TRUE BASIC

The Original BASIC

3D Graphics Toolkit

Reference Guide

© 2002-2010 True Basic

3-D Graphics Toolkit

Introduction

The True BASIC *3-D Graphics Toolkit* fully supports both perspective and parallel projections with simple and useful defaults that make it easy to get good-looking images. The *3-D Graphics Toolkit* also includes advanced routines that give you complete control over the images.

Included in this Package

The following programs are included with the True BASIC 3-D Graphics Toolkit disk:

3Dlib – 3-D Graphics Toolkit (Loadable module).

3Dcont – 3-D Contour Plotting (Not loadable).

The following Demo Programs:

Axes3	House3	Project3	Splot3	Zmesh3
Bars3	ImagData	Record	Topo3	Zplot3
Blocks3	Oblique3	Scale3	Zbar3	Zsplot3
Cube3	PlayBack	Spiral3	Zdata3	

Additional Reading

For full descriptions of 3-D graphics, consult:

Fundamentals of Computer Graphics,
Foley and van Dam, Addison-Wesley, 1983.

Principles of Interactive Computer Graphics, Second Edition,
Newman and Sproull, McGraw-Hill, 1979.

Computer Graphics: A Programming Approach,
Harrington, McGraw-Hill, 1983.

Mathematical Elements for Computer Graphics,
Rogers and Adams, McGraw-Hill, 1976.

Getting Started

Before you start, **load** the compiled version of 3Dlib*. This is not strictly necessary, but it allows the 3-D graphics programs run much faster! Then try running several sample programs. Two of them are described below; the others are described in the "Sample Programs" section.

The Axes3 Program

Call up the **Axes3** program and run it. It shows the 3-D outline of a box, and draws axes with tick marks, and a circle.

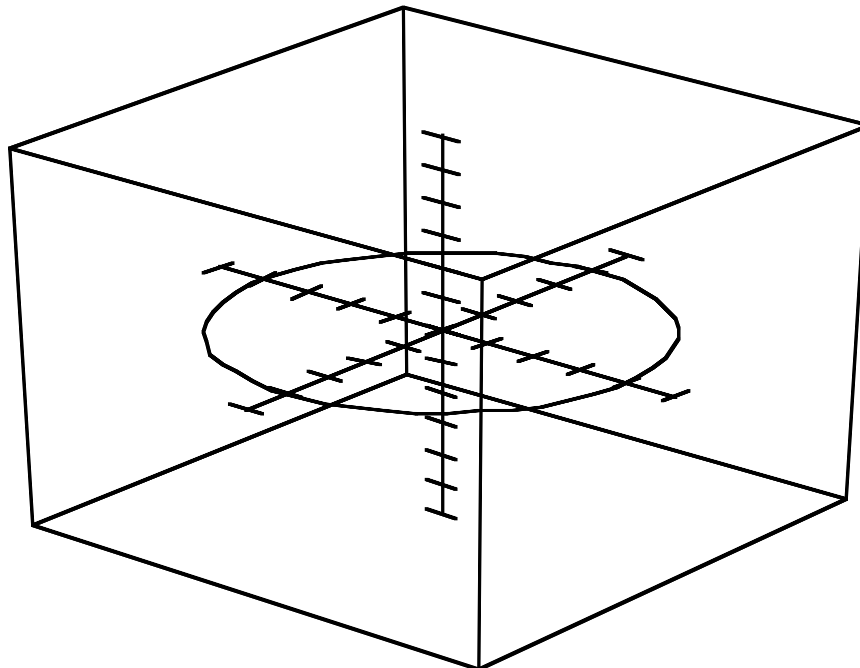

Figure 40.1: Unit Cube with Ticks and Circle.

Now look at the source program. It consists mainly of calls to 3-D graphics subroutines. In a nutshell, they are:

PersWindow opens a 3-D window onto some rectangular block of space, and positions the "camera" so it gives a clear view of that volume. *Frame3* draws the edges of the viewing volume. *Ticks3* draws the *x, y,* and *z* axes with tick marks spaced at the intervals specified. *CircleZ3* draws a circle in the plane $z = 0$, much like True BASIC's BOX CIRCLE statement.

The Zplot3 Program

Call up the **Zplot3** program and run it. It shows the “contour plot” of a fairly complicated function.

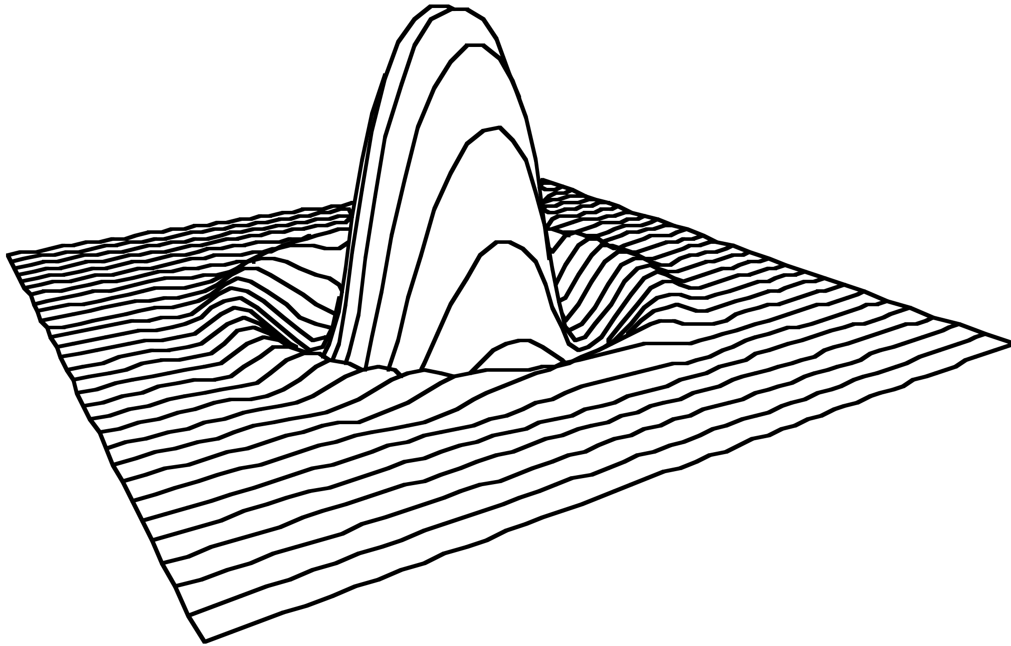

Figure 40.2: Contour Plot of a Function.

Now look at the source code. It contains only a few subroutine calls. *PersWindow*, as above, describes the viewing volume. *SetCamera3* repositions the camera to a new location in 3-D space. *Zplot* draws a contour graph of the function *F*. And finally, *Zplot3* defines the function *F* that will be plotted. That’s all it takes!

The Zbar3 Program

Call up the **Zbar3** program and run it. It shows a 3-D bar chart of a number of data points, superimposed on top of a topographic map of the same data.

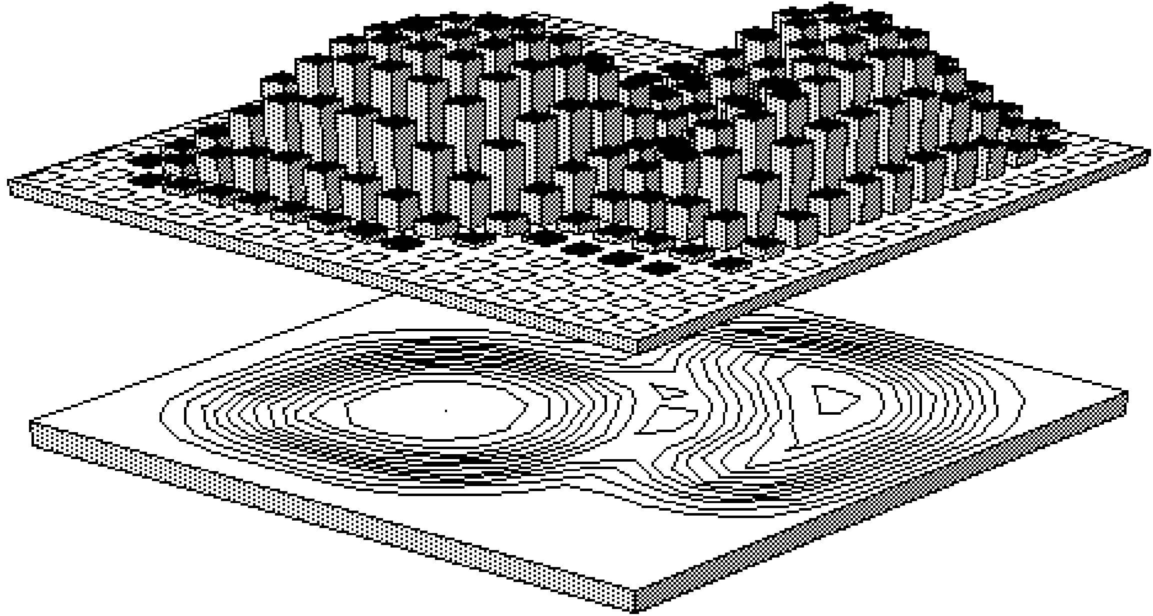

Figure 40.3: Bar Chart and Topographic Map of Data Set.

The source code contains several advanced subroutine calls. *ZbarData* and *TopoData* do most of the work; they draw the bar chart and topographic map, respectively. The *Block3* routine draws the solid slabs on which the bars and topo map sit. The other routines perform minor tasks such as setting bar colors.

What Next?

The next two sections describe in detail all the nuts and bolts of 3-D graphics. You can read them, if you wish, or you can skip them and proceed directly to descriptions of the *3-D Graphics Toolkit* subroutines.

But you'll probably want to turn back to these sections later to get a good understanding of how 3-D graphics works.

3-D Theory

Three-dimensional graphics shows "projections" of 3-D objects on a flat, two-dimensional screen. This is a complicated affair. Let's begin by going over the basics.

The Camera Analogy

Using 3-D graphics is like taking a picture with a camera.

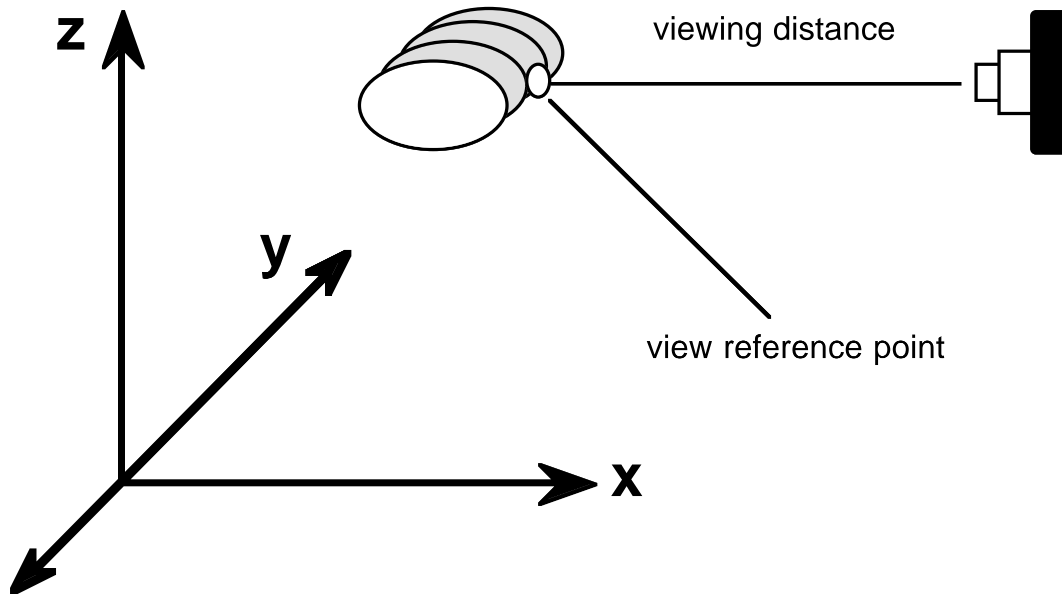

Figure 40.4: Taking a Picture.

We are taking a picture of an object. This requires both the object itself and a camera. The camera is a certain distance away from the object and aims at some point. (This point is usually somewhere on the object itself, but need not be.)

Let's now switch to computer graphics terms. The camera is aimed at the **reference point**. The film within the camera corresponds to the **view plane**. The **viewing distance** is the distance between the reference point and the view plane.

Perspective and Parallel Projections

A camera records the 3-dimensional world on a 2-dimensional piece of film. In computer graphics, this is called "projecting" objects onto the view plane. The *3-D Graphics Toolkit* handles two kinds of projections: perspective projections and parallel projections (sometimes called orthographic projections). Figure 40.5 shows the difference. When viewing an object, you must first choose whether you want to see a perspective or parallel projection.

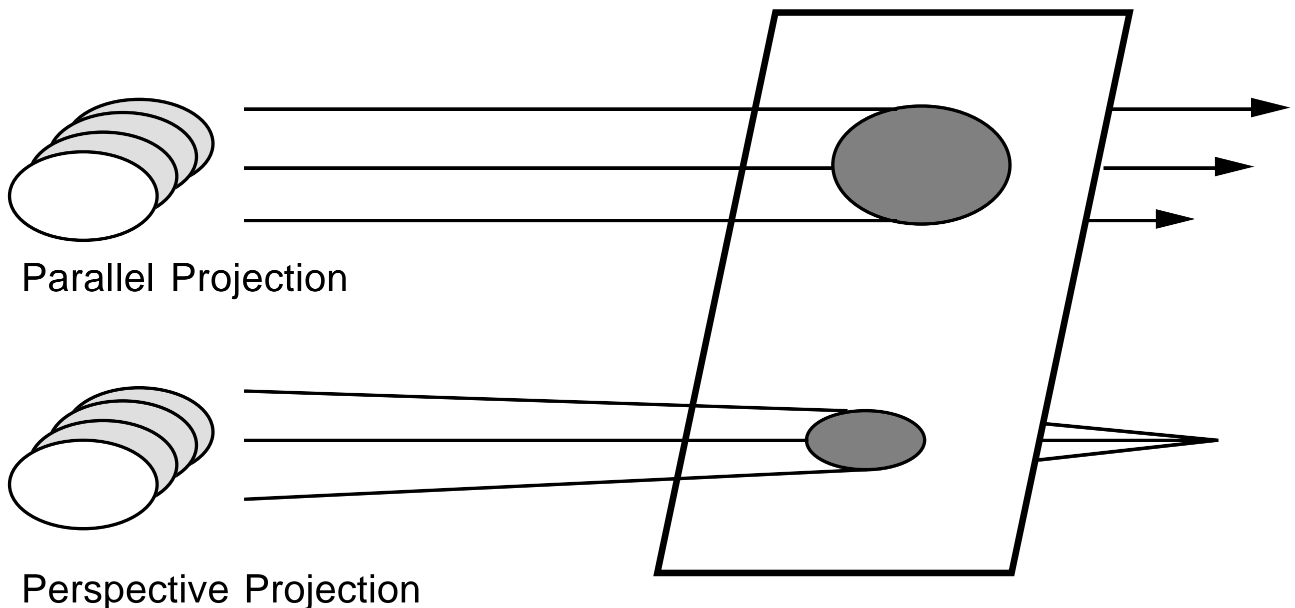

Figure 40.5: Perspective and Parallel Projections.

Perspective projections mimic how we see things. "Projection lines" emerging from an object all converge at one point — the **center of perspective**. The camera must be situated between the reference point and the center of perspective. (In fact, the *3-D Graphics Toolkit* gives an error if you try to place the camera behind the center of perspective.) As in real life, the camera position determines the size of the projected image. As the camera approaches the object, the image gets bigger. As it gets farther away, the image shrinks.

