negative z axis.

In other words, it is the view obtained with the eye at the origin looking down the negative z axis, and the plate of glass perpendicular to the line of sight.

Use `perspective()` to define a perspective projection:

```
void perspective(Angle fovy, float aspect, Coord znear, Coord zfar)
```

`perspective()` has four arguments: the *field of view* in the *y* direction, the *aspect ratio*, and the distances to the *near* and *far* clipping planes.

The field of view (*fovy*) is an angle made by the top and bottom clipping planes that is measured in tenths of degrees, so a 90 degree angle is specified as 900.

The *aspect ratio* is the ratio of the *x* dimension of the glass to its *y* dimension. It is a floating point number. For example, if the aspect ratio is 2.0, the glass is

twice as wide as it is high. Typically, you choose the aspect ratio so that it is the same as the aspect ratio of the window on the screen, but it need not be. The distances to the near and far clipping planes are floating point values.

The `keepaspect()` subroutine tells the window manager to maintain the window's *x* and *y* dimensions in a 1 to 1 ratio, that is, a square. An equally accurate picture could be made by setting a 2 to 1 ratio, but then the aspect ratio in `perspective()` would have to be changed to 2.0. You might want to try this to see how it looks. Also try varying other parameters of perspective—change the field of view and the near and far clipping planes to see the effects.

In a real application, you probably want to match the aspect ratio of `perspective()` to the aspect ratio of the window when you sweep out a window of arbitrary shape and size. Use `getsize(x,y)` to return the height and width in pixels of a GL window.

Figure 7-2 shows a frustum that demonstrates eyepoint, FOV angles, clipping planes, and aspect ratio.

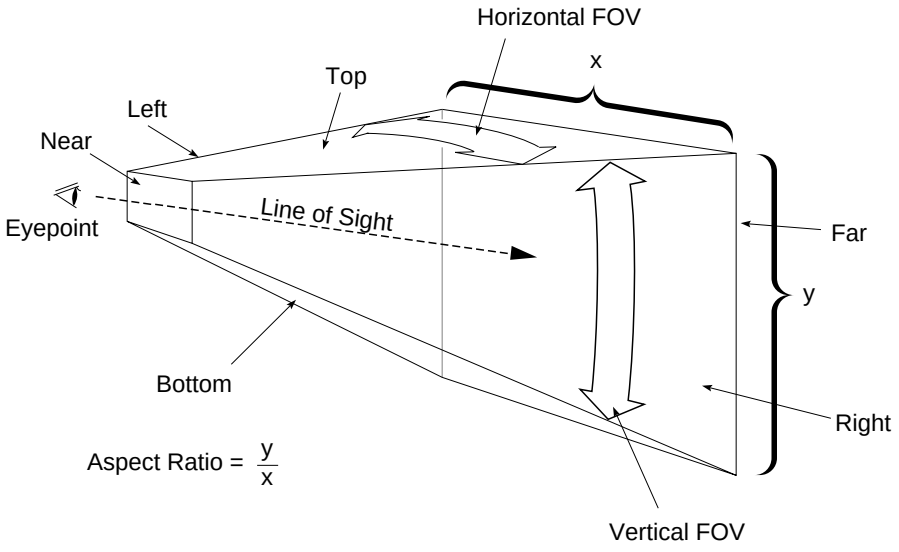

Figure 7-2 Frustum

This sample program, *perspective.c*, draws a single rectangle whose shade varies from bright red at z=0.0 to bright green at z=-4.0.

```
#include <stdio.h>
#include <gl/gl.h>
#define RGB_BLACK 0x000000
#define RGB_RED 0x0000ff
#define RGB_GREEN 0x00ff00

float v[4][3] = {
    {-3.0, 3.0, 0.0},
    {-3.0, -3.0, 0.0},
    { 2.0, -3.0, -4.0},
    { 2.0, 3.0, -4.0}
};

main()
{
    long xsize, ysize;
    float aspect;

    if (getgdesc(GD_BITS_NORM_SNG_RED) == 0) {
        fprintf(stderr, "Single buffered RGB not available\n");
    return 1;
    }
    prefsize(400, 400);
    winopen("perspective");
    mmode(MVIEWING);
    getsize(&xsize, &ysize);
    aspect = (float)xsize / (float)ysize;
    perspective(900, aspect, 2.0, 5.0);
    RGBmode();
    gconfig();
    cpack(RGB_BLACK);
    clear();
    bgnpolygon();
        cpack(RGB_RED);
        v3f(v[0]);
        v3f(v[1]);
        cpack(RGB_GREEN);
        v3f(v[2]);
        v3f(v[3]);
    endpolygon();
    sleep(10);
    gexit();
    return 0;
}
```

Figure 7-3 shows a top view of the scene drawn in the sample program *perspective.c*. This view is not the view you see on your screen when you run the program; this view lets you see outside of the viewing volume.

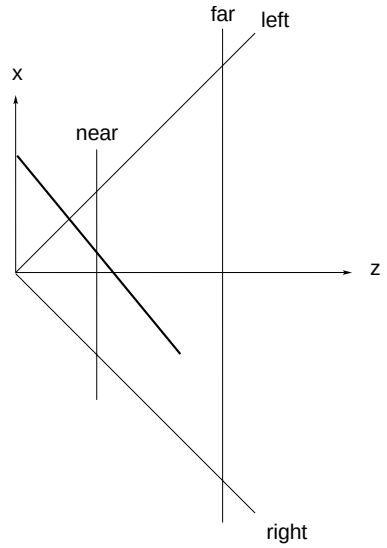

Figure 7-3 Clipping Planes

The heavy line shows the rectangle that the program draws. The eye is at (0.0, 0.0, 0.0), so the rectangle recedes into the distance, and because the field of view is 90 degrees and the near clipping plane is at 2.0, the rectangle is clipped by the top, bottom, and near clipping planes.

