PS 390 DOCUMENT SET

GRAPHICS TUTORIALS 8–16

Copyright © 1987
EVANS & SUTHERLAND COMPUTER CORPORATION
P.O. Box 8700, 580 Arapeen Drive
Salt Lake City, Utah 84108

GRAPHICS TUTORIALS

The tutorial sections in *GT1–7* and *GT8–16* contain an in-depth discussion of PS 390 programming. After reading these sections initially, you may want to use these volumes in conjunction with *RM1-16* as a reference source to program your own applications.

GT8-16 consists of sections which describe more advanced concepts of PS 390 graphics programming. Because it builds on fundamental information detailed in *GT1-7*, you should read those sections first. The following provides a capsule description of each section:

GT8 VIEWING OPERATIONS

This section describes how to look at a model from different viewpoints. This includes moving your viewpoint to another location in the coordinate system, choosing a perspective view, and specifying a viewing area.

GT9 CONDITIONAL REFERENCING

Conditional Referencing describes how detail can be added to or deleted from a view on the screen.

GT10 TEXT MODELING

Text Modeling details how to create character strings, how to use commands and functions to manipulate character strings, and how to create and use different character fonts.

GT11 PICKING

Picking describes how to use the data tablet to activate a given action by picking an object being displayed.

GT12 VIDEO OUTPUT CONTROL

This section describes how to control the video output of the PS 390 graphics system, including how to select a background color, the configuration and color of the screen cursor, a video timing format, and how to select filters to implement antialiasing.

GT13 POLYGONAL RENDERING

This section describes how to define polygonal objects and how to perform rendering operations, including cross sectioning, hidden-line removal, and static shaded image rendering.

GT14 RASTER PROGRAMMING

Raster Programming describes the use of the PS 390 as a frame buffer for displaying host-generated images.

GT15 SAMPLE PROGRAMS

This section contains sample programs illustrating various PS 390 programming techniques.

GT16 GLOSSARY

This is a glossary of terminology specific to the PS 390.

GT8. VIEWING OPERATIONS

GT8. VIEWING OPERATIONS

LOOKING AT THE MODEL

CONTENTS

ILLUSTRATIONS

Section GT8
Viewing Operations
Looking At The Model

Introduction

Once you have created a model and displayed it on the screen, you may want to look at it from different viewpoints. One way to do this is to manipulate the model into different positions. You have already learned how to do this using modeling transformations—rotations and translations. Another way to change your view is to keep the model in place and essentially move yourself as "viewer" about the model. This is done on the PS 390 using viewing transformations.

There are two basic types of viewing transformations. The first type establishes the viewer's position in the world coordinate system and the direction in which he is looking. This is known as specifying a line of sight. The second type of viewing transformation lets you specify how much of the world coordinate system will appear in your view. This is done by defining the boundaries of a viewing area or window. Objects within a window may appear in either parallel projection (an orthographic view) or in perspective projection.

Parallel projection creates a view in which the relative size of an object, or parts of an object, is maintained as specified in the original object definition, no matter where the object is located in Z. Perspective projection causes a distant object or parts of an object to diminish in size as they recede into the distance toward positive Z.

In both parallel and perspective views, clipping is used to eliminate objects or parts of objects that lie outside the boundaries of the window. In both, the illusion of depth can be enhanced using depth cueing. Depth cueing makes objects or parts of objects dimmer as they recede into the distance.

In addition to the two types of viewing transformations, you can specify a viewport. A viewport is a portion of the PS 390 display in which the window is displayed. Because the PS 390 allows the display of both antialiased wireframe models and shaded images on the same raster screen, it is necessary to distinguish between how and where each type of model is displayed.

There are two types of viewports: the dynamic viewport and the static viewport. Wireframe models are displayed and manipulated in the dynamic viewport, and hidden-line images or shaded renderings are displayed in the static viewport. An unlimited number and combination of dynamic and static viewports are available within the usable screen space, letting you display different views of the same model or view different models simultaneously. Viewports can be either full-screen or smaller portions of the screen.