Parallel projections are simpler than perspective projections since their projection lines never converge — they remain parallel to infinity. Thus moving the camera nearer to the object, or farther away, doesn't affect the size of the object's image. Parallel projections look less realistic than perspective projections but are easier to draw (by hand) and often easier to use. They are often used for architecture, mechanical engineering, and so forth. Some common forms of parallel projections are called axonometric, isometric, dimetric, cavalier, and cabinet projections. Despite the forbidding range of names, these are all parallel projections, and the *3-D Graphics Toolkit* handles them all.

To see the difference between perspective and parallel projections, call up the **Project3** program from your disk and run it.

Three-Dimensional Windows

Refering back to Figure 40.4, you'll see that there are many variables involved in pointing a camera at an object. You must supply the reference point, the camera position, the projection type, and perhaps the center of perspective. Fortunately, you don't have to control each of these variables yourself. You can give one subroutine call and have the *3-D Graphics Toolkit* supply a reasonable view.

The *PersWindow* and *ParaWindow* subroutines do all the work. The two routines are exactly the same except that one provides a perspective projection, and the other a parallel projection. Both are 3-dimensional equivalents of True BASIC's SET WINDOW statement; they specify a block viewing volume instead of a flat viewing rectangle. See Figure 40.6.

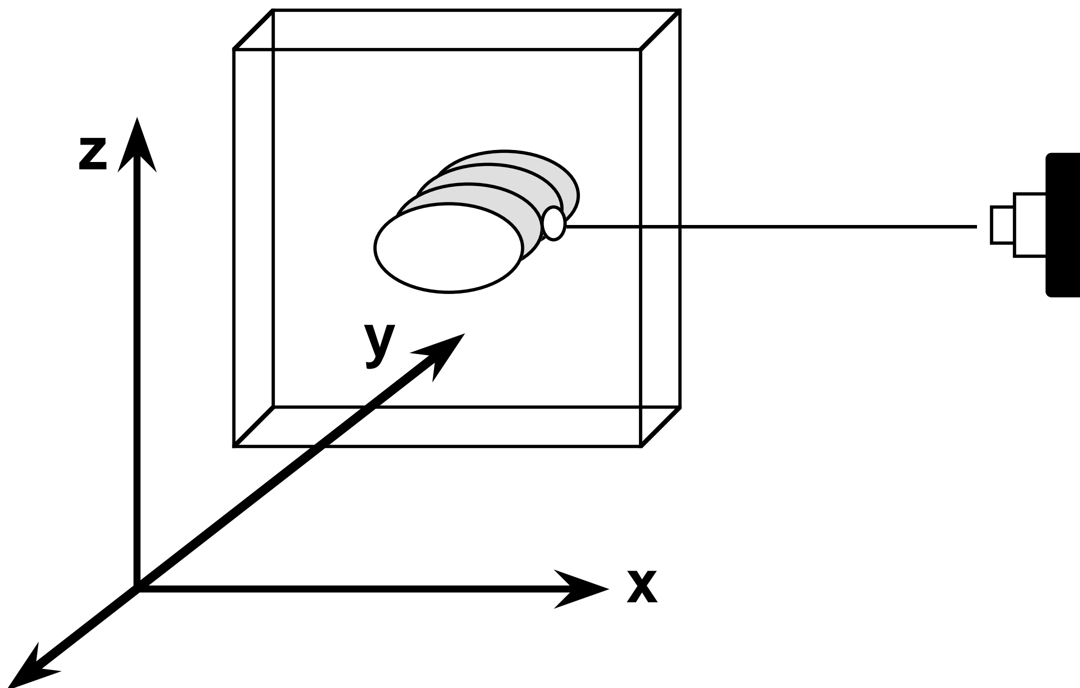

Figure 40.6: The Viewing Volume.

These routines automatically set the reference point, camera position, and (if relevant) center of perspective. The center of the viewing volume becomes the reference point. The camera is positioned so that three sides of the viewing volume are visible. The *x* axis is horizontal, and the *z* axis is vertical. The positive *y* axis recedes "into the screen."

For perspective views, the camera is positioned far enough away from the reference point so that the perspective looks natural. The center of perspective is behind the camera.

You can adjust any of these defaults after you open the window. Call *SetCamera3* to adjust the camera position, and *SetRefPt3* to adjust the reference point. (At a lower level, the *SetViewPlane3* subroutine changes the view plane normal, and *SetDistance3* changes the viewing distance. *SetCP3* controls the center of perspective.)

Your Viewing Window and its Aspect Ratio

Remember that your 3-D volume is projected onto a 2-D plane. As in True BASIC's own graphics, only a portion of this infinite plane is visible on your screen. In fact, your current True BASIC window acts as the window onto the view plane. You see exactly the part of the view plane that lies inside your current window. Figure 40.7 illustrates.

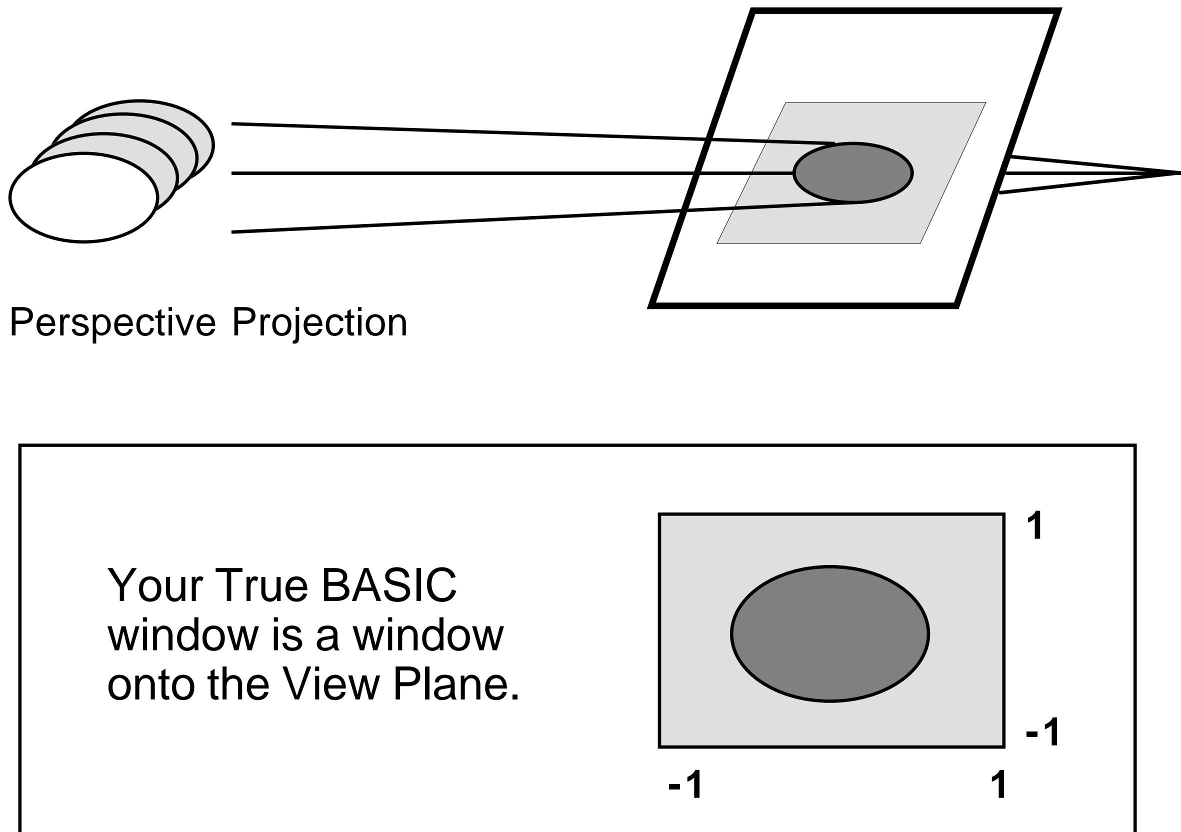

Figure 40.7. Your window onto the View Plane.

When you specify a 3-D viewing volume, the graphics routines automatically try to scale everything so that the results are all visible on your screen. In particular, they reset your current True BASIC window so the origin (0,0) is at the center.

You can use True BASIC's SET WINDOW statement, at any time, to change your window onto the view plane. The bigger you make your window, the more of the view plane you will see (but the smaller the projected image will look). And the smaller you make it, the less you'll see.

The perennial problem of "aspect ratios" crops up, again, at this point. That is, although the resulting window runs from *-n* to *n* in both directions, the window itself probably isn't square. The output is stretched to fit in the window, and hence may be distorted. To fix the aspect ratio, adjust either the horizontal or vertical window dimension. With enough fiddling, you can make squares look square and circles look circular.

Rotating the Camera

You can "rotate" the camera, keeping it focused on the same point but turning the camera 90 degrees, upside down, or whatever. The default views used in the *3-D Graphics Toolkit* keep the camera upright — i.e., keep the *z* axis vertical — but the *SetUp3* subroutine can change this view-up direction.

Changing the Parallel Projection Lines

There's a final trick you can play with parallel projections. The projection lines don't have to be perpendicular to the view plane. Instead, they can strike the view plane at some angle other than 90 degrees. Figure 40.8 illustrates.

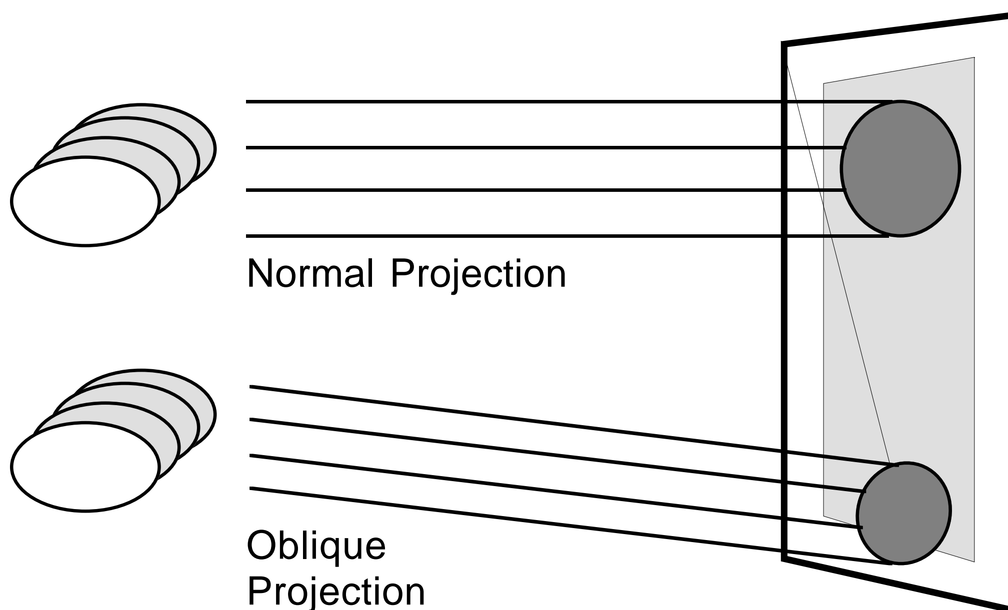

Figure 40.8. Normal and Oblique Projections.

This can be handy at times. For instance, cavalier and cabinet projections position the camera directly "in front" of an object but also show its top and sides. They do this by changing the projection lines so they no longer are perpendicular to the view plane. (See the section on "Oblique Projections" for more information.)

The *SetProj3* subroutine controls the parallel projection lines.

Wireframes vs. Hidden Surfaces

This package provides **wireframe** graphics. Hidden surfaces are not removed, so the resulting image looks like a wireframe model of the object.

More complicated graphics packages provide "hidden surface" projections as well as wireframe projections. These projections carefully remove surfaces which are hidden behind other, closer surfaces. The result looks more natural than wireframe drawings.

However, beauty has a price. Hidden surface drawings are, in general, much slower to display than wireframe drawings since the computer must do many calculations to determine which surfaces hide other surfaces. Furthermore, they require all drawing to be done in terms of polygons rather than lines, and hence are noticeably harder to use than wireframe systems.

Note: The contour-plotting subroutines do allow specialized kinds of hidden surface removal. They plot the 3-dimensional image of a data set or function by drawing bar charts or "contour lines" over the surface. These routines do remove hidden lines. See the section on "Contour Plots" for more information.

Three-Dimensional Clipping

The *3-D Graphics Toolkit* does not perform any three-dimensional clipping (although it does clip the projected image in the view plane window). Clipping is usually not needed for simple problems.

However, if you move the camera within an object, you will notice stray lines on the projected image. These are the unclipped projections of lines which lie behind the camera. The solution is simple: don't draw lines behind the camera. Increase the viewing distance so that the camera is far enough away.

Some Well-Known Projections

This section describes the older, artisan's classifications of 3-D projections in terms of modern computer graphics. Skip this section if you're not interested.

Perspective Projections

When perspective projections were drawn by hand, draftsmen used "vanishing points" to control the perspective. They classified perspective drawings by the number of vanishing points needed to draw them. You may have seen these mysterious points in books on perspective drawing. Vanishing points are no longer used with computerized graphics, but you can easily simulate them.

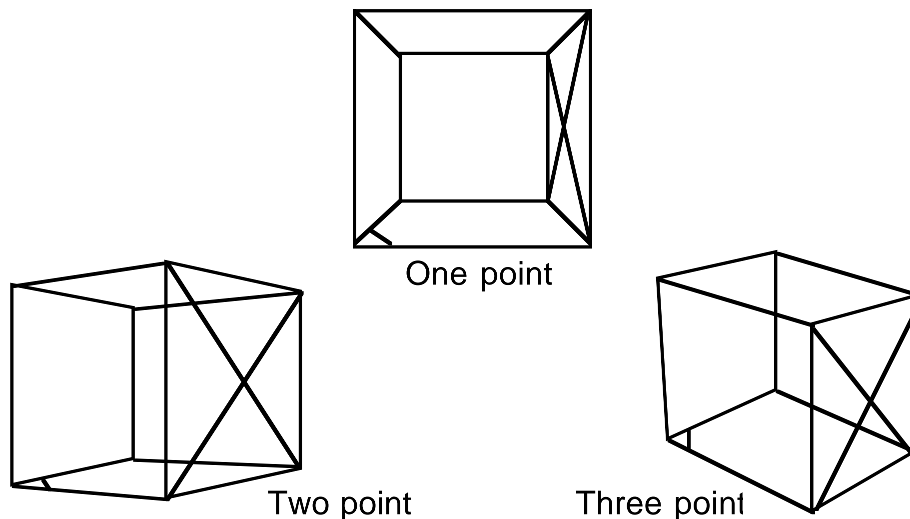

Figure 40.9: Vanishing Points in Perspective Views.

One-point perspective occurs when one of the faces of the viewing volume is parallel to the view plane: the camera position shares two coordinates with the reference point. Thus, if the reference point is *(x1, y, z)* then the camera position could be *(x2, y, z)* etc.

Two-point perspective occurs when one pair of the viewing volume's edges is parallel to the view plane (but no face). In other words, the camera position shares one coordinate with the reference point. For example, if the reference point is *(x1, y1, z)* then the camera position could be *(x2, y2, z)* etc.

Three-point perspective occurs when none of the viewing volume's edges are parallel to the view plane. The camera position shares *no* coordinates with the reference point. Thus if the reference point is *(x1, y1, z1)* then the camera position could be *(x2, y2, z2)*.

The default camera position gives three-point perspective. This is the hardest kind of view to draw by hand but the most informative — and after all, that's why we have computers.