The visible part of the polygon nearest the eye is not bright red, but already looks yellowish because the rectangle is clipped by the near clipping plane. If you bring in the near clipping plane toward you, the near end of the polygon looks more and more red.

7.2.2 Window Projection

A `window()` projection transformation shows the world in perspective view. It is similar to `perspective()`, but its viewing frustum is defined in terms of distances to the left, right, bottom, top, and near and far clipping planes:

```
void window(Coord left,Coord right,Coord bottom,Coord top,Coord near,Coord far)
```

Figure 7-4 illustrates the window projection transformation.

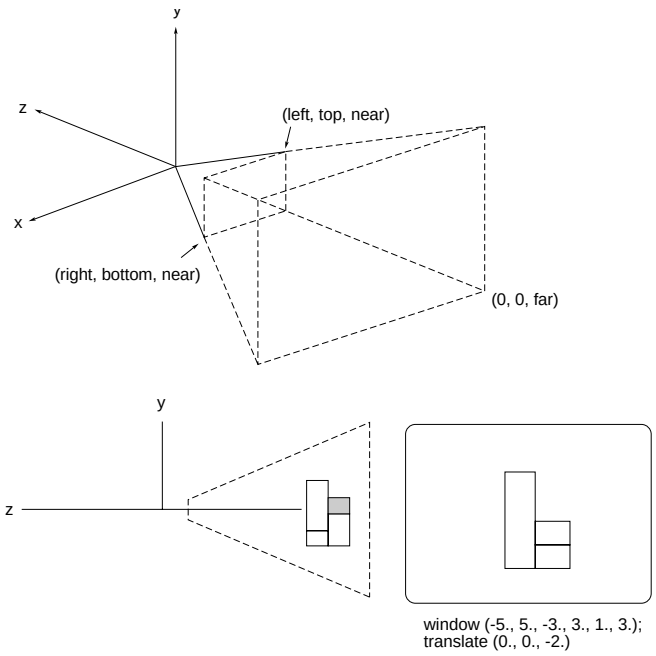

Figure 7-4 The `window()` Projection Transformation

Because `window()` allows separate specifications at all six surfaces of the viewing frustum, it can be used to specify asymmetric volumes. These are useful in special circumstances, such as multiple view simulations, and for special operations, such as antialiasing using the accumulation buffer, described in Chapter 15.

`window()` specifies the position and size of the rectangular viewing frustum closest to the eye, the location of the near and far clipping planes. `window()` projects a perspective view of the image onto the screen.

7.2.3 Orthographic Projections

The remaining two projection subroutines are the orthographic transformations, `ortho()` and `ortho2()`. Their viewing volumes are rectangular parallelepipeds (rectangular boxes). They correspond to the limiting case of a perspective frustum as the eye moves infinitely far away and the field of view decreases appropriately.

Another way to think of the `ortho()` subroutines is that the geometry outside the box is clipped out, then the geometry inside is projected parallel to the *z* axis onto a face parallel to the *x*-*y* plane.

Figure 7-5 shows an example of a 3D orthographic projection.

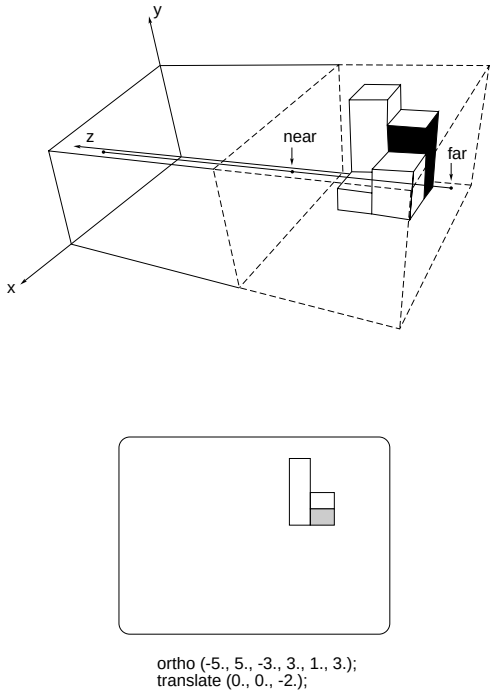

Figure 7-5 The `ortho()` Projection Transformation

`ortho()` allows you to specify the entire box—the *x*, *y*, and *z* limits. `ortho2()` requires a specification of only the *x* and *y* limits. The z limits are -1 and 1.

`ortho2()` is usually used for 2-D drawing, where all the z coordinates are zero. It is really a 3D transformation; if you use `ortho2()` and try to draw objects with *z* coordinates outside the range $-1.0 \leq z \leq 1.0$, they are clipped out.

ortho

`ortho()` defines a box-shaped enclosure in the eye coordinate system:

```
void ortho(Coord left,Coord right,Coord bottom,Coord top,Coord znear,Coord zfar)
```

The arguments *left*, *right*, *bottom*, and *top* define the *x* and *y* clipping planes. *znear* and *zfar* are distances along the line of sight and are negative. In other words, the *z* clipping planes are located at $z = -znear$ and $z = -zfar$.