The last set of viewing operations you can specify is called viewing attributes. These allow you to set an intensity range for displayed data in the dynamic viewport, and set color for displayed objects in the dynamic viewport.

When you turn on the PS 390, you are automatically provided with a default line of sight (down the positive Z axis from the origin), a window (orthographic, with dimensions from -1 to 1 in X and Y; from 10^{-15} to 10^{+15} in Z), and a dynamic viewport (which is full screen).

Most of the PS 390 viewing operations—viewing transformations, dynamic viewports, and viewing attributes—are represented in a model's display structure by operation nodes. Specifying a static viewport, however, is done through the use of the initial function instance SHADINGENVIRONMENT, and does not create a node in the display structure.

Objectives

In this section you will learn how to create various views of the world coordinate system. To do this, you should know how to:

- Define a line of sight.
- Define orthographic windows.
- Define perspective windows.
- Specify a dynamic or static viewport.
- Set an intensity range for the dynamic viewport.
- Set color in the dynamic viewport.

Prerequisites

Before reading this section, you need to know basic graphics concepts, how data structuring is done in the PS 390, and how modeling transformations work on data.

This section makes use of tutorial demonstrations. (Refer to *GT3 PS 390 Tutorial Demonstrations.*)

To do the exercises in this section, put the PS 390 in command mode.

CTRL/LINE_LOCAL (PS 300-Style Keyboard)
CTRL/CMND or ALT/CMND (PS 390-Style Keyboard)

1. Defining a Line of Sight

There are two types of viewing transformations that alter the way in which a model is viewed. The first kind of transformation defines a line of sight.

In the real world, you establish a line of sight by placing yourself in a particular position relative to the object you are viewing. The line of sight is the invisible straight line between the point you are looking from and the point you are looking at. Changing either one of these points gives you a different line of sight.

The PS 390 simulates this relative positioning with the LOOK command. The LOOK command lets you see your model from any point in the world coordinate system.

The LOOK command creates a 4x3 matrix operation node in the model's display structure. For a LOOK transformation to work correctly, it should be placed above all modeling transformations (ROTATE, TRANSLATE, SCALE) in the structure (Figure 8-1).

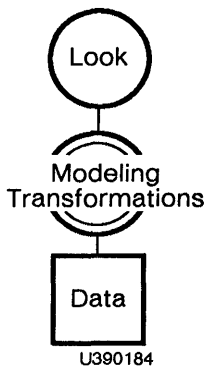


Figure 8-1. LOOK Node

Note that the operation node created by LOOK can be an interactive node, with values for the AT and FROM points being changed via a function network (F:LOOKAT and F:LOOKFROM).

The default line of sight starts at the origin and points along the positive Z axis. The viewer looks FROM 0,0,0, AT 0,0,1 (Figure 8-2).

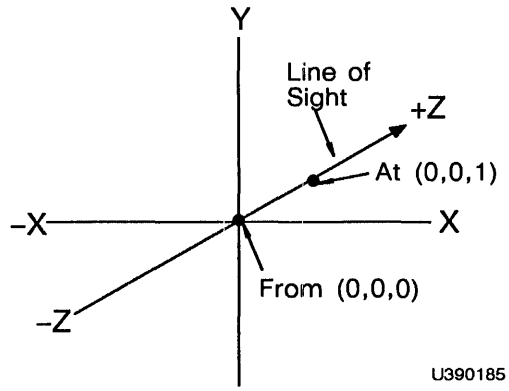


Figure 8-2. Default LOOK

Display the Car. Notice that the orientation of the car (default line of sight) is as shown in Figure 8-3.