Parallel Projections

Parallel projections can be classified as "axonometric" and "oblique" projections. Axonometric projections keep the view plane perpendicular to the projection direction. Oblique projections tilt the view plane.

The default parallel view is an axonometric (trimetric) projection.

Axonometric Parallel Projections

Axonometric projections can be most easily described by discussing their effects on a cube.

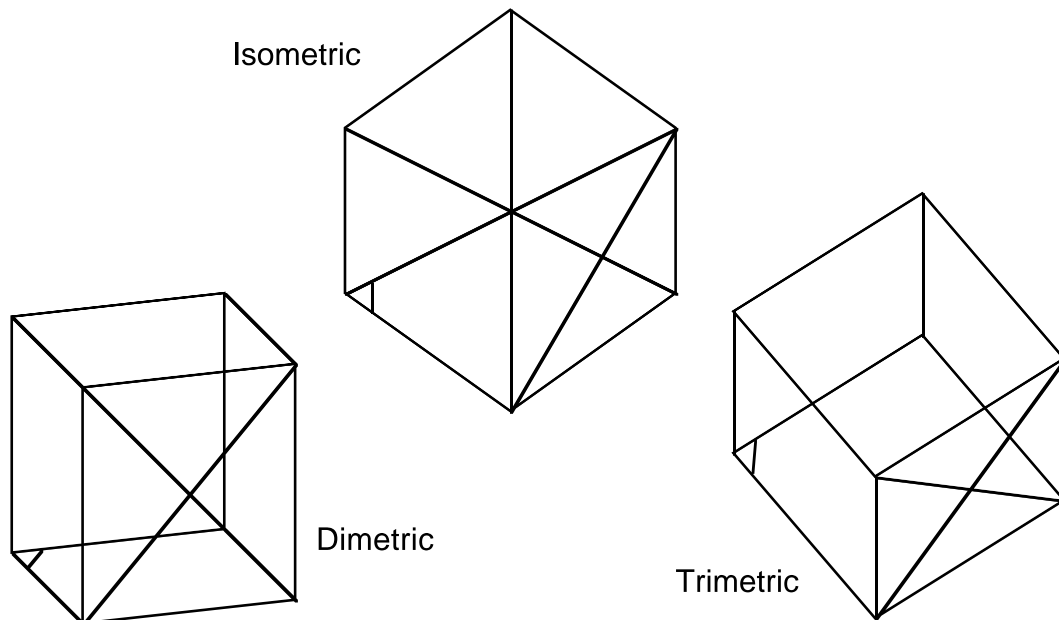

Figure 40.10: Types of Axonometric Projections.

Isometric projections shorten all edges equally. They occur when the camera is looking directly through one corner of the cube to the cube's center — the *x, y,* and *z* distances from the camera to the reference point are all equal.

Dimetric projections shorten two edges of the cube equally. In other words, two out of the three *x, y* and *z* distances from the reference point to the camera are equal.

Trimetric projections shorten each edge by a different amount. In other words, the *x, y,* and *z* distances from the reference point to the camera are all different.

Oblique Parallel Projections

Oblique projections “slant” the projection direction so that it’s no longer perpendicular to the view plane. Cavalier and cabinet projections are the two most common oblique projections; they differ only in how much the projection direction is altered.

Both change the projection direction so points farther away than the reference point are projected up and to the right. Points closer than the reference point are projected down and to the left.

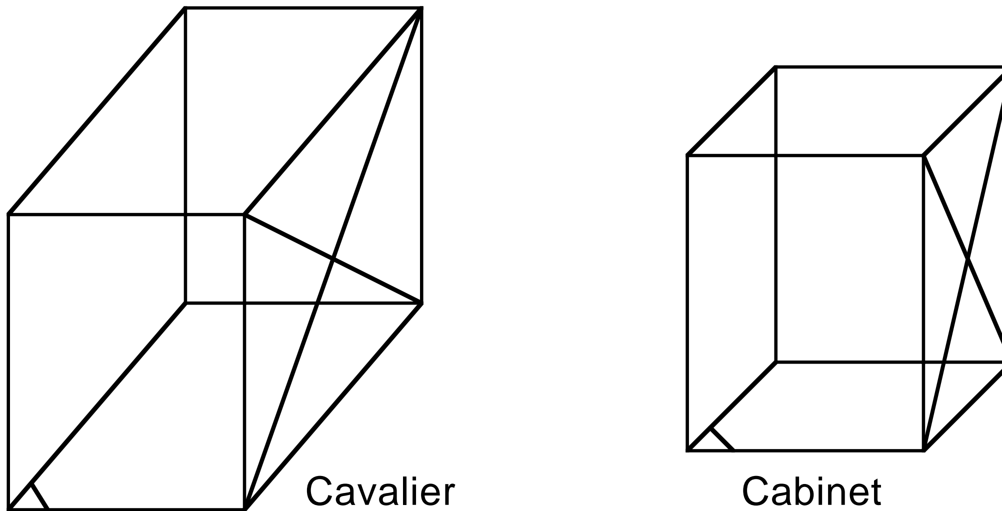

Figure 40.11: Two Common Oblique Projections.

Cavalier projections preserve “depth” distance. Suppose one point is directly behind another point. These two points will be projected to two different points on the view plane. The distance between the two projected points will exactly equal the distance between the two original points in 3-dimensional space. You can get a cavalier projection by setting a parallel viewing volume, picking a camera position, and then calling the *Cavalier3* routine.

Cabinet projections divide the “depth” distance by 2 when projecting. This looks considerably more natural than a cavalier projection, but is otherwise very similar. You can get a cabinet projection by setting a parallel viewing volume, picking a camera position, and then calling the *Cabinet3* routine.

Simple Graphics

This section describes all the routines which allow simple 3-D graphics. They let you make perspective or parallel drawings with a minimum of fuss.

PersWindow	Set perspective window.
ParaWindow	Set parallel window.
SetCamera3	Set camera position.
AskCamera3	Ask camera position.
PlotOff3	Plot point, turn beam off.
PlotOn3	Plot point, turn beam on.
PlotRoff3	Plot relative, turn beam off.
PlotRon3	Plot relative, turn beam on.
LineOn3	Plot line segment, turn beam off.
LineOff3	Plot line segment, turn beam on.
PlotText3	Plot label.
Project3	Convert 3-D to 2-D coordinates.

Default Values for a View

If you refer back to the 3-D Theory section, you'll see that proper control of the "camera" involves many different variables. The *PersWindow*, *ParaWindow*, and *SetCamera3* routines all have simple calling interfaces and provide reasonable default values for many of these variables. These default values are described for each routine. To control the variables yourself, see the Advanced Viewing section.

The Current Point and The Beam

Like True BASIC's plotting, these routines all know about the "current point." You can plot a point and leave the beam turned on; when you next plot a point, these routines will draw a line from the old current point to the new one. If you turn the beam off, it won't draw the line.

PersWindow (x1, x2, y1, y2, z1, z2)

PersWindow is like True BASIC's SET WINDOW statement. It tells what block of 3-dimensional space you want to view and sets the view reference point to be at the exact center of this window's volume. It sets up a perspective projection of the volume.

It positions the camera at a location that usually makes a clear and attractive view of the window volume. The camera sits in front of, to the side of, and above, this volume. Its exact position depends on the relative sizes of the window's *x, y* and *z* dimensions. You can use the *SetDistance3* routine to bring the camera closer to the viewing window (and thus emphasize the perspective effect), or move it farther away (and thus diminish the effect).

Furthermore, *PersWindow* points the camera at the center of the volume and sets the center of perspective to some distance directly behind the camera. This gives "natural-looking" effects. You can use the *SetViewPlane3* and *SetCP3* routines to change the camera direction and center of perspective, respectively.

Finally, *PersWindow* uses the same scale for the *x, y,* and *z* axes. See the *ScalePersWindow* routine if you want to use different scales for each dimension. (It's in the Scaled Views section.)

PersWindow doesn't clear the screen, so you must use True BASIC's CLEAR statement when you want to start a new display.

```
!  Draw the marked cube with
!  a default perspective view.
!
library "3dlib"
call PersWindow(0, 1, 0, 1, 0, 1)
call MarkedCube
end
```

Exceptions:

33 3-D window minimum = maximum.

ParaWindow (x1, x2, y1, y2, z1, z2)

ParaWindow is like True BASIC's SET WINDOW statement. It tells what block of 3-dimensional space you want to view and sets the view reference point to be at the exact center of this window's volume. It sets up a parallel projection of the volume.

Like *PersWindow*, it positions the camera at the position in front of, above, and to the side of the viewing window, and points the camera directly at the reference point. This gives rise to parallel view in which all three sides of the viewing volume are visible, with all of them having different lengths.

Finally, *ParaWindow* uses the same scale for the *x, y,* and *z* axes. See the *ScaleParaWindow* routine if you want to use different scales for each dimension. (It's in the "Scaled Views" section.)

ParaWindow doesn't clear the screen, so you must use True BASIC's CLEAR statement when you want to start a new display.

```
!  Draw the marked cube with
!  a default parallel view.
!
library "3dlib"
call ParaWindow(0, 1, 0, 1, 0, 1)
call MarkedCube
end
```

Exceptions:

33 3-D window minimum = maximum.

AskWindow3 (x1, x2, y1, y2, z1, z2)

AskWindow3 returns the current viewing window's coordinates in *x1, x2, y1, y2, z1,* and *z2*.

AskView3 (view$)

AskView3 returns either "PERSPECTIVE" or "PARALLEL" in *view$,* depending on whether the current view is a perspective view or a parallel view.

SetCamera3 (x, y, z)

SetCamera3 moves the camera to a new location (*x, y, z*). It works for both perspective and parallel projections, and does not change what's already drawn on the screen.

SetCamera3 always points the camera directly at the reference point. You can then use the *SetDistance3* routine to move the camera closer to (or farther from) the reference point; *SetDistance3* doesn't affect the camera angle.

The resulting image is always "normalized" so its size remains roughly constant. In other words, moving the camera farther from the object does not make the object smaller. To control the image size, use the *SetScale3* routine.

If you want to control the camera aim and distance independently, use the *SetViewPlane3* and *SetDistance3* routines. See the Advanced Viewing section for more information, and see the **Cube3** program for an example of *SetCamera3*.

Exceptions:

42 Viewing distance is zero.

AskCamera3 (x, y, z)

AskCamera3 returns the current camera position. It works for either parallel or perspective views.

PlotOff3 (x, y, z)

PlotOff3 moves to the point (*x, y, z*) and turns the beam off. If the beam was on, it draws a line as it moves from the previous point to the new point. Hence it's the 3-dimensional equivalent of True BASIC's statement:

```
PLOT x, y
```

You can use it either to draw an isolated point or to finish drawing a line segment. To simply turn off the beam, you can use True BASIC's empty PLOT statement.

PlotOn3 (x, y, z)

PlotOn3 moves to the point (*x, y, z*) and turns the beam on. If the beam was already on, it draws a line as it moves from the previous point to the new point. Hence it's the 3-dimensional equivalent of True BASIC's statement:

```
PLOT x, y;
```

This is the general workhorse for drawing 3-dimensional graphics since most graphs are composed from lines. To turn off the beam, you can use True BASIC's empty PLOT statement or the *PlotOff3* routine.

When you draw a complicated object, with many line segments, you may want to use *MatLines3* rather than many calls to *PlotOn3*.

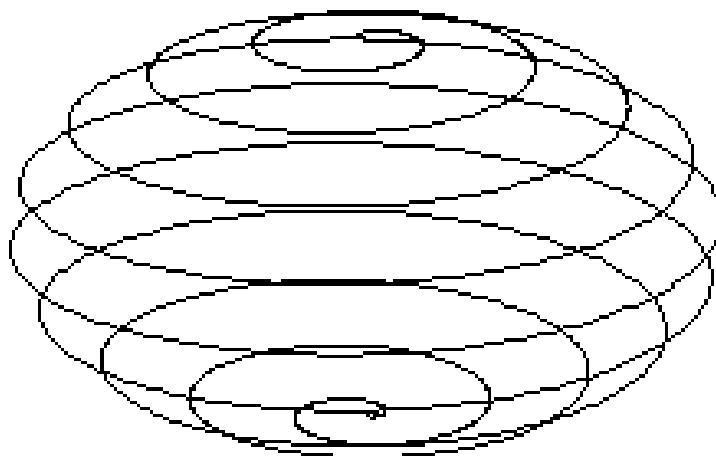

Figure 40.12: Simple 3-D Graphics — The SPIRAL3 Program.

PlotRoff3 (dx, dy, dz)

PlotRoff3 moves (*dx, dy, dz*) units away from the current point and turns off the beam. If the beam was on before this routine was called, it draws a line as it moves to the point.

PlotRon3 (dx, dy, dz)

PlotRon3 moves (*dx, dy, dz*) units away from the current point and turns on the beam. If the beam was on before this routine was called, it draws a line as it moves to the new point.

LineOn3 (x1, y1, z1, x2, y2, z2)

LineOn3 draws a line segment from (*x1, y1, z1*) to (*x2, y2, z2*) and leaves the beam turned on. You could do the same thing by calling *PlotOn3* twice but this is more convenient.

LineOff3 (x1, y1, z1, x2, y2, z2)

LineOff3 draws a line segment from (*x1, y1, z1*) to (*x2, y2, z2*) and turns the beam off. You could do the same thing by calling *PlotOn3* and then *PlotOff3* but this is more convenient.

PlotText3 (x, y, z, text$)

PlotText3 works like True BASIC's PLOT TEXT statement. It prints the text with its anchor point (lower, left corner) placed at the point (*x, y, z*). You can use True BASIC's SET TEXT JUSTIFY statement to change the anchor point's position.

The text itself is not affected by perspective or the viewing angle. It always appears normal and two-dimensional.

Project3 (x, y, z, flatx, flaty)

Project3 converts a 3-dimensional (*x, y, z*) coordinate to the corresponding flat True BASIC window coordinate (*flatx, flaty*).

This is useful if you want to mix 2- and 3-dimensional graphics. For instance, it makes it easy to draw a circle around a given point in a three dimensional space to illustrate its position.

Advanced Drawing Routines

This section contains the advanced drawing routines. These routines are usually not needed for graphics work but can prove handy from time to time.

MatPoints3	Plot array of points.
MatLines3	Plot array of line segments.
MatArea3	Plot polygon surface.
MatPlot3	Plot segments and/or points.
MatProject3	Project array of points.
Frame3	Draw outline of current window.
Box3	Draw outline of "block."
Block3	Draw shaded "block" volume.
Axes3	Draw axes.
Ticks3	Draw axes with tick marks.
AxesSub3	Low-level axes routine.
UnitCube	Draw unit cube outline.
MarkedCube	Draw marked unit cube outline.
RectX3	Draw rectangle in x plane.
RectY3	Draw rectangle in y plane.
RectZ3	Draw rectangle in z plane.
FillRectX3	Draw filled rectangle in x plane.
FillRectY3	Draw filled rectangle in y plane.
FillRectZ3	Draw filled rectangle in z plane.
CircleX3	Draw ellipse in x plane.
CircleY3	Draw ellipse in y plane.
CircleZ3	Draw ellipse in z plane.
FillCircleX3	Draw filled ellipse in x plane.
FillCircleY3	Draw filled ellipse in y plane.
FillCircleZ3	Draw filled ellipse in z plane.
GridX3	Draw grid in x plane.
GridY3	Draw grid in y plane.
GridZ3	Draw grid in z plane.

MatPoints3 (pts(,))

MatPoints3 is like True BASIC's MAT PLOT points statement except that it works in 3 dimensions. It draws a series of points, each of which is saved in the *pts(,)* array.

The array may have any lower bound in either dimension but it must have exactly three elements across — they're used as the *x, y,* and *z* coordinates of the points.