Note: Because `ortho()` allows separate specification of the left, right, bottom, and top clipping planes rather than just the width and height of the viewing volume, it can move the effective viewpoint off the positive *z* axis. Thus, although `ortho()` is a projection transformation, it exhibits some aspects of a viewing transformation. Be careful when using `ortho()` with asymmetric volume boundaries so that you do not duplicate this viewpoint offset in your viewing transformation.

ortho2

`ortho2()` defines a 2-D clipping rectangle:

```
void ortho(Coord left,Coord right,Coord bottom,Coord top,Coord near,Coord far)
```

The arguments *left*, *right*, *bottom*, and *top* define the sides of the rectangle. Choose the values for `ortho2()` carefully, because of the transformation of floating point values to integers as the coordinate systems are modified to use screen coordinates, and because of the need to align lines on pixel centers rather than on pixel boundaries. You need to add a .5 to each value in the `ortho2()` call, so that pixels are centered on whole numbers. If you do not do this, you might find that accumulated round-off errors in the arithmetic of pixel operations cause your pixel specifications to be off by a pixel or more.

The `ortho2()` statement in the following code fragment defines the correct clipping rectangle for the window created with the corresponding `prefposition()` statement:

```
prefposition(101, 500, 101, 500);
ortho2(100.5, 500.5, 100.5, 500.5);
```

This causes the clipping rectangle to clip only items that are completely within the window defined by the `prefposition()` statement. These two statements ensure that the `ortho2()` statement clips on (and point-samples within) boundaries that are completely visible within the window defined by the `prefposition()` statement.

7.3 Viewing Transformations

All the projection transformations discussed so far have assumed that the eye is at least looking toward the negative *z* axis, and the two perspective subroutines actually assume that the eye is at the origin. For the orthogonal transformations, it does not make sense to talk about the exact position of the eye, only about the direction it is looking.

The viewing transformations allow you to specify the position of the eye and the direction toward which it is looking. `polarview()` and `lookat()` provide convenient ways to do this.

`polarview()` assumes that the object you are viewing is near the origin. The eye position is specified by a radius (distance from the origin) and by angles measuring the azimuth and elevation. The specification is similar to a polar coordinates. There is still one degree of freedom, because these values tell only where the eye is relative to the object.

`lookat()` allows you to specify both the viewpoint and the reference point toward which the eye is looking, and a *twist* angle to specify which way is up.

Both viewing subroutines work with a projection subroutine. If you want to view a point (for example: 1, 2, 3) from another point (4, 5, 6) in a perspective view, use both `perspective()` and `lookat()`.

When the orthographic projections are used, the exact position of the eye used in the viewing subroutines does not make a difference; all that matters is the viewing line of sight.

The viewing transformations work mathematically by transforming, with rotations and translations, the position of the eye to the origin and the viewing direction so that it lies along the negative *z* axis.

7.3.1 Viewpoint in Polar Coordinates

`polarview()` defines the viewer's position in polar coordinates. The first three arguments, *dist*, *azim*, and *inc*, define a viewpoint. *dist* is the distance from the viewpoint to the origin in world coordinates. *azim* is the azimuthal angle in the *x*-*y* plane, measured from the *y* axis. *inc* is the incidence angle in the *y*-*z* plane, measured from the *z* axis. The line of sight is the line between the viewpoint and the origin (0,0,0).

The *twist* rotates the viewpoint around the line of sight using the *right-hand rule* (as you look down the positive rotation axis to the origin, positive rotation is counterclockwise). All angles are specified in tenths of degrees and are integers.

Figure 7-6 shows examples of `polarview()`.You don't see anything in the top picture because the eye is positioned on the origin, looking at the origin.

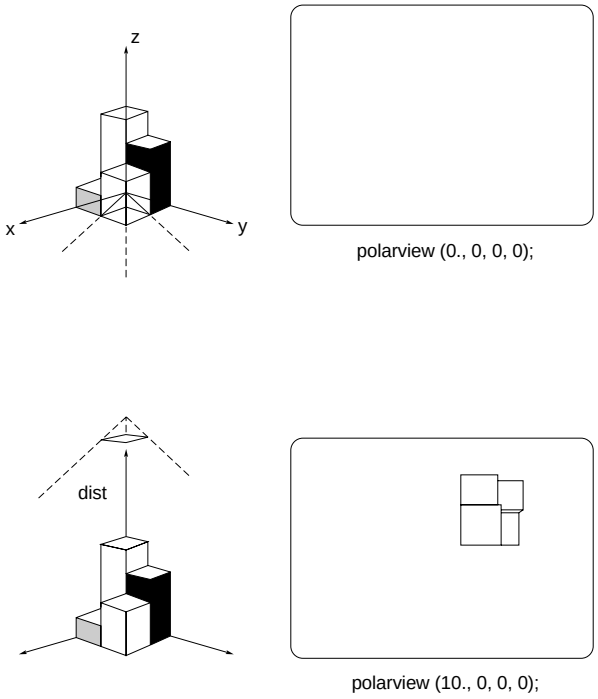

Figure 7-6 The `polarview()` Viewing Transformation

7.3.2 Viewpoint along a Line of Sight

`lookat()` defines a viewpoint and a reference point on the line of sight in world coordinates. The viewpoint is at (*vx*, *vy*, *vz*) and the reference point is at (*px*, *py*, *pz*). These two points define the line of sight. The *twist* measures right-hand rotation about the *z* axis in the eye coordinate system.

Figure 7-7 shows examples of `lookat()`.