Enter:

```
DISPLAY Car;
```

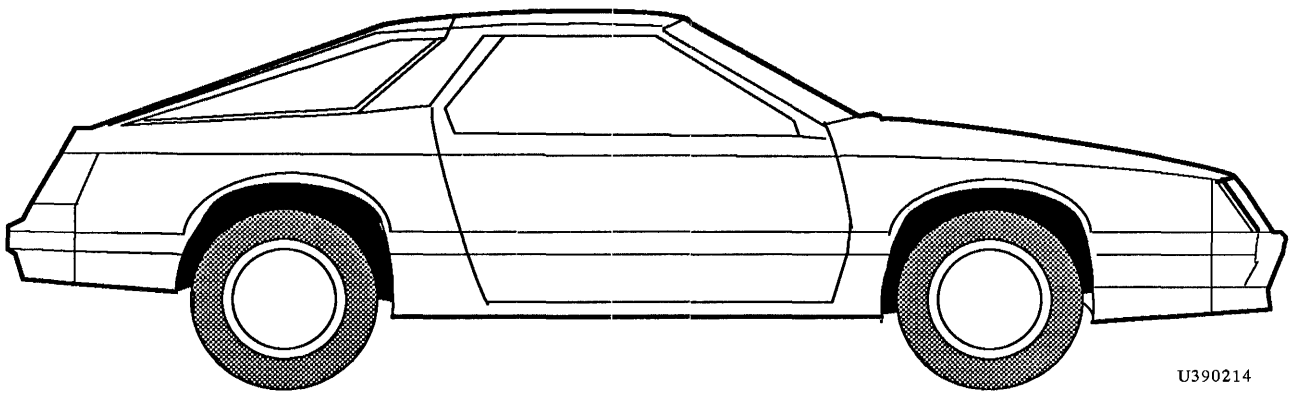


Figure 8-3. Car

To see the other side of the car, specify a LOOK (Left_View) with the FROM point on the positive Z axis (0,0,.1) looking AT the origin (0,0,0). Apply that line of sight to Car. Then DISPLAY Left_View.

Enter:

```
Left_View := LOOK FROM 0,0,.1 AT 0,0,0  APPLIED TO Car;
REMOVE Car;
DISPLAY Left_View;
```

You should now see the car from the left side as shown in Figure 8-4.

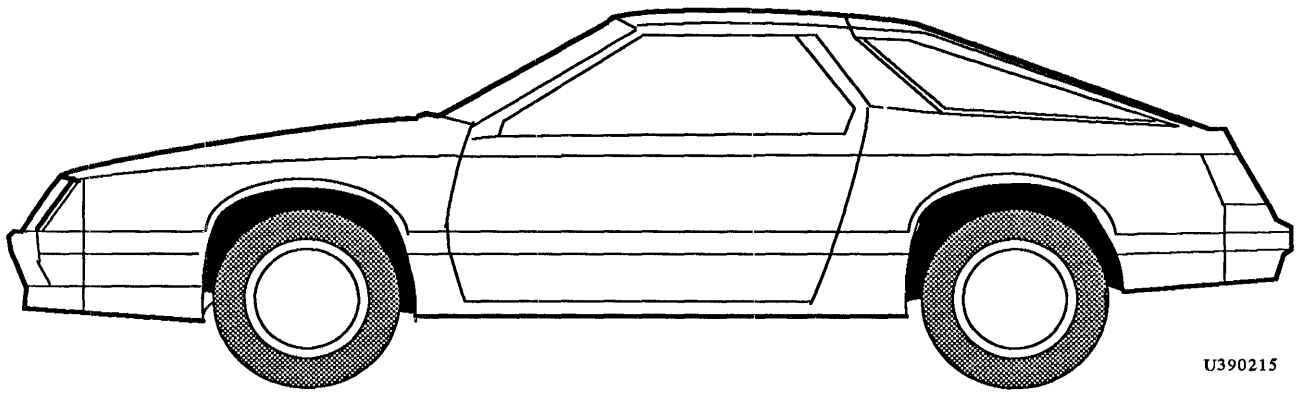


Figure 8-4. Car From Left Side

To create Left_View, the PS 390 first translates all points in the world coordinate system to put the FROM point (0,0,.1) at the origin. Then all points in the world coordinate system are rotated around the FROM point (the origin) until the AT point is on the positive Z axis. This orients the car correctly for the LOOK specified in Left_View, as shown in Figure 8-5. (Note that the translation shown in Figure 8-5 is exaggerated for clarity.)