Exceptions:

32 Bad 3-D graphics MAT points array.

MatLines3 (pts(,))

MatLines3 is like True BASIC's MAT PLOT LINES statement, except that it works in 3 dimensions. It draws a series of line segments between the points saved in the *pts(,)* array and turns off the beam when done.

The array may have any lower bound in either dimension, but it must have exactly three elements across — they're used as the *x, y,* and *z* coordinates of the points.

When you draw a complicated object with many line segments, you may want to use *MatLines3* rather than many calls to *PlotOn3*.

Exceptions:

32 Bad 3-D graphics MAT points array.

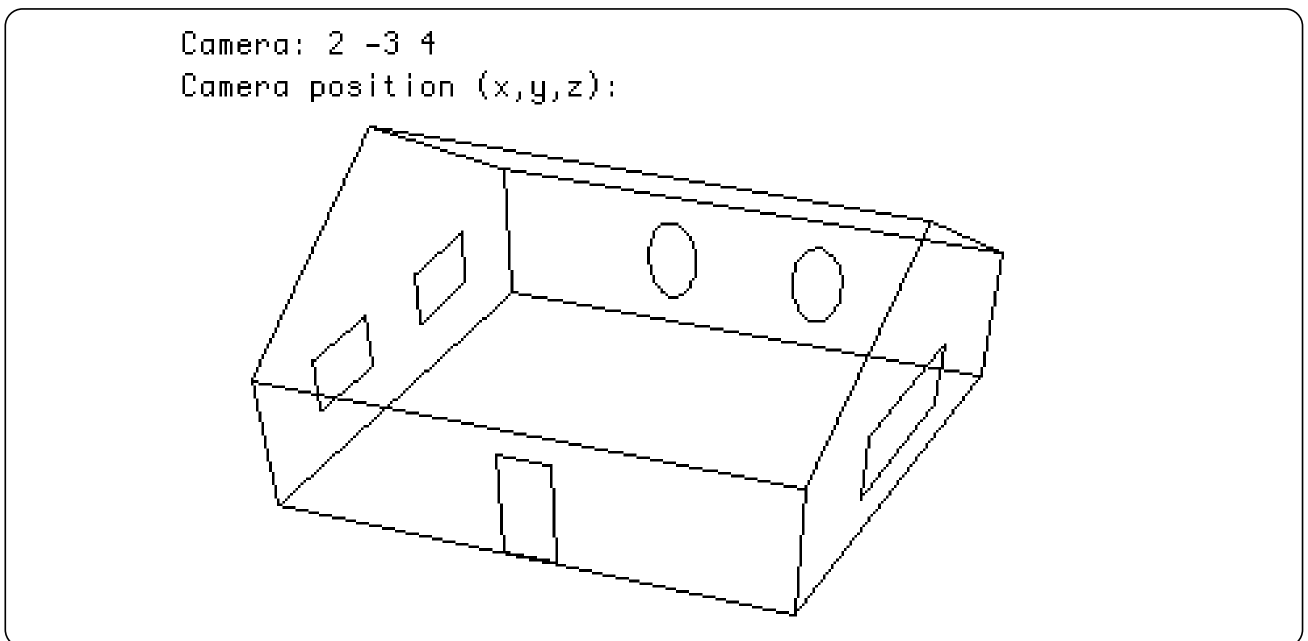

Figure 40.13: MatLines3, RectX3, CircleX3— The HOUSE3 Program.

MatArea3 (pts(,))

MatArea3 is like True BASIC's MAT PLOT AREA statement, except that it works in 3 dimensions. It draws the edges of the polygon defined by the points in the *pts(,)* array and then fills in the area of this polygon in the current color.

The array may have any lower bound in either dimension, but it must have exactly three elements across — they're used as the *x, y,* and *z* coordinates of the points.

Exceptions:

32 Bad 3-D graphics MAT points array.

MatPlot3 (pts(,))

MatPlot3 draws one or more points or line segments. The *pts(,)* array contains a series of 3-D points; and for each point, it also contains a flag saying whether the beam should be turned on or off after that point.

The array may have any lower bound in either dimension, but it must have exactly four elements across — the first three are used as the *x, y,* and *z* coordinates of a point, and the last is used as the beam flag. If the flag is zero, the beam is turned off after that point; otherwise, it's turned on.

```
!  Draw an "X" shape.
!
library "3dlib"
dim pts(4,4)
call PersWindow(-2, 2, -2, 2, -2, 2)
mat read pts
data -2,-2,-2,1, 2,2,2,0, -2,2,-2,1, 2,-2,-2,0
call MatPlot3(pts)
end
```

Exceptions:

32 Bad 3-D graphics MAT points array.

MatProject3 (pts(,))

MatProject3 finds the projected images of one or more points. The *pts(,)* array contains a series of 3-D points. For each point, *MatProject3* returns the corresponding flat True BASIC window coordinate. Thus it's a version of *Project3* which works on a number of points in one operation.

The array may have any lower bound in either dimension, but it must have exactly five elements across — the first three are used as the *x, y,* and *z* coordinates of a point. The last two are the resulting *flatx* and *flaty* image coordinates.

Exceptions:

32 Bad 3-D graphics MAT points array.

Frame3

Frame3 draws the outline of your current viewing volume. It's often useful for checking your current camera position to make sure that everything is visible and neatly arranged.

Box3 (x1, x2, y1, y2, z1, z2)

Box3 draws the outline of the 3-dimensional block specified by the given coordinates.

Block3 (x1, x2, y1, y2, z1, z2, col1$, col2$, col3$, frame)

Block3 draws a "shaded" block in 3 dimensions. This is suitable for histogram blocks and so forth.

The block's edges are defined by the *x, y,* and *z* coordinates. Then the routine draws the visible sides of the blocks using *col1$, col2$,* and *col3$* as the colors for each of the three visible sides. These color strings can be True BASIC colors, such as "red" or "yellow", or can represent color numbers such as "1" or "25." They don't affect the current color.

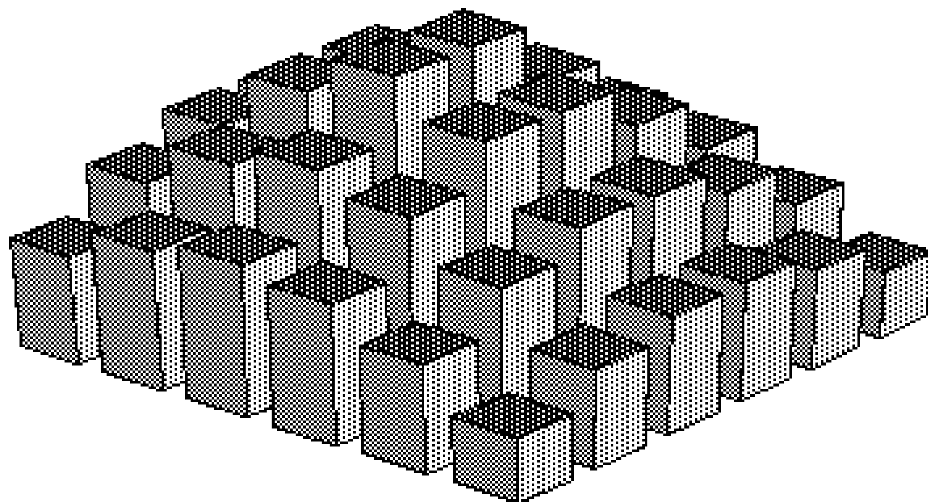

Figure 40.14: A Collection of Blocks — The BLOCKS3 Program.

If *frame* is nonzero, the routine finishes by tracing the block's edges with lines drawn in the current color. See the **Blocks3** program for examples.

Exceptions:

11008	No such color: *xxx*
-11008	On DOS systems

Axes3

Axes3 draws the *x-y-z* axes in your current window. It does not draw any "tick marks" on the axes. Use *Ticks3* to get tick marks.

Ticks3 (xi, yi, zi)

Ticks3 draws the axes in your current window with tick marks at *xi, yi*, and *zi* units apart. If any of the tick mark increments is zero, that axis will be drawn without ticks. Negative increments are treated exactly like positive increments.

If the tick marks are too large or too small for your taste, change the ticksize variable in the *AxesSub3* routine. Make it bigger for bigger ticks, and smaller for smaller ticks. See the **Axes3** program for an example.

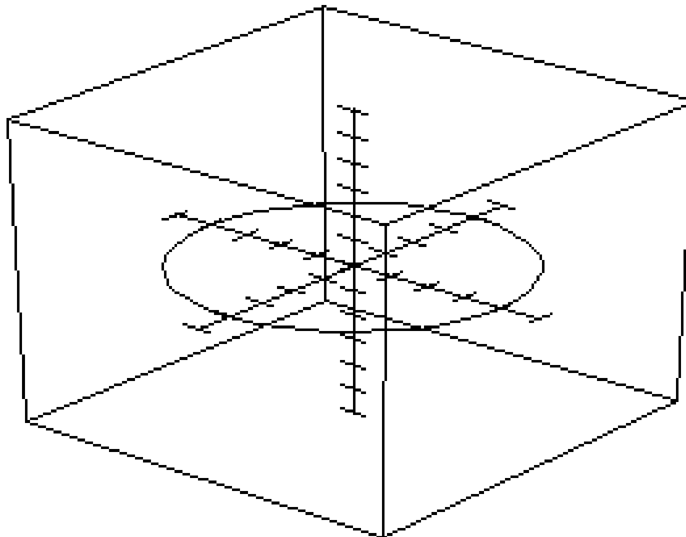

Figure 40.15: Unit Cube with Ticks and Circle — The AXES3 Program.

AxesSub3 (x1, x2, y1, y2, z1, z2, xi, yi, zi)

AxesSub3 is a low-level subroutine used by the *Axes3* and *Ticks3* routines. You will probably not need to use it. However, it is slightly more flexible than the other two routines.

AxesSub3 draws tick marks along portions of the *x-y-z* axes. Tick marks are placed between *x1* and *x2* along the *x*-axis, spaced *xi* units apart. (The other two axes are treated similarly.) If *x1* = *x2,* that axis is not drawn. If *xi* = 0, no tick marks are drawn.

```
!  Draw the X-Y axes, but not the Z axis.
!
library "3dlib"
call PersWindow(-10, 10, -10, 10, -10, 10)
call AxesSub3(-10, 10, -10, 10, 0, 0, 2, 2, 0)
call UnitCube
end
```

UnitCube

UnitCube draws the "unit cube," that is, the cube whose opposite corners are at the origin (0,0,0) and (1,1,1).

It's handy for calibrating your view if you start to adjust the viewing parameters. However, you may find the *MarkedCube* routine (below) more useful.

MarkedCube

MarkedCube is like *UnitCube* except that it has a small diagonal line cutting across the *x-y* plane's corner just past the origin. It also has diagonal lines making an X across the face where *x* = 1. This makes it easier to see which way the cube is lying, in case you are completely confused as to which direction is upwards, etc. (Which is often the case!)

The CUBE3 program, on your disk, shows off *MarkedCube*.

```
!  Show the marked cube in an
!  easy-to-see angle.
!
library "3dlib"
call PersWindow(0, 1, 0, 1, 0, 1)
call SetCamera3(3, 4, 5)
call MarkedCube
end
```

RectX3 (x, y1, y2, z1, z2)

RectX3 draws a two-dimensional rectangle perpendicular to the *x* axis. That is, the rectangle's corners are (*x, y1, z1*) and (*x, y2, z2*).

The **House3** program uses *RectX3* to draw windows on the side of the house.

RectY3 (y, x1, x2, z1, z2)

RectY3 draws a two-dimensional rectangle perpendicular to the *y* axis. That is, the rectangle's corners are (*x1, y, z1*) and (*x2, y, z2*).

RectZ3 (z, x1, x2, y1, y2)

RectZ3 draws a two-dimensional rectangle perpendicular to the *z* axis. That is, the rectangle's corners are (*x1, y1, z*) and (*x2, y2, z*).

FillRectX3 (x, y1, y2, z1, z2)

FillRectX3 is like *RectX3* except that it colors the entire area of the rectangle rather than drawing the outline.

FillRectY3 (y, x1, x2, z1, z2)

FillRectY3 is like *RectY3* except that it colors the entire area of the rectangle rather than drawing the outline.

FillRectZ3 (z, x1, x2, y1, y2)

FillRectZ3 is like *RectZ3* except that it colors the entire area of the rectangle rather than drawing the outline.

CircleX3 (x, y1, y2, z1, z2)

CircleX3 draws a two-dimensional ellipse perpendicular to the *x* axis. That is, the ellipse is inscribed within a rectangle whose corners are (*x, y1, z1*) and (*x, y2, z2*). It's similar to True BASIC's BOX CIRCLE statement.

```
!  Draw unit cube with circle
!  inscribed on each side.
!
library "3dlib"
call PersWindow(0, 1, 0, 1, 0, 1)
call UnitCube

for side = 0 to 1
    call CircleX3(side, .2, .8, .2, .8)
    call CircleY3(side, .2, .8, .2, .8)
    call CircleZ3(side, .2, .8, .2, .8)
next side
end
```

CircleY3 (y, x1, x2, z1, z2)

CircleY3 draws a two-dimensional ellipse perpendicular to the *y* axis. That is, the ellipse is inscribed within a rectangle whose corners are (*x1, y, z1*) and (*x2, y, z2*). It's similar to True BASIC'sBOX CIRCLE statement.

CircleZ3 (z, x1, x2, y1, y2)

CircleZ3 draws a two-dimensional ellipse perpendicular to the *z* axis. That is, the ellipse is inscribed within a rectangle whose corners are (*x1, y1, z*) and (*x2, y2, z*). It's similar to True BASIC's BOX CIRCLE statement.

FillCircleX3 (x, y1, y2, z1, z2)

FillCircleX3 is like *CircleX3* except that it colors the entire area of the ellipse rather than drawing the outline.

FillCircleY3 (y, x1, x2, z1, z2)

FillCircleY3 is like *CircleY3* except that it colors the entire area of the ellipse rather than drawing the outline.

FillCircleZ3 (z, x1, x2, y1, y2)

FillCircleZ3 is like *CircleZ3* except that it colors the entire area of the ellipse rather than drawing the outline.

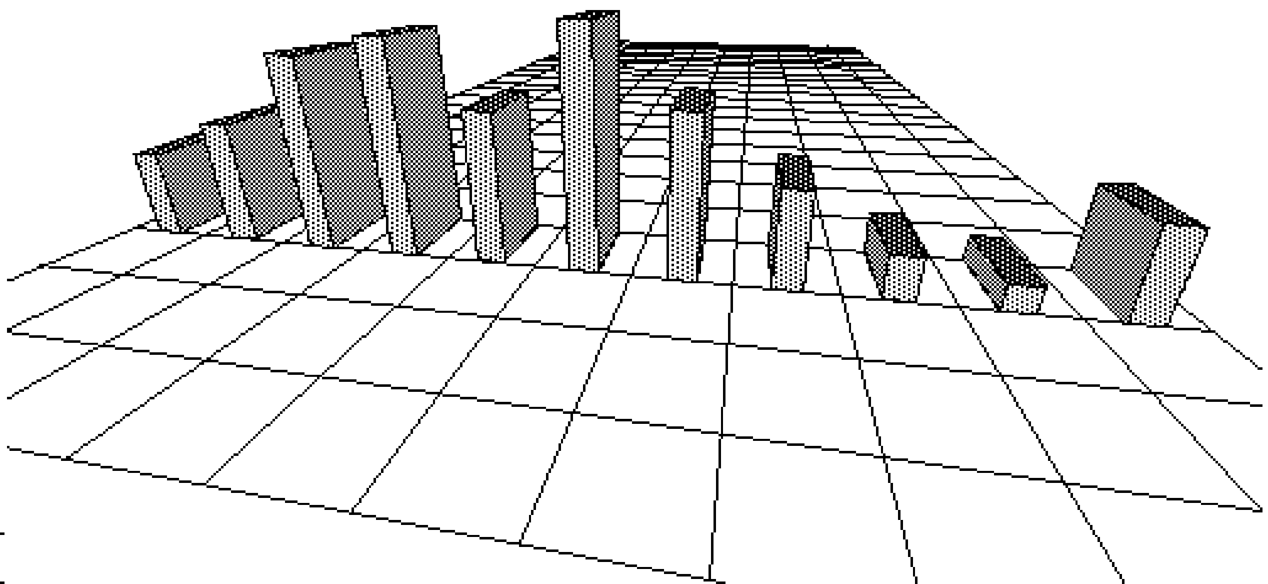

Figure 40.16: Blocks on a Grid — The BARS3 Program.