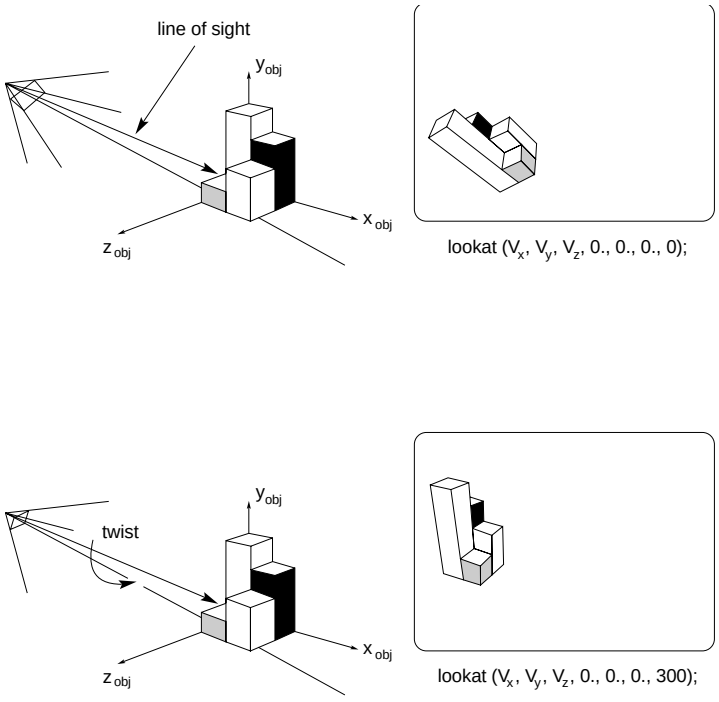

Figure 7-7 The `lookat()` Viewing Transformation

The *lookat.c* sample program draws a wireframe cube centered at the origin. Its window has an aspect ratio of 3:2, or 1.5:1, so the aspect ratio of `perspective()` is 1.5 to match. `lookat()` looks from the point (5.0, 4.0, 6.0) at the corner (1.0, 1.0, 1.0) of the cube, so that corner appears centered in the window.

The near and far clipping planes are at 0.1 and 10.0. Nothing is clipped out on the near end, but the far corner of the cube is about 10.49 away from the eye, so the far clipping plane clips a bit of the corner.

This program, *lookat.c*, illustrates how to use a viewing transformation with a projection transformation:

```
#include <gl/gl.h>

long v[8][3] = {
    {-1, -1, -1},
    {-1, -1, 1},
    {-1, 1, 1},
    {-1, 1, -1},
    { 1, -1, -1},
    { 1, -1, 1},
    { 1, 1, 1},
    { 1, 1, -1},
};
int path[16] = {
    0, 1, 2, 3,
    0, 4, 5, 6,
    7, 4, 5, 1,
    2, 6, 7, 3
};

void drawcube()
{
    int i;
    bgnline();
    for (i = 0; i < 16; i++)
        v3i(v[path[i]]);
    endline();
}

main()
{
    prefsize(600, 400);
    winopen("lookat");
    mmode(MVIEWING);
    perspective(300, 1.5, 0.1, 10.0);
    lookat(5.0, 4.0, 6.0, 1.0, 1.0, 1.0, 0);
    color(BLACK);
    clear();
    color(WHITE);
    drawcube();
    sleep(10);
    gexit();
    return 0;
}
```

7.4 Modeling Transformations

When you create a geometry, the GL creates it with respect to its own coordinate system. You can manipulate the entire object using the *modeling transformation* subroutines: `rotate()`, `rot()`, `translate()`, and `scale()`. Each time you specify a transformation such as `rotate()` or `translate()`, the software automatically generates a *transformation matrix* that premultiplies the current matrix by the factors by which the coordinate system is to be rotated or translated. See Appendix C for the transformation matrices used by the GL.

Figure 7-8 shows some examples of modeling transformations.

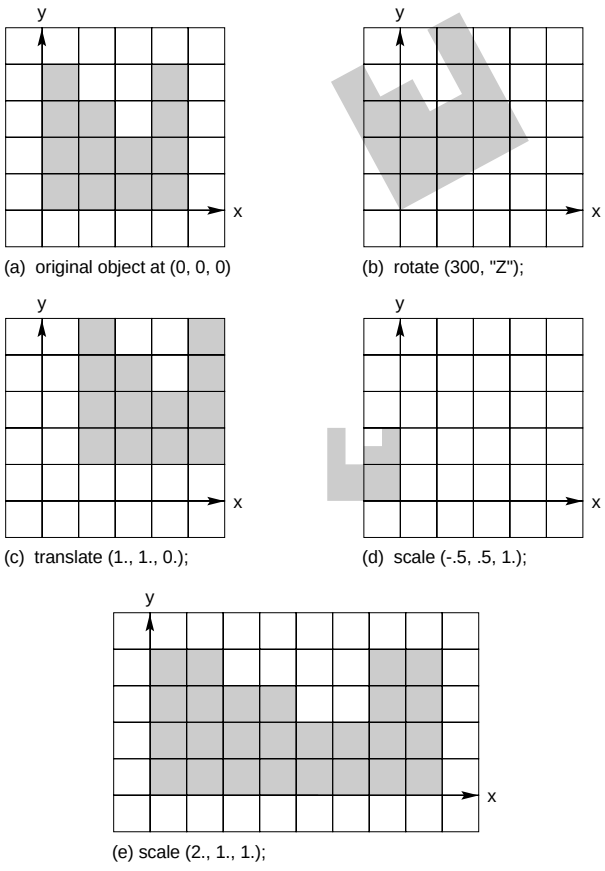

Figure 7-8 Modeling Transformations

7.4.1 Rotation

Use `rotate()` to rotate a geometry by specifying an *angle* and an *axis* of rotation:

```
void rotate(Angle a, char axis)
```

The angle is given in tenths of degrees according to the right-hand rule. A character (either upper- or lowercase *x*, *y*, or *z*), defines the axis of rotation. The angle and axis are used to compute a 4x4 rotation matrix *R* (see Appendix C), that premultiplies the current matrix *T* to give *RT*.