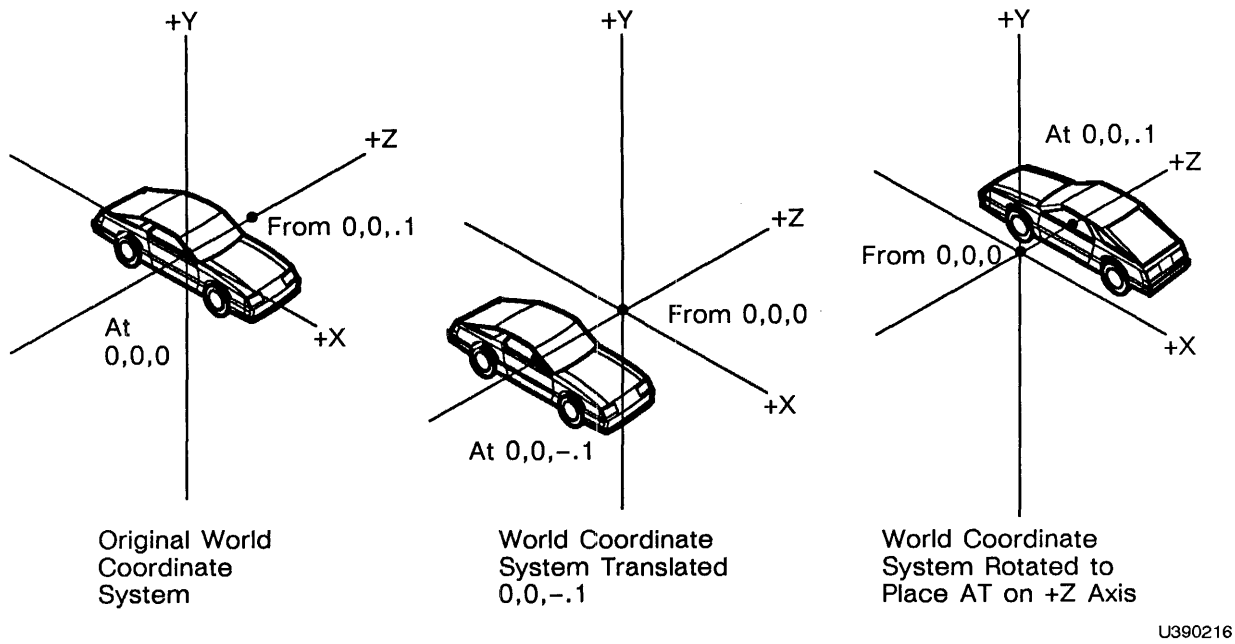


Figure 8-5. LOOK Transformation Sequence

1.1 Looking Straight Up or Straight Down

For any LOOK, an UP direction is specified by the system if you do not specify one yourself. The default UP direction is derived by taking the vector that defines the AT point (X,Y,Z) and adding 1 to the Y component. The resulting vector is placed in the positive half of the Y/Z plane, thereby defining UP. The rotation for UP occurs after the translation that puts the FROM point on the origin (0,0,0) and the rotations that put the AT point on the positive Z axis.

For example, if the FROM point in a LOOK is 0,1,0 and the AT point is 1,1,1, the default UP point defining the Y/Z plane would be 1,2,1.

If the FROM point of a LOOK is directly above or below the AT point, the system has to define an alternate UP direction. What would normally be the UP direction is now collinear with the line of sight (Figure 8-6).