GridX3 (x, y1, y2, z1, z2, ystep, zstep)

GridX3 draws a two-dimensional grid that's perpendicular to the *x* axis. That is, the grid is drawn over a rectangle whose corners are (*x, y1, z1*) and (*x, y2, z2*). The *ystep* and *zstep* parameters control the spacing of the grid lines in the *y* and *z* directions.

GridY3 (y, x1, x2, z1, z2, xstep, zstep)

GridY3 draws a two-dimensional grid that's perpendicular to the *y* axis. That is, the grid is drawn over a rectangle whose corners are (*x1, y, z1*) and (*x2, y, z2*). The *xstep* and *zstep* parameters control the spacing of the grid lines in the *x* and *z* directions.

GridZ3 (z, x1, x2, y1, y2, xstep, ystep)

GridZ3 draws a two-dimensional grid that's perpendicular to the *z* axis. That is, the grid is drawn over a rectangle whose corners are (*x1, y1, z*) and (*x2, y2, z*). The *xstep* and *ystep* parameters control the spacing of the grid lines in the *x* and *y* directions. See the **Bars3** program for an example.

Contour Plots

This section describes specialized routines that draw contour graphs of 3-D data sets or of "*z* functions" of two variables $z = F(x,y)$. These routines draw the *z* surface of the data set or function by drawing blocks or following contour lines along the *x* and/or *y* directions.

Note: *You must include a* **LIBRARY "3dcont"** *statement in any program that uses one of these routines.*

Zmesh	Plot *z* function surface with a mesh.
ZmeshData	Plot data-set surface with a mesh.
Zplot	Plot *z* function surface, hiding lines.
ZplotData	Plot data surface, hiding lines.
Splot	Plot *z* function surface, clipping.
Zbar	Plot *z* function surface by bar chart.
ZbarData	Plot data surface by bar chart.
Tplot	Plot topographic map of *z* function.
TplotData	Plot topographic map of data set.
ZplotRect	Plot *z* function surface over *x-y* rectangle.
ZmeshRect	Plot *z* function mesh over *x-y* rectangle.
ZplotSub	Lower level routine for Zplot.
SetMesh3	Set number of divisions in mesh.
SetZlines3	Set number of contour lines.
SetZseg3	Set number of segments per contour line.
SetTlines3	Set number of lines in topo plot.
SetBarSize3	Set "footprint" of 3-D bars.
SetBarColor3	Set color scheme for 3-D bars.
SetColor3	Set color scales for contour plot or topo plot.
AskMesh3	Ask number of divisions in mesh.
AskZlines3	Ask number of contour lines.
AskZseg3	Ask number of segments per contour line.
AskTlines3	Set number of lines in topo plot.
AskBarSize3	Set "footprint" of 3-D bars.
AskBarColor3	Set color scheme for 3-D bars.
AskColor3	Ask color scales for contour plot or topo plot.

Plotting a Function F(x,y) or Data Set

To plot the surface of a function, you must define an external function F(*x,y*) that returns a numeric result. *Zmesh*, *Zplot*, etc., will graph this function over an *x-y* rectangle. To graph several functions, use a PUBLIC variable to communicate between your main program and *F*, which can compute different functions based on its value.

To graph a data set, put your data into a two-dimensional array. The first dimension is the *x* direction; the second is the *y* direction. The array will be drawn as data points evenly spaced over an *x*/*y* rectangle. Each element gives the height at the corresponding point.

Adjusting the Contour Lines

Zplot displays a surface by drawing "contour lines" along the surface. *SetMesh3* controls the number of lines for drawing mesh plots. *SetZlines3* sets the number of contour lines for *Zplot*, and *SetZseg3* sets the number of segments in each contour line. Figure 40.17 shows contour lines and segments for the **Zplot3** sample program.

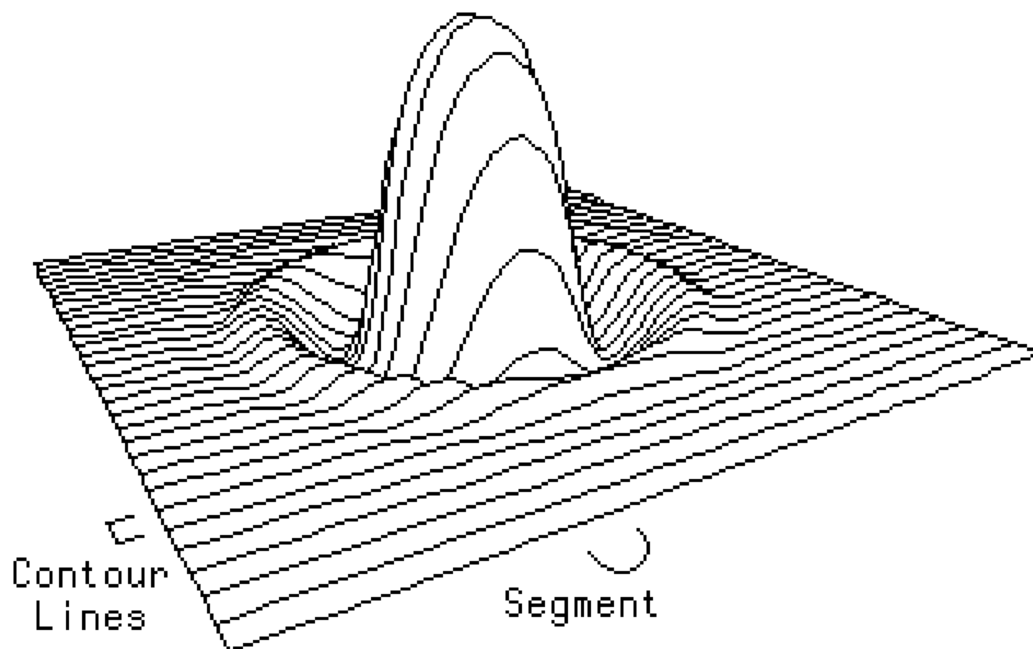

Figure 40.17: Adjusting Contour Lines.

Zmesh

Zmesh draws a graph of the function F(*x,y*) by drawing contour lines along *x* and *y* directions. You must define an external numeric function F(*x,y*). *Zmesh* will draw this function's value over *x* values ranging from the viewing volume's *xmin* to *xmax* and *y* values ranging from *ymin* to *ymax*.

Zmesh will use the current viewing volume, camera angle, viewing distance, and so forth, when graphing the function. You may use it with any kind of perspective or parallel view. Use *SetMesh3* to adjust the fineness of the covering mesh.

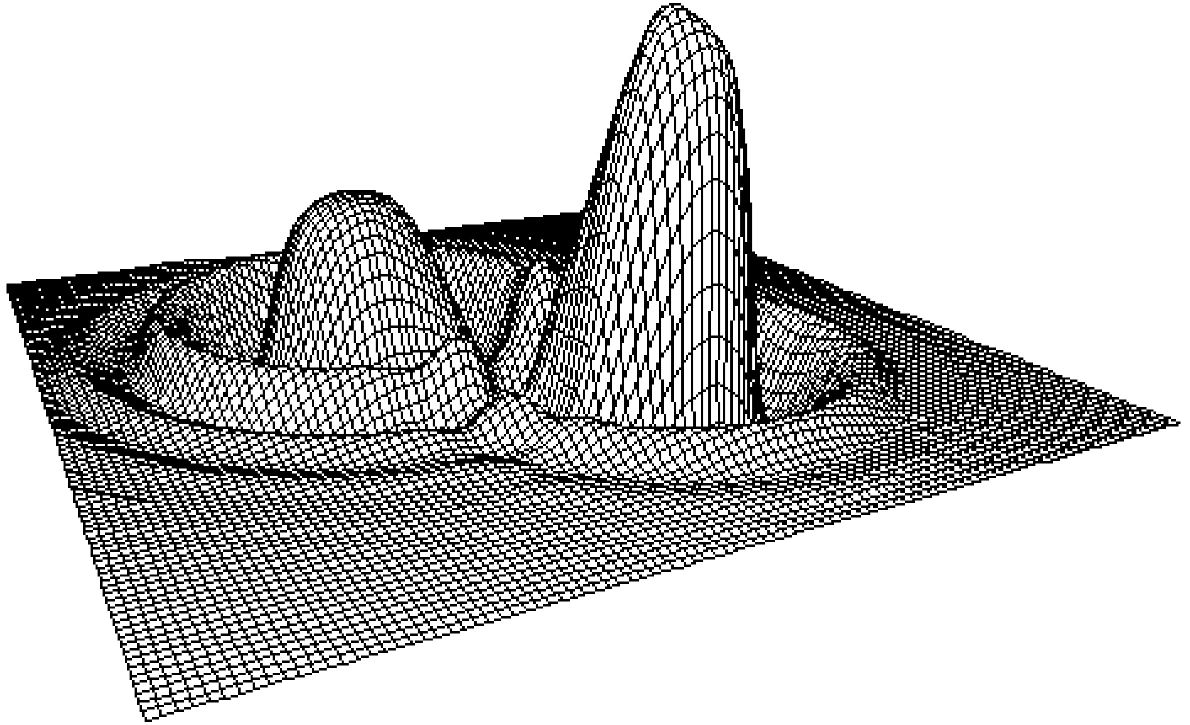

Figure 40.18: A Function Plotted with Zmesh — The Zmesh3 *Program.*

ZmeshRect (xmin, xmax, ymin, ymax)

ZmeshRect is like *Zmesh* except that you supply the *xmin, xmax, ymin,* and *ymax* rectangle instead of using the viewing volume's *x-y* range.

Zplot

Zplot draws a graph of the function F(*x,y*) by drawing contour lines along either the *x* or *y* direction. You must define an external numeric function F(*x,y*). Zplot will draw this function's value over *x* values ranging from the viewing volume's *xmin* to *xmax* and *y* values ranging from *ymin* to *ymax*.

Zplot will use the current viewing volume, camera angle, viewing distance, and so forth, when graphing the function. Use it with any kind of perspective or parallel view.

Zplot will choose to run contour lines along either the *x* or *y* direction, depending on which looks best. It prefers to use contours parallel to the *x* axis but will use *y* contours when the view runs "down" the *x* direction (i.e., more or less parallel to the *x* axis). See Figure 40.17 for a sample image.

```
!  A simple Zplot.
!
library "3dlib"
call PersWindow(-pi, pi, -pi, pi, -1, 1)
call Zplot
end

def F(x, y) = Sin(Sqr(x^2+y^2))
```

ZplotRect (xmin, xmax, ymin, ymax)

ZplotRect is like *Zplot* except that you supply the *xmin, xmax, ymin,* and *ymax* rectangle instead of using the viewing volume's *x-y* range.

ZplotSub (xmin, xmax, ymin, ymax, xy, nlines, step)

ZplotSub is a subroutine used by *Zplot* and *ZplotRect*. It's more general than *Zplot* but harder to use. You probably won't want to use it.

Like *Zplot*, *ZplotSub* graphs a function F(*x,y*) where the *x* values range from *xmin* to *xmax* and the *y* values range from *ymin* to *ymax*. The *xy* parameter controls the contour lines: *xy*=0 means the contour lines parallel the *x* axis, *xy*=1 means they parallel the *y* axis.

Nlines controls the number of contour lines. *Step* controls how many line segments are used to draw each contour line. As *step* gets bigger, the graph gets more accurate but slower. As it gets smaller, the graph looks more jagged but is faster.

Warning: ZplotSub does not always draw the contour lines in the right order! It ranges both the *x* and *y* values from their minimums to their maximums but this order is wrong for many views. The output will look peculiar. Furthermore, it does not correctly handle some cases where the view looks "down" the contour lines — i.e., runs more or less parallel to the contours. For these reasons, you should probably stick to the *Zplot* routine.

```
!  A simple ZplotSub.
!
library "3dlib"
call PersWindow(-pi, pi, -pi, pi, -1, 1)
call SetCamera3(0, -5, 3)
call ZplotSub(0, -pi, pi, -pi, pi, 0, .5, 25)
end

def F(x, y) = Sin(Sqr(x^2+y^2))
```

Splot

Splot draws the surface of a function F(*x,y*) running the values of *x* and *y* from the viewing volume's *xmin* to *xmax* and *ymin* to *ymax*.

It ignores points which do not have values, i.e., those for which F(*x,y*) causes an exception. And it "clips" contour lines that extend outside of the viewing volume.

This makes *Splot* suitable for viewing functions such as half spheres, which are not defined over an entire rectangle, and parabolic functions, which are hard to understand if not clipped to the viewing volume.

Unlike *Zplot*, *Splot* does not hide hidden surfaces.

```
!  A simple Splot on a half-sphere.
!
library "3dlib"
call PersWindow(-2, 2, -2, 2, 0, 1)
call Splot
end

def F(x, y) = Sqr(4 - (x^2+y^2))
```

Zplot versus Splot

The program **Zsplot3**, on your disk, shows how you can use either *Zplot* or *Splot* to graph a "tricky" function. Figure 40.19 shows the output for both an equation of a half-sphere, and one of a quadratic surface.

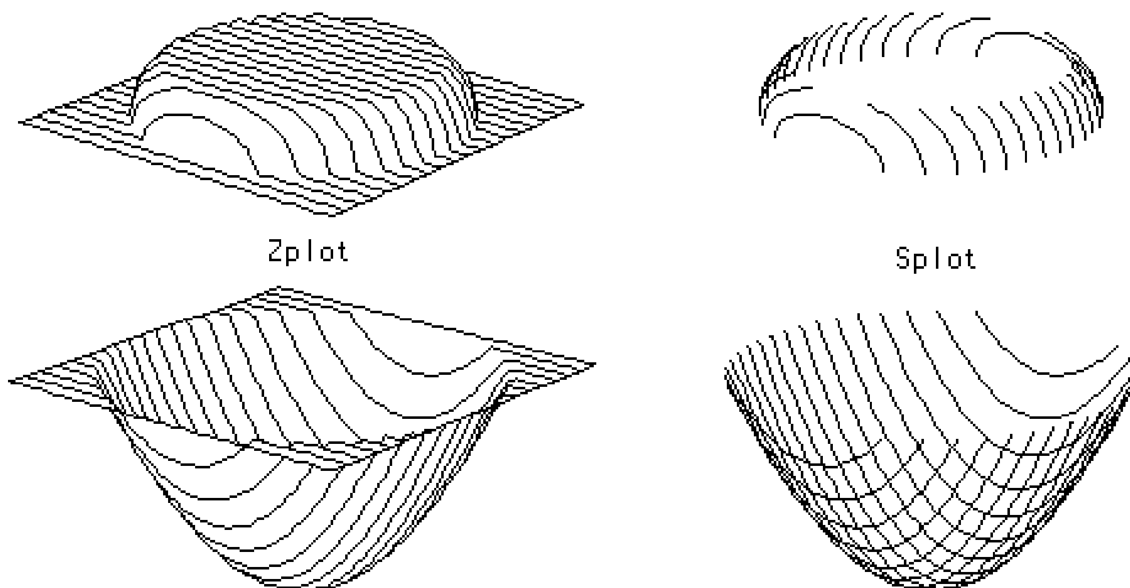

Figure 40.19. Zplot vs. Splot on some Tricky Cases.