`rot()` is the same as rotate; it specifies an angle and an axis of rotation in floating point values, but the angle is measured in degrees:

```
void rot(Angle a, char axis)
```

You need to pay close attention to the order in which you specify transformation operations, or your program might provide you with surprising results. See Section 7.5, "Controlling the Order of Transformations," and Figure 7-9 for more information about the importance of the order of modeling transformations and how to preserve untransformed coordinates.

The geometry in Figure 7-8 (a) is rotated 30 degrees with respect to the *y* axis in Figure 7-8 (b). All geometries drawn after you call `rotate()` or `rot()` are rotated. Use `pushmatrix()` and `popmatrix()` to preserve and restore the unrotated coordinate system.

7.4.2 Translation

Use `translate()` to move the coordinate system origin to a point (*x*,*y*,*z*) specified in the current coordinate system:

```
void translate(Coord x, Coord y, Coord z)
```

The *x*,*y*, and *z* coordinates are used to compute a 4×4 translation matrix *X* (see Appendix C), that premultiplies the current matrix *T*, to give *XT*.

The geometry in Figure 7-8 (a) is translated by (1,1,0) in Figure 7-8 (c).

All geometries drawn after you call `translate()` are translated. Use `pushmatrix()` and `popmatrix()` to preserve and restore the untranslated coordinate system.

7.4.3 Scaling

Use `scale()` to shrink, expand, or mirror a geometry:

```
void scale(float x, float y, float z)
```

The *x*, *y*, and *z* scale factors are used to compute a 4×4 scale matrix *S* (see Appendix C) that premultiplies the current matrix *T* to give *ST*. Values with magnitudes of more than 1 cause expansion; values with magnitudes of less than 1 cause shrinkage. Negative values cause mirroring.

Note: There may be a performance penalty for using non-uniform scaling if you are using lighting calculations.

The geometry in Figure 7-8 (a) is shrunk to one-quarter of its original size and is mirrored about the *y* axis in Figure 7-8 (d). It is scaled only in the *x* direction in Figure 7-8 (e). All geometries drawn after you call `scale()` are scaled. Use `pushmatrix()` and `popmatrix()` to preserve and restore the unscaled coordinate system.

You can combine `rotate()`, `rot()`, `translate()`, and `scale()` to produce more complicated transformations. The order in which you apply these transformations is important. Figure 7-9 shows two sequences of `translate()` and `rotate()`. Each sequence has different results.

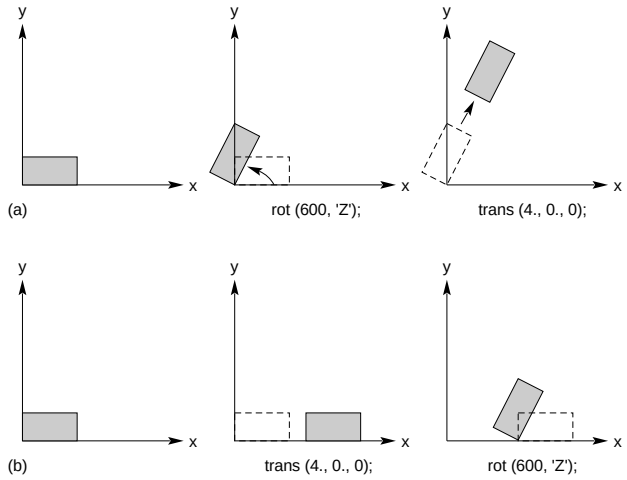

Figure 7-9 Effects of Sequence of Translations and Rotations

7.5 Controlling the Order of Transformations

Each time you specify a transformation such as `rotate()` or `translate()`, the software automatically generates a *transformation matrix* that specifies the amount by which the coordinate system is to be rotated or translated. The current transformation matrix is then premultiplied by the generated matrix, effecting the desired transformation. The actual transformations are done in an order opposite to that specified. In other words, you specify the viewing matrix first, followed by the modeling transformations, so that vertices are first positioned correctly in world coordinates, then the eye point moves to the origin looking down the negative *z* axis.

The reason for the reverse order is that the transformations are accomplished in the hardware by matrix multiplication, and historically, the matrix multiplication hardware allows only left multiplications. Thus, a vector *v*, transformed by a modeling transformation *M* and a viewing transformation *V* (in that order), undergoes the following multiplications:

$$v \rightarrow vM \rightarrow (vM)V = vMV$$

The hardware concatenates modeling and viewing transformations onto one matrix to save time, but because it performs multiplication only on the left, it must start with *V*, then generate *MV*. Because the Projection matrix is stored separately from the ModelView matrix, it does not matter whether projection is specified before or after the modeling and viewing transformations.

7.5.1 Current Matrix Mode (mmode)

The graphics system maintains three transformation matrices—the ModelView matrix, the Projection matrix, and the Texture matrix. As described at the beginning of this chapter, the ModelView matrix transforms coordinates from object coordinates to eye coordinates. The Projection matrix transforms coordinates from eye coordinates to clip coordinates. The Texture matrix transforms texture coordinates directly from object coordinates to clip coordinates. Its transformation is typically unrelated to that specified by the ModelView and Projection matrices.