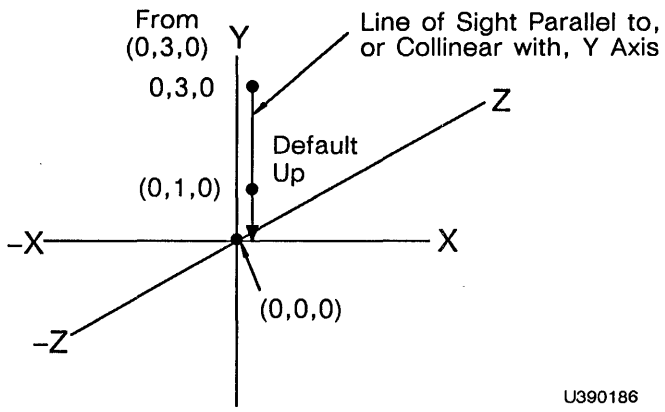


Figure 8-6. Line of Sight Collinear with UP Direction

In such cases the system takes the vector that is the AT point, adds one to its Z component, and rotates the world to place that point in the positive half of the Y/Z plane. To demonstrate this, enter:

```
REMOVE Left_View;

Top_View := LOOK FROM 0,.1,0 AT 0,0,0  APPLIED TO Car;

DISPLAY Top_View;
```

The direction that is positive Z in the original model of Car is now up in Top_View (Figure 8-7). That direction was derived by adding 1 to the Z component of the AT vector in Top_View, and using that point (0,0,1) to define UP as shown in Figure 8-7. (Note that in Figure 8-7 the distance from the FROM point to the AT point is exaggerated for clarity.)

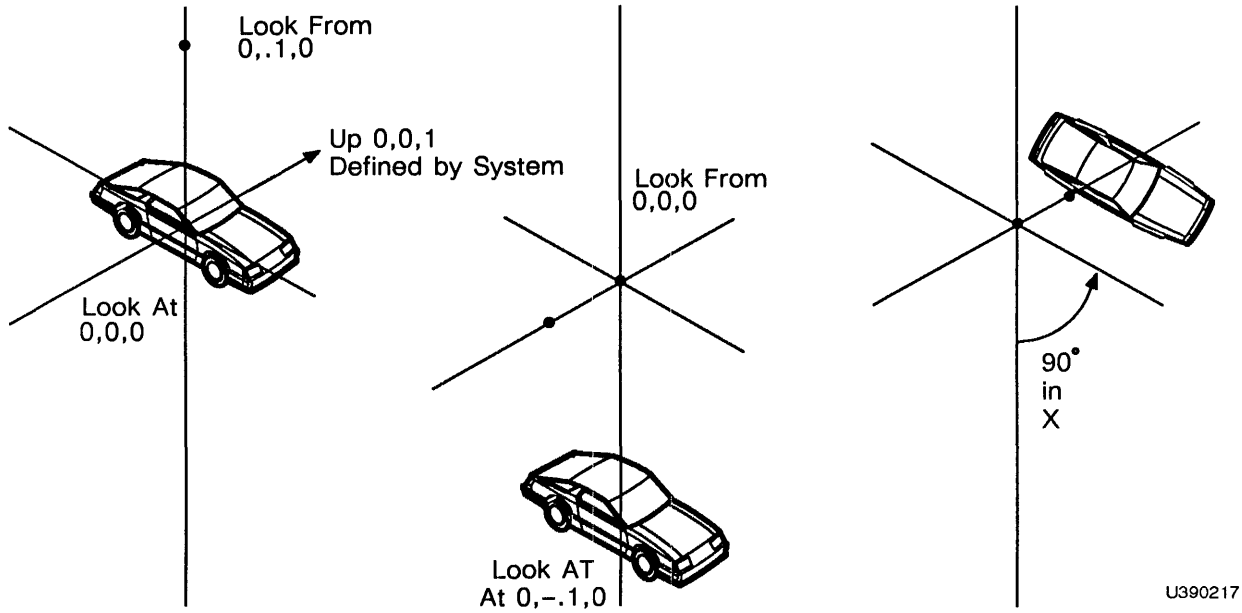


Figure 8-7. LOOKing Down

UP can be specified in a LOOK command even if the line of sight does not define a straight-up or straight-down view. Redefine Top_View to change the UP direction to what is positive X in the original model of Car by entering:

```
Top_View := LOOK FROM 0,.1,0 AT 0,0,0 UP 1,0,0
            APPLIED TO Car;
```