Both functions are clipped to the viewing volume; points outside the 3D window are not shown. The half-sphere equation has many undefined points – those off the sphere's surface.

Splot automatically handles clipping and undefined points. You can also plot such surfaces with *Zplot,* provided that you carefully write the function F so it clips points and handles the undefined points.

Zsplot3 uses a function *F* which can draw 4 different functions. Two of them, public *fn* = 0 or 2, are used for calls from *Zplot*. Notice how the function with *fn* = 0 defines a value of 0 for all undefined points on the sphere and clips *z* values greater than 1. If you don't take this special care, F would give errors with "Square root of a negative number" and would draw function surfaces extending outside the viewing volume. F with *fn* = 2 also clips *z* values.

Sometimes *Zplot* works better and sometimes *Splot* works better. Use whichever you think works best for your problem.

Graphing Multiple Functions

The short program below shows how you can use a PUBLIC variable to plot several different functions in one program:

```
!  Show how to graph multiple functions.
!
library "3Dcont"
public fn

open #1: screen 0, .5, .1, .9
let fn = 1
call PersWindow(-pi, pi, -pi, pi, -1, 1)
call Zplot

open #2: screen .5, 1, .1, .9
let fn = 2
call PersWindow(-pi, pi, -pi, pi, -1, 1)
call Zplot
end

def F(x, y)
    declare public fn
    if fn = 1 then
       let F = sin(x+y)
    else
       let F = sin(x)^2+cos(y)^2
    end if
end def
```

For another example, see the **Zsplot3** program on your disk. It draws four different functions in four windows.

Tplot (zplane)

Tplot draws a "topographic map" of the function F(*x,y*). This map is flat — it's a two-dimensional projection of the function, as seen from above. You must define the external numeric function F(*x,y*).

Tplot will draw this function's value over the *x* values ranging from viewing volume's *xmin* to *xmax* and *y* values ranging from *ymin* to *ymax*. F is sampled at a rectangular mesh of points; use *SetMesh3* to control the number of points in the mesh.

The flat map is drawn with its *z* coordinate at *zplane*.

Tplot uses the current viewing volume, camera angle, distance, and so on, when graphing. You can use it with perspective or parallel views. See the **Topo3** program for an example.

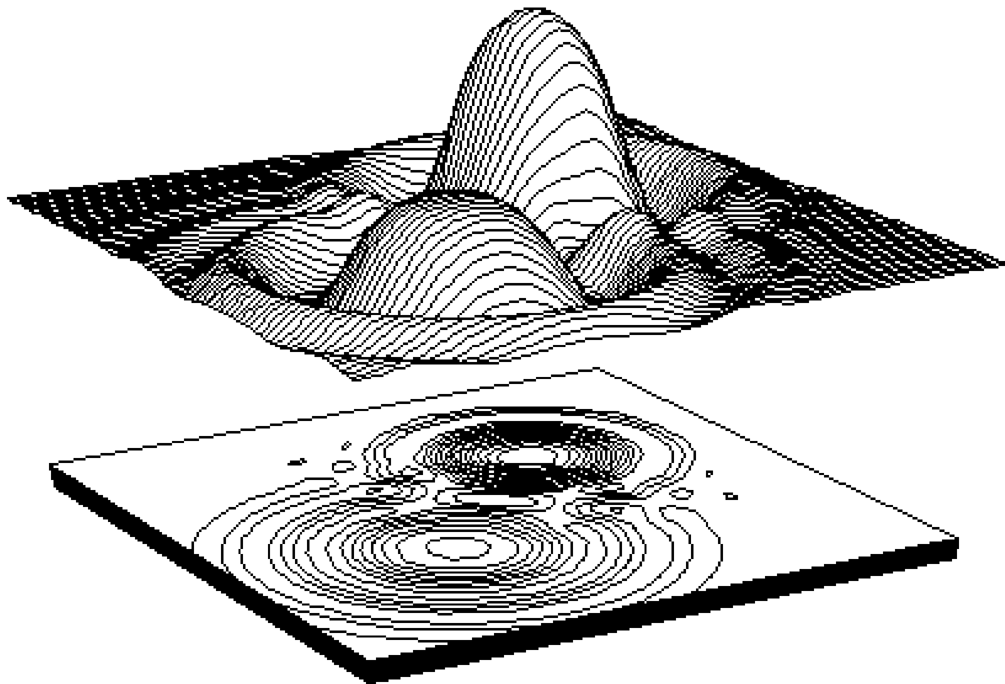

Figure 40.20. A Topographic Map with Zplot — The Topo3 *Program.*

Zbar

Zbar is like *Zplot* or *Zmesh*, but plots the function values as a 3-D bar chart. *SetMesh3* controls the number of bars used for the chart.

You can control the colors and shapes of the bars by *SetBarColor3* and *SetBarSize3*. See their descriptions for more detail.

Zbar does not usually give very interesting graphs; it's included mainly for the sake of completeness. The corresponding routine for graphing data, *ZbarData*, is more useful.

ZmeshData (data(,), x1, x2, y1, y2)

ZmeshData is like *Zmesh* but plots the image of data points instead of a function's surface. You must pass a collection of *z* values in the *data(,)* array. They will be drawn as points evenly spaced over the interval *x1* to *x2* and *y1* to *y2*.

In general, you should have at least 20 elements in each dimension of data for the Z-mesh to look good.

ZplotData (data(,), x1, x2, y1, y2)

ZplotData is like *Zplot* but plots the image of data points instead of a function's surface. You must pass a collection of *z* values in the *data(,)* array. They will be drawn as points evenly spaced over the interval *x1* to *x2* and *y1* to *y2*.

In general, you should have at least 20 elements in each dimension of data for the Z-plot to look good.

ZbarData (data(,), x1, x2, y1, y2)

ZbarData draws a 3-D bar chart of the *data()* elements. It evenly spaces the bars over the indicated *x-y* region. The base of each bar is at *z* = 0; the height of each bar is given by the corresponding element of the *data()* array.

You can use *SetBarColor3* to control the colors of the sides and tops of the bars. Proper use of *SetBarColor3* and *SetColor3* let you get bars of different colors depending on their heights. The edges of the bars are always outlined in the current color.

You can also use *SetBarSize3* to change the "footprint" of each bar, that is, the size and shape of each bar's horizontal cross-section.

Figure 40.21 shows samples of *ZbarData* and *TplotData*.

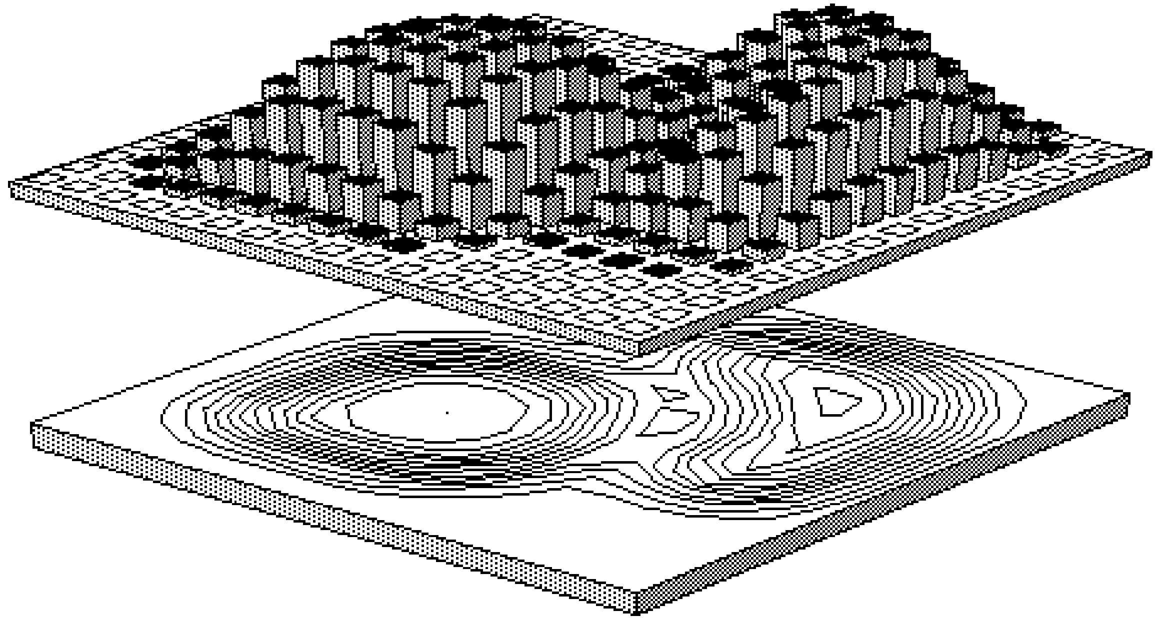

Figure 40.21. ZbarData and TplotData — The ZDATA3 Program.

TplotData (data(,), x1, x2, y1, y2, zplane)

TplotData draws the topographic map of a set of data points. You must pass a collection of *z* values in the *data(,)* array. They will be taken as points evenly spaced over the interval *x1* to *x2* and *y1* to *y2,* and the contour map will be drawn appropriately in the plane defined by $z = zplane$.

In general, you should have at least 10 elements in each dimension of data for the T-plot to look good. Figure 40.21 shows *TplotData* and *ZbarData* in action.

SetMesh3 (xmesh, ymesh)

SetMesh3 controls the number of sampling points used for *Zmesh*, *Zbar*, and *Tplot*. The *xmesh* and *ymesh* parameters control the number of sampling intervals in the *x* and *y* directions respectively.

For example, if $xmesh = 10$ and $ymesh = 15$, *TopoPlot* divides the map into 150 rectangles when drawing the contours. As these numbers get higher, your maps will get more accurate– but will take longer to compute. Values between 10 and 30 are usually a good compromise.

If either number is less than 1, *SetMesh3* uses 1 instead.

AskMesh3 (xmesh, ymesh)

AskMesh3 is the opposite of *SetMesh3*. It returns the current number of sampling intervals.

SetZlines3 (n)

SetZlines3 controls the number of contour lines used in *Zplot* or *Splot*. Larger values of *n* give finer drawings but take longer to compute and display.

If *n* is less than 2, *SetZlines3* uses 2 instead.

AskZlines3 (n)

AskZlines3 is the opposite of *SetZlines3*. It returns the current number of contour lines.

SetZseg3 (n)

SetZseg3 controls the number of line segments used to draw one contour line in *Zplot* or *Splot*. Larger values of *n* give finer drawings but take longer to compute and display.

If *n* is less than 1, *SetZseg3* uses 1 instead.

AskZseg3 (n)

AskZseg3 is the opposite of *SetZseg3*. It returns the current number of intervals used when drawing a contour line.

SetTlines3 (n)

SetTlines3 controls the number of topographic contour lines used in *Tplot*. For instance, $n = 8$ gives 8 dividing lines between the lowest and highest points on the map. Larger values of *n* give more detail but can be harder to read.

If *n* is less than 1, *SetTlines3* uses 1 instead.

AskTlines3 (n)

AskTlines3 is the opposite of *SetTlines3*. It returns the current number of topographic contour lines.

SetBarSize3 (x, y)

SetBarSize3 controls the "footprint" of the bars used for *Zbar* and *ZbarData*. The *x* and *y* parameters control the proportion of the *x-y* graphing rectangle that will be used.

For example, if $x = 1$ and $y = .1$, then the resulting bars will be long and thin. The entire *x* interval will be used, so bars will touch each other along the *x* direction. Only a small fraction of each *y* interval will be used, so the bars will be thin in that direction.

Bars sizes are preset to .6 in each direction. If either number is less than .1 or greater than 1, *SetBarSize3* will use .1 or 1 instead.

AskBarSize3 (x, y)

AskBarSize returns the current "footprint" of the bars used for *Zbar* and *ZbarData*. It's the opposite of the *SetBarSize3* routine.

SetBarColor3 (a$, b$, c$)

SetBarColor3 controls the colors used for the sides of bars drawn by *Zbar* or *ZbarData*. You pass three colors, one for each of the three visible sides.

Each color can be a name such as "red." Or it can be a number such as "3." Most flexibly, you can pass the null string as a color. Then the *Zbar* routines will use *AskColor3* to find the color for this size. Thus you can set a bar's color depending on its height.

The colors are preset to "0" so the bars are drawn in the background color. (The Zbar routines also frame the edges of the bars, so the bars are visible.)

AskBarColor3 (a$, b$, c$)

AskBarColor3 returns the current colors used for shading bars. It's the opposite of the *SetBarColor3* routine.

SetColor3(s$)

SetColor3 controls the coloring used for *Zmesh*, *Zplot*, *Splot*, and *TopoPlot*. It lets you show different altitudes in different colors. The parameter *s$* indicates the dividing lines between colors. A typical color scale might be:

```
"red [0] blue [1] green"
```

This means that all z values ≤ 0 should be shown in red. Values ≤ 1 should be shown in blue. And all other values should be shown in green. Roundoff error is always lurking in computers so you may want to fudge dividing values a little — use [0.1] instead of [0] etc.

The colors can be True BASIC color names, such as "red" or "yellow," or can be color numbers such as "1" or "25."

If you pass the null string, the function plotting routines go back to using the current color for the graph. See the **Topo3** program for an example.

Exceptions:

43 Bad color scale: *xxx*

AskColor3(s$)

AskColor3 returns in *s$* the last color scale you've used. If you haven't called *SetColor3* yet, it returns "".

Oblique Projections

This section describes how to get "oblique" projections. These are special forms of parallel projections that change the projection direction so that it's no longer perpendicular to the view plane.

Cavalier and cabinet projections are two common kinds of oblique projections, and the *3-D Graphics Toolkit* gives them special support.

Cavalier3	Set a cavalier projection.
Cabinet3	Set a cabinet projection.
Oblique3	Set an oblique projection.

Using Oblique Projections

Oblique projections are, in a sense, distortions of parallel "head-on" views. The camera faces one side of an object, but the resulting image also shows another side and the top or bottom. Figure 40.22 illustrates.

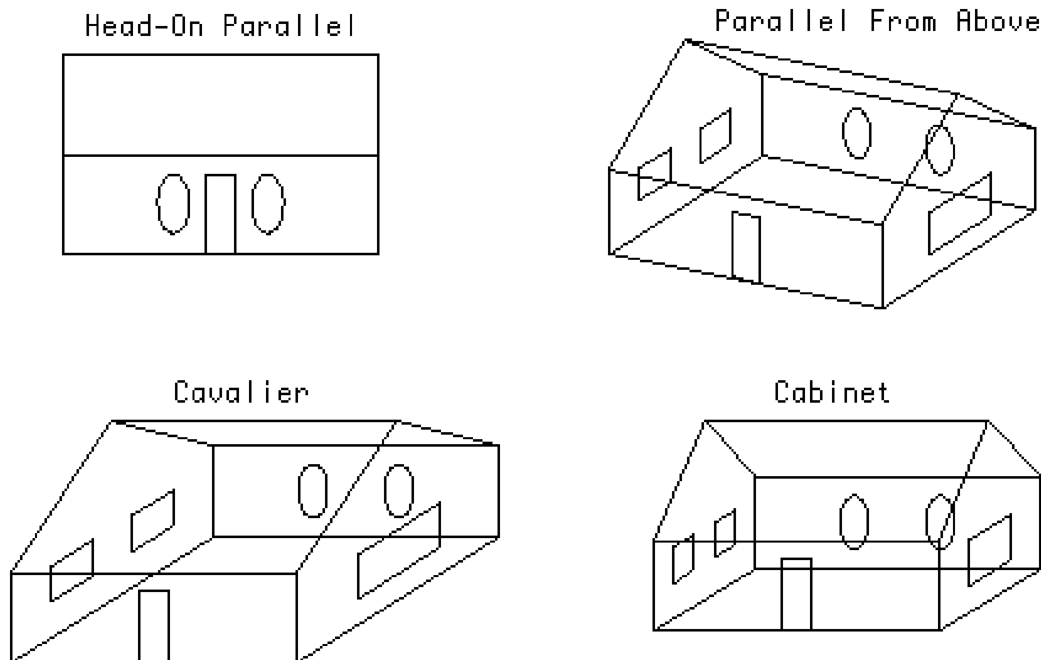

Figure 40.22. Oblique Projections — The Project3 *Program.*

Before you use any oblique projection, set up the camera so that only one face of the viewing volume is visible — that is, so that the view plane is parallel to one side of the viewing volume. Otherwise the results look ugly and are hard to decipher.