All programs should set matrix mode to `MVIEWING`, `MPROJECTION`, or `MTEXTURE`, depending on which operation is to be done, before any matrix operations are performed. A fourth matrix mode, `MSINGLE`, reconfigures the graphics system to have only a single matrix that transforms vertices directly

from object coordinates to clip coordinates. This mode is obsolete and should not be used. For historical reasons, however, `MSINGLE` is the default.

Use `mmode()` to specify which of three matrices is the current matrix: ModelView (`MVIEWING`), Projection (`MPROJECTION`), or Texture (`MTEXTURE`):

```
void mmode(short m)
```

The current matrix is on the top of the matrix stack.

When you are not doing lighting calculations, you should use MVIEWING for the modeling, viewing, and projection transformations. See Chapter 9 for information about the transformation matrices when lighting is used.

Note: Even in `mmode(MVIEWING)`, any calls to projection transformations (`perspective, window, ortho or ortho2`) will affect the Projection matrix.

7.6 Hierarchical Drawing with the Matrix Stack

A drawing can be composed of many copies of simpler drawings, each of which can be composed of still simpler drawings, and so on. For example, if you were writing a program to draw a picture of a bicycle, you might want to have one subroutine that draws a wheel, and to call that subroutine twice to draw two wheels, appropriately translated. The wheel itself might be drawn by calling the spoke drawing subroutine 36 times, appropriately rotated. In a still more complicated drawing of many bicycles, you might like to call the bicycle drawing routine many times.

Suppose the bicycle is described in a coordinate system where the bottom bracket (the hole through which the pedal crank's axle runs) is the origin. You would draw the frame relative to this origin, but translate forward a few inches before drawing the front wheel (defined, say, relative to its axis). Then you would like to remove the forward translation to get back to the bicycle's frame of reference, and translate back to draw another instance of the wheel.

The modeling transformation that describes the bicycle's frame of reference is *M*, and that *S* and *T* are transformations (relative to *M*) to move forward for drawing the front wheel, and back for the back wheel, respectively. You would like to draw the wheel using transformation *SM* for the front wheel and *TM* for the back wheel.

This is easily accomplished using the ModelView matrix stack. At any point in a drawing, the current ModelView matrix sits at the top of the matrix stack; it contains all the modeling and viewing transformations called thus far. In the bicycle example, this lumped-together transformation is called *M*. Any vertex is transformed by the top matrix, which is what you want to do for drawing the frame.

Two subroutines, `pushmatrix()` and `popmatrix()`, push and pop the ModelView matrix stack. `pushmatrix()` pushes the matrix stack down and copies the current matrix to the new top. Thus, after a `pushmatrix()`, there are two copies of *M* on top. Translating by a translation matrix *T* leaves the stack with *TM* on top and *M* underneath. The wheel is then drawn once using the *SM* transformation. `popmatrix()` eliminates the *TM* on top, leaving *M*, and another `pushmatrix()` makes two copies of *M*.

```
... /* code to get M on top of the stack */
pushmatrix();
translate(-dist_to_back_wheel, 0.0, 0.0);
drawwheel();
popmatrix();
pushmatrix();
translate(dist_to_front_wheel, 0.0, 0.0);
drawwheel();
popmatrix();
drawframe();
```

pushmatrix

Use `pushmatrix()` to push down the transformation stack, duplicating the current matrix:

```
void pushmatrix(void)
```

If the transformation stack originally contains one matrix, *M*, it will contain two copies of *M* (after a `pushmatrix()`). You can modify only the top copy. Because only the ModelView matrix is stacked, you should call `pushmatrix()` only while `mmode()` is `MVIEWING`.

popmatrix

Use `popmatrix()` to pop the top matrix off the transformation stack:

```
void popmatrix(void)
```

Call `popmatrix()` only while `mmode()` is `MVIEWING`.

Note: It is important to have the same number of matrix pushes and matrix pops, so you don't try to pop a matrix off an empty stack.

This sample program, *hierarchy.c*, uses a hierarchical description of a simple car. It is so simple that the car body is a rectangle, the wheels are square, and everything is 2D. Note that the positions and orientations of the nine cars are independent, and each wheel has a different rotation.

```
#include <math.h>
#include <gl/gl.h>

#define X    0
#define Y    1
#define XY   2

float carbody[4][XY] = {
   {-0.1, -0.05},
   { 0.1, -0.05},
   { 0.1, 0.05},
   {-0.1, 0.05}
};
float wheel[4][XY] = {
   {-0.015, -0.015},
   { 0.015, -0.015},
   { 0.015, 0.015},
   {-0.015, 0.015}
};

void drawwheel()
{
   color(GREEN);
   bgnpolygon();
      v2f(wheel[0]);
      v2f(wheel[1]);
      v2f(wheel[2]);
      v2f(wheel[3]);
   endpolygon();
}
```

```
void drawcar()
{
   int i;
   color(RED);
   bgnpolygon();
      v2f(carbody[0]);
      v2f(carbody[1]);
      v2f(carbody[2]);
      v2f(carbody[3]);
   endpolygon();
   for (i = 0; i < 4; i++) {
      pushmatrix();
         translate(carbody[i][X], carbody[i][Y], 0.0);
         rotate(200*(i+1), 'z');
         drawwheel();
      popmatrix();
   }
}
main()
{
   float xoffset, yoffset;
   Angle ang;

   prefsize(400, 400);
   winopen("hierarchy");
   mmode(MVIEWING);
   ortho2(-1.0, 1.0, -1.0, 1.0);
   color(BLACK);
   clear();
   for (xoffset = -0.5; xoffset <= 0.5; xoffset += 0.5) {
      for (yoffset = -0.5; yoffset <= 0.5; yoffset += 0.5) {
         ang = 3600 * drand48();
         pushmatrix();
            translate(xoffset, yoffset, 0.0);
            rotate(ang, 'z');
            drawcar();
      popmatrix();
   }
 }
 sleep(10);
 gexit();
 return 0;
}
```