The view is reoriented to place the up point (1,0,0) in the positive half of the Y/Z plane (up) in Top_View.

```
REMOVE Top_View;
```

1.1.1 Exercise

Refer to Section *GT3 PS 390 Tutorial Demonstrations* and run the LOOK demonstration program.

1.2 Using a 4x3 Matrix to Specify a Line of Sight

You can build your own 4x3 matrix in lieu of the one created by the LOOK command by using the MATRIX_4x3 command:

```
Name := MATRIX_4X3
            m11,m12,m13
            m21,m22,m23
            m31,m32,m33
            m41,m42,m43 APPLIED TO Another_Name;
```

(For more details, refer to section *RM1 Command Summary.*)

2. Defining An Orthographic Window

The second type of viewing transformation defines a viewing area—a portion of the world coordinate system that is displayed on the screen. This section introduces the first of three possible ways to define a viewing area, using the WINDOW command.

The WINDOW command allows you to specify a three dimensional viewing area (right rectangular prism) in which objects may be viewed. Once a window transformation is applied, all points in the world coordinate system is translated so that the central axis of the window coincides with the positive Z axis (the line of sight).

Objects inside a window appear in orthographic or parallel projection. That is, far objects (relative to the front window plane) do not appear to be smaller than near objects, so the location of an object in Z has no effect on its size on the screen. Perspective does not exist. Farther away parts of objects will appear to be dimmer in the default view. This is called depth cueing.

The WINDOW transformation is a 4x4 matrix operation represented by an operation node in the model's display structure. In the PS 390, a 4x4 matrix overrides all transformations in effect when the matrix is encountered. A 4x4 matrix must be the topmost matrix operation node along any branch in a display structure. If it is not, any operations above it will have no effect. Figure 8-8 illustrates this rule.

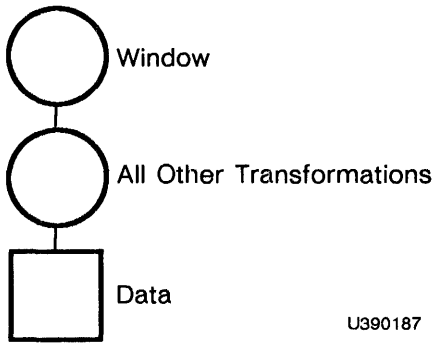


Figure 8-8. WINDOW Node

Just as there is a default LOOK imposed by the PS 390, there is also a default window. The default window is an orthographic window that extends from –1 to 1 in the X and Y dimensions, and from 10^{-15} to 10^{+15} in Z. Any object that lies within this viewing area (Figure 8-9) will appear on the screen when displayed. Objects outside the window in Z will be displayed unless depth clipping is enabled. Refer to section 2.3.