A Note on Windows...

If you refer back to Figure 40.8, you'll see that oblique projections display the image on a different part of the view plane than do "head-on" parallel projections.

The routines described in this section automatically change the view plane window so that it shows the new, oblique image. You'll notice, though, that the new window coordinates are not "nice" numbers, and that the origin no longer lies at the center of the window.

...And a Note on Aspect Ratios

Finally, all these routines assume that your current window on the screen has a 1:1 aspect ratio — that is, that a square drawn in the window would look square on the screen. In general, your current window will not have a 1:1 aspect ratio, and so angles will not come out precisely.

To correct the aspect ratio, juggle either the horizontal or vertical window coordinates until the result looks right.

Cavalier3 (angle)

Cavalier3 changes the current parallel projection direction to make a cavalier projection. The current reference point and camera position are unchanged but the projection direction is altered so that it's no longer perpendicular to the view plane.

The *angle* must be given in degrees; for example, 90 gives a right angle. (Remember that your current window's aspect ratio may distort angles; see the beginning of this section for advice.)

```
!  Show off a cavalier projection.
!
library "3dlib"
call ParaWindow(0, 1, 0, 1, 0, 1)
call Cavalier3(45)
call MarkedCube
end
```

Exceptions:

40 Can't do this with perspective view.

Cabinet3 (angle)

Cabinet3 changes the current parallel projection direction to make a cabinet projection. The current reference point and camera position are unchanged, but the projection direction is altered so that it's no longer perpendicular to the view plane. See *Cavalier3* above for information about the *angle*.

```
!  Show off a cabinet projection.
!
library "3dlib"
call ParaWindow(0, 1, 0, 1, 0, 1)
call Cabinet3(45)
call MarkedCube
end
```

Exceptions:

40 Can't do this with perspective view.

Oblique3 (angle, xratio, yratio)

Oblique3 gives access to a general oblique projection. As with the *Cavalier3* and *Cabinet3* routines, you must pass the *angle* for the oblique projection (in degrees). You must also pass an *xratio* and *yratio* which control how the *z* dimension appears to be shifted on the view plane.

Cavalier3 passes 1 for both the *xratio* and *yratio*. *Cabinet3* passes 1/2 for both. In general, the bigger you make the ratios, the "deeper" the result looks. The smaller you make them, the "shallower" it looks. The *xratio* controls the horizontal shifting of depth; the *yratio* controls its vertical shifting.

(Remember that your current window's aspect ratio may distort angles; see the beginning of this section for advice.) The angle will also be distorted if you give different numbers for *xratio* and *yratio*.

```
!  Show a shallow marked cube.
!
library "3dlib"
call ParaWindow(0, 1, 0, 1, 0, 1)
call Oblique3(45, .3, .3)
call MarkedCube
end
```

Exceptions:

40 Can't do this with perspective view.

Scaled Views

These routines create scaled perspective and parallel windows.

ScalePersWindow	Create scaled perspective window.
ScaleParaWindow	Create scaled parallel window.

What is Scaling?

The *PersWindow* and *ParaWindow* routines, described in earlier sections, treat each of the *x-y-z* dimensions equally. That is, if you define a viewing volume which is 100 units by 1 unit by 1 unit, it will look like a long, thin rod.

Sometimes you need to independently scale each dimension; you want the 100 x 1 x 1 volume to look like a cube rather than a long, thin rod.

ScalePersWindow and *ScaleParaWindow* are like *PersWindow* and *ParaWindow* except that they automatically scale each dimension. The viewing volume will always look like a cube on your screen. Figure 40.23 illustrates the **Scale3** program on your disk; it shows an unscaled and scaled view of a 10 x 10 x 1 volume.

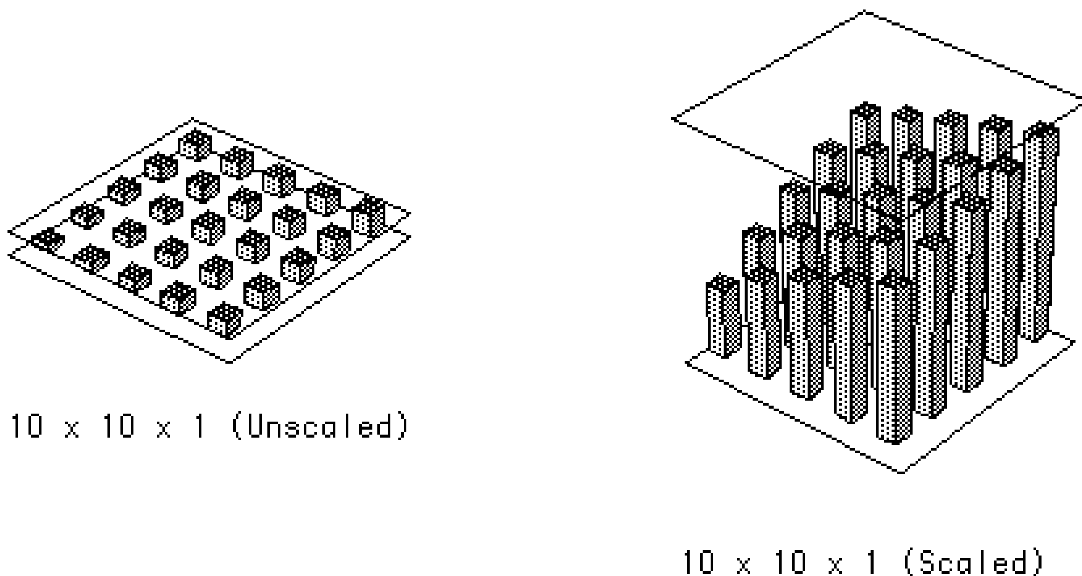

Figure 40.23. Scaled vs. Unscaled Views — The Scale3 *Program.*

ScalePersWindow (x1, x2, y1, y2, z1, z2)

ScalePersWindow works exactly like *PersWindow* except that it creates a scaled view of the viewing volume. The viewing dimensions will be normalized so they all appear to be the same length.

ScalePersWindow is appropriate for perspective views of some volume which has very different measures used along the *x, y,* and *z* axes. See the **Scale3** program for an example.

Exceptions:

33 3-D window minimum = maximum.

ScaleParaWindow (x1, x2, y1, y2, z1, z2)

ScaleParaWindow works exactly like *ParaWindow*, except that it creates a scaled view of the viewing volume. The viewing dimensions will be normalized so they all appear to be the same length.

ScaleParaWindow is appropriate for parallel views of some volume which has very different measures used along the *x, y,* and *z* axes.

Exceptions:

33 3-D window minimum = maximum.

Advanced Viewing Routines

This section describes advanced routines that let you independently manipulate the camera angle, distance, view plane normal, and "up" direction. Since it's tricky to properly coordinate these related parameters, beginners should probably stick to the *SetCamera3* routine which gives an easy interface to them.

However, advanced users may wish to set the parameters separately for various special effects.

SetRefPt3	Set reference point.
SetViewPlane3	Set view plane normal.
SetDistance3	Set viewing distance.
SetScale3	Set scale for flat image.
SetCP3	Set center of perspective.
SetProj3	Set projection plane normal.
SetUp3	Set "up" direction.
AskRefPt3	Ask reference point.
AskViewPlane3	Ask view plane normal.
AskDistance3	Ask viewing distance.
AskScale3	Ask scale for flat image.
AskCP3	Ask center of perspective.
AskProj3	Ask projection plane normal.
AskUp3	Ask "up" direction.

SetRefPt3 (x, y, z)

SetRefPt3 sets the new reference point for viewing. It does not change the view plane normal, viewing distance, or any other related parameter.

If you want to change all these parameters together, try the *SetCamera3* routine. It lets you set the camera position and then picks plausible values for all three related parameters.

Exceptions:

38 View plane is behind center of projection.

SetViewPlane3 (dx, dy, dz)

SetViewPlane3 sets a new view plane normal, that is, changes the direction in which the view plane is tilting.

This routine can give lots of different errors. "View plane normal is zero" means that *dx, dy* and *dz* are all zero — hence no direction has been given. "Can't set view plane along normal" means that you're trying to adjust the view plane so that it intersects the refer-

ence point. "View plane is behind center of projection" probably means that you've got the view plane pointing away from the reference point.

Exceptions:

34 View plane normal is zero.
35 Can't set view plane along normal.
38 View plane is behind center of projection.

SetDistance3 (d)

SetDistance3 sets the viewing distance, i.e., the distance from the reference point to the projection plane. It does not affect the view plane normal, and so keeps the same "camera angle" and simply moves the camera closer to, or farther from, the reference point.

SetDistance3 normalizes the resulting image so it always appears the same size on your screen. (Thus it has no effect for parallel views, and simply changes the degree of perspective for perspective views.) To make the image larger or smaller, use *SetScale3*.

Exceptions:

38 View plane is behind center of projection.
42 Viewing distance is zero.

SetScale3 (h, v)

SetScale3 controls the scale of the flat image displayed on your screen. The two parameters *h* and *v* control horizontal and vertical scaling, respectively.

By default, both horizontal and vertical scales are set to 1. This gives a clear picture where every part of the 3-D image is visible. To magnify the image, pass numbers greater than 1. To shrink the image, pass numbers less than 1. For example:

```
CALL SetScale3(2,1.5)
```

expands the flat image by a factor of 2 horizontally and 1.5 vertically.

SetCP3 (x, y, z)

SetCP3 sets a new center of perspective for perspective views. It does not move the camera at all.

Exceptions:

38 View plane is behind center of projection.
41 Can't do this with a parallel view.

SetProj3 (dx, dy, dz)

SetProj3 changes the direction of the parallel projection lines. (Hence it can only be used with parallel projections.) The direction is specified as an *x-y-z* vector.

By default, projection lines leave the reference point and strike the view plane at a right angle. However, you may change this angle as you wish, so long as the projection lines are not parallel to the view plane.

The view plane itself is not affected by *SetProj3*. Because the projection lines are no longer coming at the same angle, the projected image will appear to shift along the view plane as you change the projection lines' direction. In fact, the image may shift outside of the current viewing window and no longer be visible.

See the 3-D Theory section for a brief description of the effects of changing projection line directions.

Exceptions:

36 Parallel projection direction is zero.
37 Projection is parallel to view plane.
40 Can't do this with perspective view.

SetUp3 (dx, dy, dz)

SetUp3 controls the "camera rotation." In other words, it controls how the projected image is rotated to appear on the computer's screen. By default, the *3-D Graphics Toolkit* arranges the image so that the *x* axis is horizontal and the *z* axis is vertical, but this routine lets you change that.

The three arguments *dx, dy,* and *dz* are taken as a vector which describes the camera's new orientation.

In the simplest cases,

```
CALL SetUp3(1,0,0)
```

makes the *x* axis vertical,

```
CALL SetUp3(0,1,0)
```

makes the *y* axis vertical, and

```
CALL SetUp3(0,0,1)
```

makes the *z* axis vertical. If you use -1 instead of 1, the result will be upside down.

If you try to arrange the camera so the view plane intersects the reference point, you get the "Can't set view plane along normal" error.

Exceptions:

35 Can't set view plane along normal.
39 View up direction is zero.

AskRefPt3 (x, y, z)

AskRefPt3 returns the current coordinates (*x, y, z*) of the reference point.

AskViewPlane3 (dx, dy, dz)

AskViewPlane3 returns the current "view plane normal" in *dx, dy,* and *dz*. These numbers will have been normalized to make a unit vector; however, the direction of the vector is unchanged from that which you set by *SetViewPlane3*.

AskDistance3 (d)

AskDistance3 returns the current viewing distance in *d*.

AskScale3 (h, v)

AskScale3 returns the current horizontal and vertical scaling factors. If you haven't called *SetScale3*, both numbers will be 1.

AskCP3 (x, y, z)

AskCP3 returns the current center of perspective in *x, y,* and *z*. It gives an error if there is no center of perspective — i.e., you are using a parallel projection.

Exceptions:

41 Can't do this with a parallel view.

AskProj3 (dx, dy, dz)

AskProj3 returns the current parallel projection line direction in *dx, dy,* and *dz*. These numbers will have been normalized to make a unit vector but the direction of the vector is unchanged from that which you set by *SetProj3*.

Exceptions:

40 Can't do this with perspective view.

AskUp3 (dx, dy, dz)

AskUp3 returns the current "up" direction for the camera in *dx, dy,* and *dz*. These numbers will have been normalized to make a unit vector but the direction of the vector is unchanged from that which you set by *SetUp3*.

Sample Programs

Your disk includes a selection of sample programs to help you understand how to use the *3-D Graphics Toolkit*. They're ready to go; just call them up and run them.

Cube3

Cube3 displays the unit cube in a perspective view. It runs in an infinite loop asking you to supply a new "camera" location and displaying the view from that location.

Camera positions like (3, 4, 5) give a fairly bland view of the cube. Try moving the camera right to the edge of the cube to see how distorted the cube can look if the camera's too close to the object. Move far away and watch how the perspective effect begins to grow less.

Move the camera inside the unit cube — say, to (.7, .7, .7) — and see how things look when you draw lines behind the camera. Since the *3-D Graphics Toolkit* does not clip lines behind the camera, they appear as seemingly random lines drawn across the viewing surface.

Finally, change *PersWindow* to *ParaWindow* and experiment with a parallel projection. Notice that moving closer to the object, or farther away, does not affect parallel views.

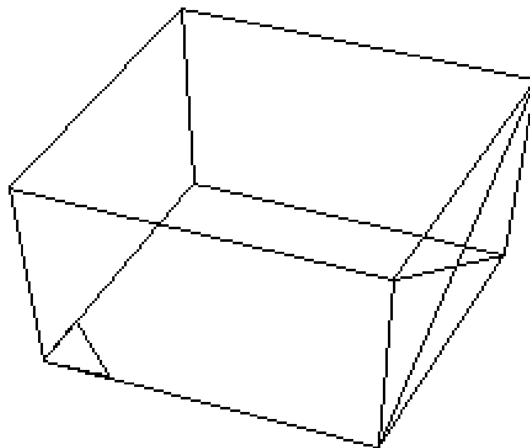

Figure 40.24: Cube3 *Program*

House3

The **House3** program draws a perspective view of a “house.” It uses various graphics primitives such as line-drawing, rectangles, and circles.