This shorter and perhaps more interesting program illustrates hierarchy and more complex transformations. It is a computer model of a ring and gear drawing toy. In the toy, a plastic ring is pinned to a piece of paper, and you place a pen through a hole in a smaller gear inside the ring and rotate the gear. As the moving gear rolls around the fixed gear, you draw interesting patterns.

```
#include <gl/gl.h>

#define PEN_TO_CENTER  0.2
#define R0             0.35
#define R1             0.6

void drawdot()
{
   translate(PEN_TO_CENTER, 0.0, 0.0);
   pnt2i(0, 0);
}

void draw1(theta)
float theta;
{
   pushmatrix();
      rot(theta, 'z');
      translate(R1 + R0, 0.0, 0.0);
      rot(-theta * R1 / R0, ‘z’);
      drawdot();
   popmatrix();
}

main()
{
   float theta;

   prefsize(400, 400);
   winopen("spirograph");
   mmode(MVIEWING);
   ortho2(-2.0, 2.0, -2.0, 2.0);
   color(BLACK);
   clear();
   color(WHITE);
   for (theta = 0.0; theta < 3600.0; theta += 0.25)
      draw1(theta);
   sleep(10);
   gexit();
   return 0;
}
```

In this example, R0 and R1 are the radii of the two gears. and PEN_TO_CENTER is the distance from the pen to the center of the moving gear. With a slight modification of this program, you can build a sophisticated drawing program that has three levels of gears, each moving along the next at a uniform rate. It would be difficult to build this one out of plastic!

```
#include <gl/gl.h>

#define PEN_TO_CENTER0  0.2
#define R0              0.35
#define R1              0.6
#define R2              0.8

void drawdot()
{
   translate(PEN_TO_CENTER, 0.0, 0.0);
   pnt2i(0, 0);
}

void draw1(theta)
float theta;
{
   pushmatrix();
      rot(theta, ‘z’);
      translate(R1 + R0, 0.0, 0.0);
      rot(-theta * R1 / R0, 'z');
      drawdot();
   popmatrix();
}

void drawx(theta)
float theta;
{
   pushmatrix();
      rot(theta, ‘z’);
      translate(R2 + R1, 0.0, 0.0);
      rot(-theta * R2 / R1, 'z');
      draw1(theta);
   popmatrix();
}
```

```
main()
{
 float theta;
 prefsize(400, 400);
 winopen("spirograph2");
 mmode(MVIEWING);
 ortho2(-3.0, 3.0, -3.0, 3.0);
 color(BLACK);
 clear();
 color(WHITE);
 for (theta = 0.0; theta < 18000.0; theta += 0.25)
   drawx(theta);
 sleep(10);
 gexit();
 return 0;
}
```

7.7 Viewports, Screenmasks, and Scrboxes

The *viewport* is the area of the window that displays an image. You specify it in window coordinates, where the coordinates of the pixel at the lower-left corner of the window are (0, 0). The visible screen area varies from system to system.

viewport

Use `viewport()` to specify, in window coordinates, the area of the window that displays an image:

```
void viewport(Screencoord left,
              Screencoord right,
              Screencoord bottom,
              Screencoord top)
```

Its arguments (*left, right, bottom, top*) define a rectangular area on the window by specifying the left, right, bottom, and top coordinates. The portion of eye coordinates that `window()`, `ortho()`, or `perspective()` describes is mapped into the viewport. By default, when you open a window on the screen, its viewport is set to cover the whole window.

Although window coordinates are continuous, not discrete, the parameters passed to `viewport()` are integer values. Thus, the viewport is always an integer number of pixels wide and high. Pixel *x,y* is included in the viewport

if $x \geq left$ and $x \leq right$ and $y \geq$ bottom and $y \leq top$. Because pixel centers have integer coordinates in the continuous window coordinate space, the window area included in a viewport is exactly:

($left$-0.5) $\leq x <$ ($right$+0.5), ($bottom$-0.5) $\leq y <$ (top+0.5)

Note: To correctly map object coordinates one-to-one to window coordinates, call:

```
ortho(-0.5, width -0.5, -0.5, height-0.5)
viewport(0, width-1, 0, height-1);
```

where *width* and *height* are the integer pixel sizes of the window.

getviewport

Use `getviewport()` to return the current viewport:

```
void getviewport(Screencoord *left,
            Screencoord *right,
            Screencoord *bottom,
            Screencoord *top)
```

The arguments (*left*, *right*, *bottom*, *top*) are the addresses of four memory locations. These are assigned the left, right, bottom, and top coordinates of the current viewport.

scrmask

The screenmask is a specified rectangular area of the screen to which all drawings are clipped. Use `scrmask()` to define the screenmask:

```
void scrmask(Screencoord left,
          Screencoord right,
          Screencoord bottom,
          Screencoord top)
```

The viewport maps coordinates to the window, and the screenmask specifies the portion of the window to which the geometry can be drawn. The screenmask is a setting that regards only the physical display within the window. The screenmask and viewport are usually set to the same area. `viewport()` sets both the viewport and the screenmask to the same area; `scrmask()` sets only the screenmask, which must be placed entirely within the viewport. See Chapter 3 for one use of a screenmask—clipping characters.