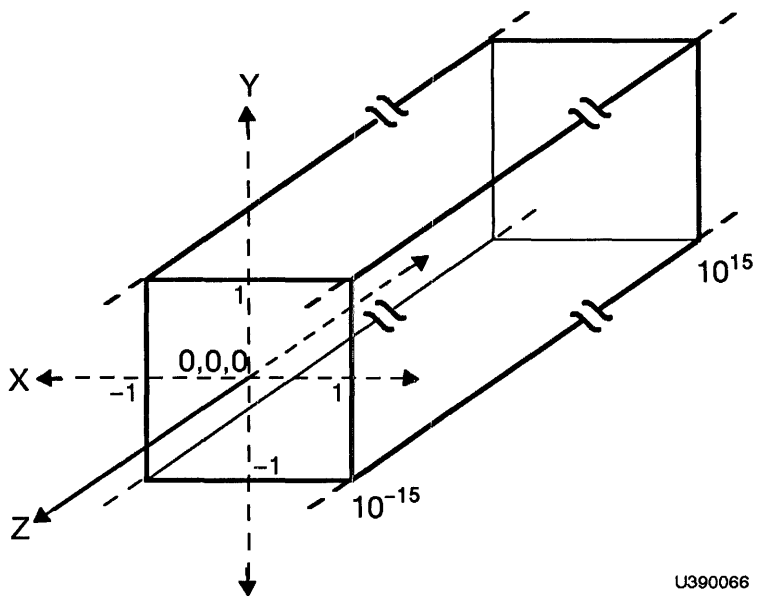


Figure 8-9. Default WINDOW

To see an object, it must be located within the X and Y boundaries of the viewing window. Any object outside these boundaries is removed from view via clipping.

If a part of a model is not entirely within the X and Y boundaries of a window, only a portion of the model appears. For example, the following line of sight effectively moves the object so that part of the Car falls outside the viewing area:

```
Another_View := LOOK AT 1,0,0 FROM 1,0,-.1
                APPLIED TO Car;
```

The part of the Car that appears on the screen is inside the boundaries of the default window. The part of the Car that is clipped falls outside the default window boundaries in X (Figure 8-10).

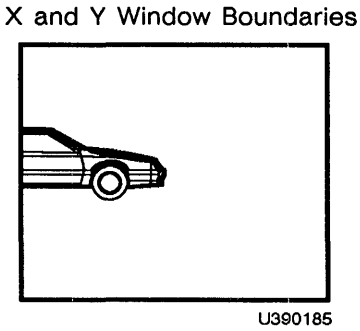


Figure 8-10. Clipped View of Car

2.1 Altering the Size of a Window

The X, Y, and Z boundaries of the default window may be changed to affect window size. Boundaries may be changed using the WINDOW command.

The size of the window influences the apparent size of objects being viewed. If the window is enlarged, objects will appear smaller; if the window size is reduced, objects will appear larger. Altering window size may cause an object to appear so large that it is completely or partially clipped from view.

For example, the default window for Another_View clips off part of Car. You can redefine a window for Another_View that does not clip any part of the car.

2.1.1 Exercise

Define Another_View of Car as shown in the previous example and display Another_View to see the effect (Figure 8-10). Now enlarge the window and apply the new window specification to the LOOK called Another_View.

Enter:

```
Large_window := WINDOW   X=-2:2   Y=-2:2
                APPLIED TO Another_View;
DISPLAY Large_Window;
REMOVE Another_View;
```

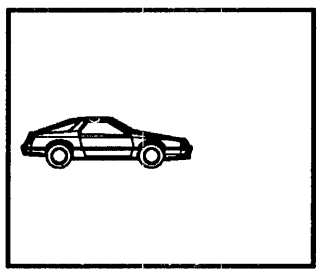


Figure 8-11. Car in Large Window

All of the car appears in Large_Window (Figure 8-11). The car appears smaller than it did in Another_View because Large_Window encompasses more area than the default window used in Another_View.

```
REMOVE Large_Window;
```

The display structure created by the above sequence of commands is shown in Figure 8-12.