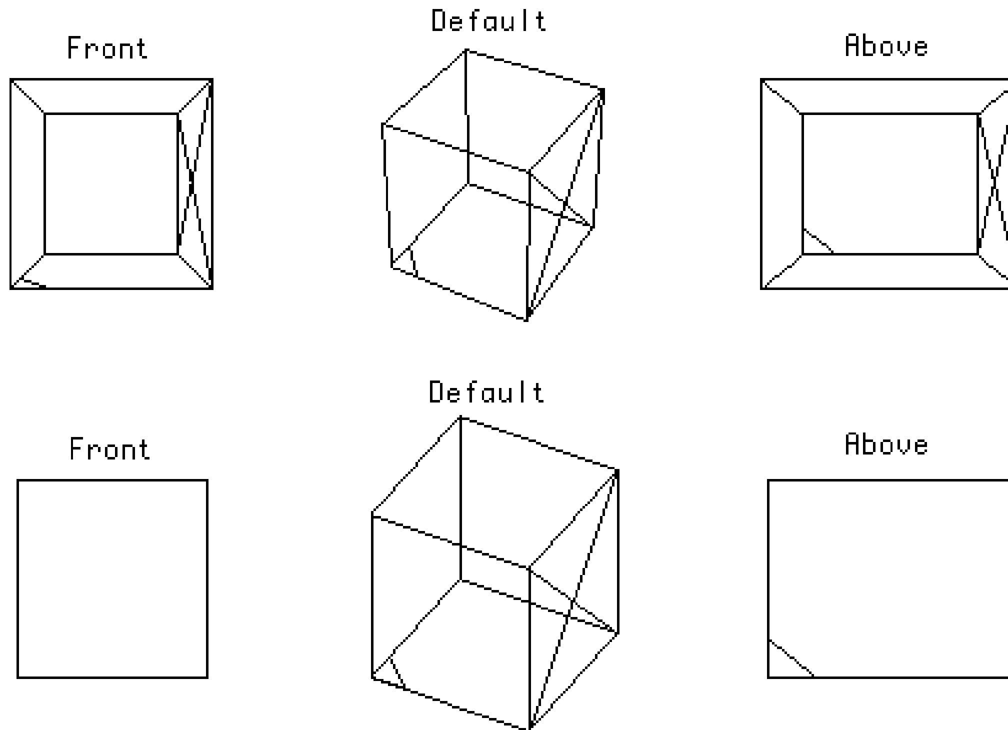

Figure 40.25: Project3 *Program*

Project3

The **Project3** program opens several True BASIC windows and displays various perspective and parallel views of a house within the windows. It shows how to coordinate 3-D graphics and True BASIC windows.

Unlike the built-in True BASIC graphics, each window does not have its own, independent 3-D window coordinates. Rather, they all share the same 3-D view. You must switch 3-D windows explicitly every time you switch to another True BASIC window.

Oblique3

The **Oblique3** program opens four True BASIC windows and displays a different parallel projection in each window. The top two views are both axonometric projections; the left is “head on” and the right is from above and to one side.

The bottom two views are both oblique projections, taken from the same camera position as the top left view. The bottom left is a cavalier projection; the bottom right is a cabinet projection.

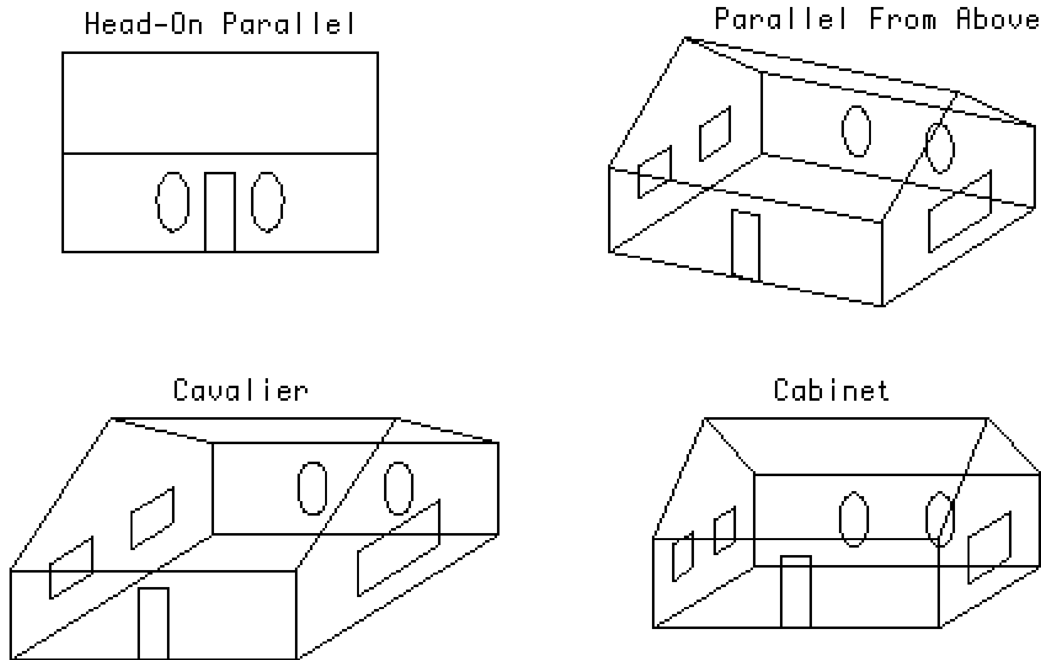

Figure 40.26: Oblique3 *Program*

Bars3

The **Bars3** program displays a three-dimensional bar chart in a perspective view. It uses the *GridZ3* subroutine to get an underlying grid and the *Block3* routine to draw each bar.

Notice how **Bars3** alters the viewing parameters to get a more dramatic display. It places the camera rather low and then changes the reference point so it's no longer in the center of the row of bars. Finally, it resets the 2-D window coordinates to enlarge the resulting view. (As a result, the image spills off the edges of the screen.)

Blocks3

The **Blocks3** program is somewhat like **Bars3** except that it displays a cluster of blocks laid out over a grid. The image resembles a city block seen from the air.

Blocks3 uses a "painter's algorithm" to give a realistic view of the blocks. In less fancy words, it draws the rearmost blocks first, then nearer blocks. Thus the blocks in front hide the blocks behind.

Zmesh3

The **Zmesh3** program plots a perspective image of a rather peculiar F(*x,y*). It uses a very fine mesh setting so be patient!

Zplot3

The **Zplot3** program shows off the Zplot subroutine. It plots the image of a photogenic function as seen in perspective from a certain camera position. Then it lets you set a new camera position, and displays the function from that angle.

Notice how the *Zplot* subroutine will automatically switch between x and y contour lines, depending upon the camera location.

Zsplot3

The **Zsplot3** program shows off the *Splot* subroutine. It also shows how various "tricky" functions — which aren't defined over an entire rectangle — can be handled with the *Zplot* subroutine. See the Contour Plots section for more information.

Topo3

The **Topo3** program shows the topographic map for a complicated function with two bumps. It also shows a *Zplot* of the function to make the topographic map clearer.

In addition, **Topo3** uses *SetColor3* to draw different heights in different colors. This accentuates the features of the function's surface.

Zdata3

The **Zdata3** program shows a *ZplotData* image of a large number of data points. If you have enough patience, you might change the program to use *ZmeshData* or *ZbarData* instead. (Both are slower than *ZplotData*.)

Zbar3

The **Zbar3** program shows a *ZbarData* plot of a 17 x 17 data set, superimposed atop a topographic plot of the same data.

The two layers have been accentuated by using *Block3* to add solid blocks below the layers.

Scale3

The **Scale3** program shows the difference between a scaled and unscaled view of the same volume. Both views show a volume that ranges from 0 to 10 in both the x and y dimensions and from 0 to 1 in the z dimension.

Notice that the unscaled view preserves relative sizes, making the result look very shallow. The scaled view exagerates the z dimension to bring it into balance with the x and y dimensions.

See the Scaled Views section for more information on getting scaled viewing volumes.

Making 3-D "Movies"

The **Record** and **PlayBack** sample programs let you create and show 3-D "movies." These movies are created by animation: **Record** draws a movie one frame at a time and stores the frames in a file on disk. **PlayBack** reads this file and shows the frames quickly to give the impression of action.

Call up the PLAYBACK program and run it. When it asks for a file name, reply:

```
imagdata.
```

It will show a pre-recorded movie of the "house" picture. (If your computer cannot handle the movie's recording mode, **PlayBack** will give an error. If this happens, just call up **Record** and run it instead. This re-records the movie in your current mode and then plays that.)

Movies require lots of memory — at least 100 to 200k on most computers — so you may not be able to run a movie if you do not have large amounts of memory available.

Record

Record draws a movie frame by frame. To make a new movie, you must rewrite parts of the **Record** sample program. It consists of one main loop which draws new frames from various camera viewpoints. For each frame, it picks a new camera location and draws the "house" picture from that camera angle. Then it uses BOX KEEP to grab the frame. Finally it stores the frames in a movie file, **ImagData**, and chains to **PlayBack** passing the file name. Then **PlayBack** shows the movie.

PlayBack

PlayBack shows a movie. It opens a movie image file, reads all the frames into memory, and then displays the frames one after another. It shows the film as a "loop" so that it goes back to the first frame after showing the last one.

Note how **PlayBack** includes a PROGRAM statement. This way, it can receive the movie file name if some other program (such as **Record**) chains to it. Otherwise it simply asks the user for a movie file name.

PlayBack delays about .05 seconds after showing each frame. Otherwise the movie goes too fast.

The Movie File Format

Movie files, such as **ImagData**, are stored as True BASIC record files. They contain more than movie frames. The first record contains the following items in a packed format:

- Number of frames.
- Frame's screen *xmin, xmax, ymin, ymax.*
- Recording mode

The remaining records are all movie frames. This format is a little more general than is needed for these simple 3-D movies. But it allows later expansion to more advanced movie systems.

This format is different from Version 1.0 movie files. We've had to add the recording mode since some computers cannot play back movies recorded in different modes. For instance, IBM PCs with Hercules cards can't play back the standard IBM PC graphics mode.

If You Want to Experiment More...

Movies require a great deal from your computer. A simple movie, like the one on your disk, needs a lot of memory and processing speed to run. The size of the frames, and the number of frames, have been carefully chosen to fit in a reasonable amount of memory and run without too much flicker. On computers with "modes," we've chosen the graphics mode that requires the least amount of memory for each frame.

Nevertheless, some computers can handle bigger movies and show them faster. You may want to experiment with the frame sizes and number of frames to see how things work on your computer. Just adjust the window size and number of frames in **Record** and then run it.

Remember that **PlayBack** pauses for .05 seconds between each frame. Instead of pausing, you can use this time to display a frame from another movie on another part of the screen. Experiments have shown that the Macintosh Plus, for instance, can play three movies simultaneously without annoying flicker. (Each movie frame, though, is a bit smaller than the one used by **Record**.) You may wish to experiment with running several 3-D views simultaneously.

Finally, it's not absolutely essential to keep the entire movie in memory. If you have a computer with a small memory or want to show very long movies, you can modify **PlayBack** to read frames one by one from disk (rather than read them all into memory). This makes movies considerably slower but allows much longer and more complicated movies.

3-D Transformations

This section describes, very broadly, how the *3-D Graphics Toolkit* creates projections. This subject is complicated and you should read any of the books listed in the 3-D Theory section if you want to learn more about 3-D projections.

There are two major parts to a 3-D graphics package. One part sets up a view and readies all the information for making projections. The other part actually draws images. It uses the information created by the viewing set-up routines.

The set-up routines take all the information about a viewing position — reference point, camera position, view plane normal, etc. — and encode it into a “transformation matrix.” This is explained at length below. Whenever the viewing parameters are changed, the transformation matrix must be recomputed.

The drawing routines use this transformation matrix to project 3-D images onto the view plane.

Absolute and Camera Coordinate Systems

Two different coordinate systems are used in a projection. The first is the “absolute” coordinate system with its origin at (0,0,0). The second is the “camera” coordinate system with its origin at the camera position.

In the camera coordinate system, the x-y plane acts as the view plane. The z coordinate is simply the distance from the view plane. In this coordinate system, the center of the view plane — the camera position — is at the origin.

The origins of these two systems (reference point and camera position) will never coincide. And in general the axes for these two coordinate systems will not coincide. Usually they will not even be parallel. Instead, the camera view plane will be at some 3-dimensional angle to the absolute coordinate system.

The projection package must first, therefore, transform the absolute coordinates into camera coordinates. This requires translating the absolute origin to the camera coordinate origin and then rotating each axis to bring it into alignment.

Once the *3-D Graphics Toolkit* has transformed points from the absolute system to the camera system, it can draw images on the view plane. For most parallel projections, it can draw the image’s line by simply ignoring the z dimension (distance from the camera). For perspective projections, the package must scale the x-y size by the z coordinate so farther objects appear smaller.

Transformation Matrices

Internally, most graphics packages work with transformation matrices. These contain, in a concise form, all the calculations which must be applied to a point to give its projected image. Compare these matrices with their 2-D equivalents shown in Section __).

Translation Matrix

The translation matrix shifts the absolute reference point to the camera position. If *dx, dy,* and *dz* are the *x, y,* and *z* distances between the reference point and the camera location, then the translation matrix is:

Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

Rotation Matrices

Once the origin has been moved to the camera location, the package must re-orient all three axes. In general this requires 3 “rotations”: each of the *x, y,* and *z* axes must be brought into line.

Assuming that you need to rotate the axes by angles *ax, ay,* and *az* respectively, the three rotation matrices are:

X-axis Rotation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(ax) & \sin(ax) & 0 \\ 0 & -\sin(ax) & \cos(ax) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-axis Rotation Matrix

$$\begin{bmatrix} \cos(ay) & 0 & -\sin(ay) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(ay) & 0 & \cos(ay) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-axis Rotation Matrix

$$\begin{bmatrix} \cos(az) & \sin(az) & 0 & 0 \\ -\sin(az) & \cos(az) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composition of Transformations

These transformations could be applied one after another: first the translation, then three separate rotations. However, the transformations can also be "composed" into a single operation — which is much more efficient than performing four operations.

The transformations are composed by multiplying the transformation matrices in order. Thus the final transformation matrix is computed by:

```
transform = Translate(dx,dy,dz) * RotX(ax) * RotY(ay) * RotZ(az)
```

This transformation matrix can be computed once the camera position and reference point are known. Hence the *3-D Graphics Toolkit* computes the matrix once whenever you fix the viewing parameters. Then it uses the matrix whenever you display an image.

This toolkit's *MakeViewPlaneTransformation* subroutine creates and composes the transformation matrices.

(For efficiency, the transformation matrix is not kept as an actual True BASIC 4x4 matrix but rather as a related set of variables. This removes the small time necessary for True BASIC to calculate and check subscripts. And since the last columns of all matrices are invariably [0, 0, 0, 1], this column is ignored.

It has no effect on the calculations, so why waste time doing useless multiplications and additions with its elements?)

The Final Touches

After translating a point from absolute coordinates to camera coordinates, we must finally project it onto the view plane. This is done in different ways for perspective and parallel projections.

Perspective projections must scale the resulting image by the distance from the view plane. This makes farther objects look smaller. In the camera coordinate system, the z coordinate is simply the distance from the view plane, so the final perspective transformation is just a division by the z coordinate.

Parallel projections do not scale the image based on distance. In fact, the z coordinate is usually ignored. However, the z coordinate is used for oblique projections, and so the final transformation uses the view plane's "tilt" and the z coordinates to get the projected image.

Both of these final transformations can be encoded within the transformation matrix. The *3-D Graphics Toolkit*, however, keeps them as separate operations. This toolkit's *Transform* subroutine transforms a 3-D point to its 2-D image. It uses the transformation matrix to convert from absolute to camera coordinates and then applies the final touches as described in this section.