getscrmask

Use `getscrmask()` to return the coordinates of the current screenmask in the arguments *left*, *right*, *bottom*, and *top*:

```
void getscrmask(Screencoord *left,
             Screencoord *right,
             Screencoord *bottom,
             Screencoord *top)
```

pushviewport

The system maintains a stack of viewports, and the top element in the stack is the current viewport. Use `pushviewport()` to duplicate the current viewport and screenmask and push them onto the stack:

```
void pushviewport(void)
```

popviewport

Use `popviewport()` to pop the stack of viewports and set the viewport and screenmask (the viewport on top of the stack is lost):

```
void popviewport(void)
```

scrbox

`scrbox()` is a dual of the scrmask capability. Rather than limiting drawing effects to a screen-aligned subregion of the viewport, it tracks the screen-aligned subregion (screen box) that has been affected. Unlike `scrmask()`, which defaults to the viewport boundary if not explicitly enabled, `scrbox()` must be explicitly turned on to be effective. Call scrbox with the desired mode:

```
void scrbox(long arg)
```

While enabled (mode `SB_TRACK`), `scrbox()` maintains left-most, right-most, lowest, and highest window coordinates of all pixels that are scan-converted. By default, `scrbox()` is reset (mode `SB_RESET`), forcing the left-most and lowest screen box values to be greater than the right-most and highest screen box values. While `scrbox()` is set to mode `SB_HOLD`, the current boundary values are unchanged, regardless of any drawing operations.

Because `scrbox()` operates on the pixels that result from the scan conversion of points, lines, polygons, and characters, it correctly handles wide lines, antialiased (smooth) points and lines, and characters. `scrbox()` results are only guaranteed to bound the modified frame buffer region, but they might exceed the bounds of this region due to implementation.

getscrbox

Use `getscrbox()` to return the current scrbox into *left*, *right*, *bottom*, and *top*:

```
void getscrbox(long *left, long *right, long *bottom, long *top)
```

7.8 User-Defined Transformations

Modeling and viewing transformation commands premultiply the current matrix (one of ModelView, Projection, or Texture) with a 4×4 matrix that they compute based on their parameters.

The current matrix is the ModelView matrix on the top of the ModelView stack if `mmode()` is `MVIEWING`, the Projection matrix if `mmode()` is `MPROJECTION`, or the Texture matrix if `mmode()` is `MTEXTURE`.

You can also premultiply, or replace, the current matrix with a 4x4 matrix of your own.

Use `multmatrix()` to premultiply the current matrix by the given matrix (*m*):

```
void multmatrix (Matrix m)
```

That is, if *T* is the current matrix, `multmatrix(m)` replaces *T* with *MT*.

Use `getmatrix()` to copy the current matrix to an array:

```
void getmatrix(Matrix m)
```

Use `loadmatrix()` to load a 4×4 floating point matrix, *m*, onto the stack, replacing the current top of the stack:

```
void loadmatrix(Matrix m)
```

7.9 Additional Clipping Planes

Geometry is always clipped against the boundaries of the six-plane viewing volume defined by the current Projection matrix. `clipplane()` allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *z* axis, against which all geometry is clipped. You can specify up to six additional planes. Because the resulting clipping region is always the intersection of the (up to) 12 half-spaces, it is always convex.

`clipplane()` uses the following format:

```
void clipplane(long index,long mode,float params[])
```

where:

index	integer in the range 0 through 5. Indicates which of the six clipping planes is being modified.
mode	one of three tokens:
`CP_DEFINE`	Use the plane equation passed in *params* to define a clipping plane. The clipping plane is neither enabled nor disabled.
`CP_ON`	Enable the (previously defined) clipping plane.
`CP_OFF`	Disable the clipping plane (default).
params	array of four floats that specify a plane equation. A plane equation is usually thought of as the column vector *A,B,C,D*. In this case, A is the first component of the *params* array, and D is the last. A four-component vertex array (see `v4f()` in Chapter 2) can be passed as a plane equation, where vertex *x* becomes *A*, *y* becomes *B*, and so on.

`clipplane()` specifies a half-space using a four-component plane equation. When you call clipplane with mode `CP_DEFINE`, this object coordinate plane equation is transformed to eye coordinates using the inverse of the current Model View matrix. (You cannot use `clipplane()` when the matrix mode is `MSINGLE`).

Once you have defined a clipping plane, you enable it by calling `clipplane()` with the `CP_ON` argument, and with arbitrary values passed in *params*.

While the program is drawing, after a clipping plane has been defined and enabled, each vertex is transformed to eye coordinates, where it is dotted with the clipping plane P_{eye}, as shown in (EQ 7-1).

(EQ 7-1)

$$P_{\text{object}} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

$$P_{\text{eye}} = M^{-1}_{\text{modelview}} \cdot P_{\text{object}}$$

Eye coordinates whose dot product with P_{eye} is positive or zero are inside the region, and require no clipping. Those eye coordinate vertices whose dot product is negative are clipped. Because `clipplane()` clipping is done in eye coordinates, changes to the Projection matrix have no effect on its operation.

By default, all six clipping planes are undefined and disabled. The behavior of enabled but undefined clipping plane(s) is also undefined.

It is sometimes convenient to define a clipping plane based on a point, and a direction in object coordinates. The plane equation is obtained from the dot product of the point and the normal:

(EQ 7-2)

$$\text{Point} = \begin{bmatrix} P_x & P_y & P_z \end{bmatrix}$$

$$\text{Normal} = \begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix}$$

$$\text{Plane} = \begin{bmatrix} N_x \\ N_y \\ N_z \\ -\begin{bmatrix} P_x & P_y & P_z \end{bmatrix} \bullet \begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix} \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$