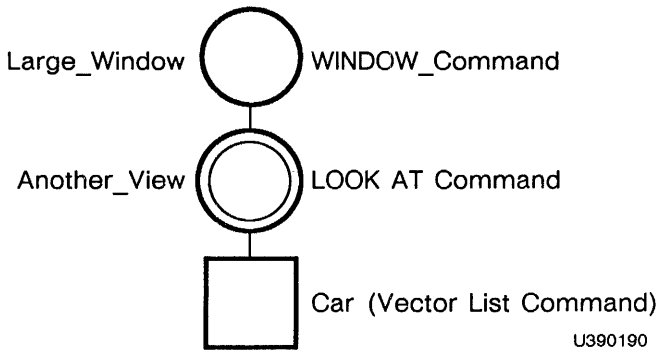


Figure 8-12. Display Structure for Large Window

2.2 Moving the Window

Another way to define a window for Another_View that does not clip any part of the car is to move the window to encompass Car. Moving a window causes the line of sight to be shifted to a new, parallel line of sight.

If an orthographic window is defined as shown in Figure 8-13 so that its center is not coincident with the Z axis, the PS 390 translates everything in the world coordinate system to center the window about the Z axis. You do not need to use a LOOK to move the line of sight to the Z axis.

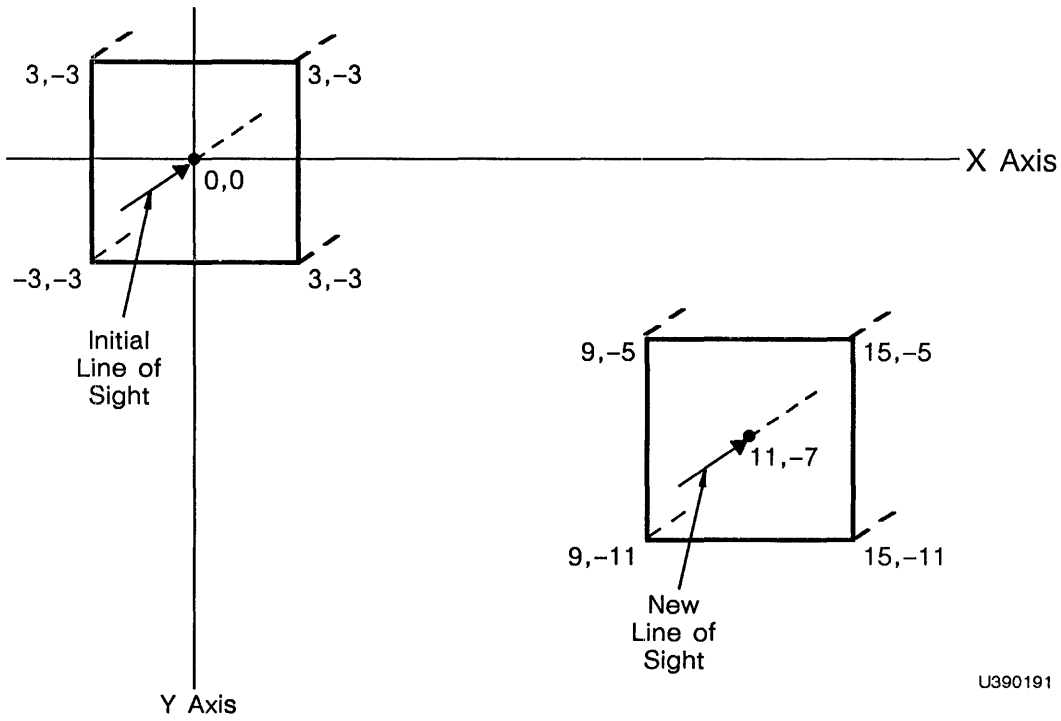


Figure 8-13. Relocated Window

2.2.1 Exercise

Define a "moved" window the same size as the default window (2 units in x by 2 units in y), but place it so that the car in Another_View will be in it:

```
DISPLAY Another_View;
Move_Window :=  WINDOW   X=-2:0  Y=-1:1
                APPLIED TO Another_View;
DISPLAY Move_Window;
REMOVE Another_View;
```

Move_Window clips no part of the car.

```
REMOVE Move_Window;
```

Figure 8-14 shows the sequence of transformations that makes Move_Window encompass the car.

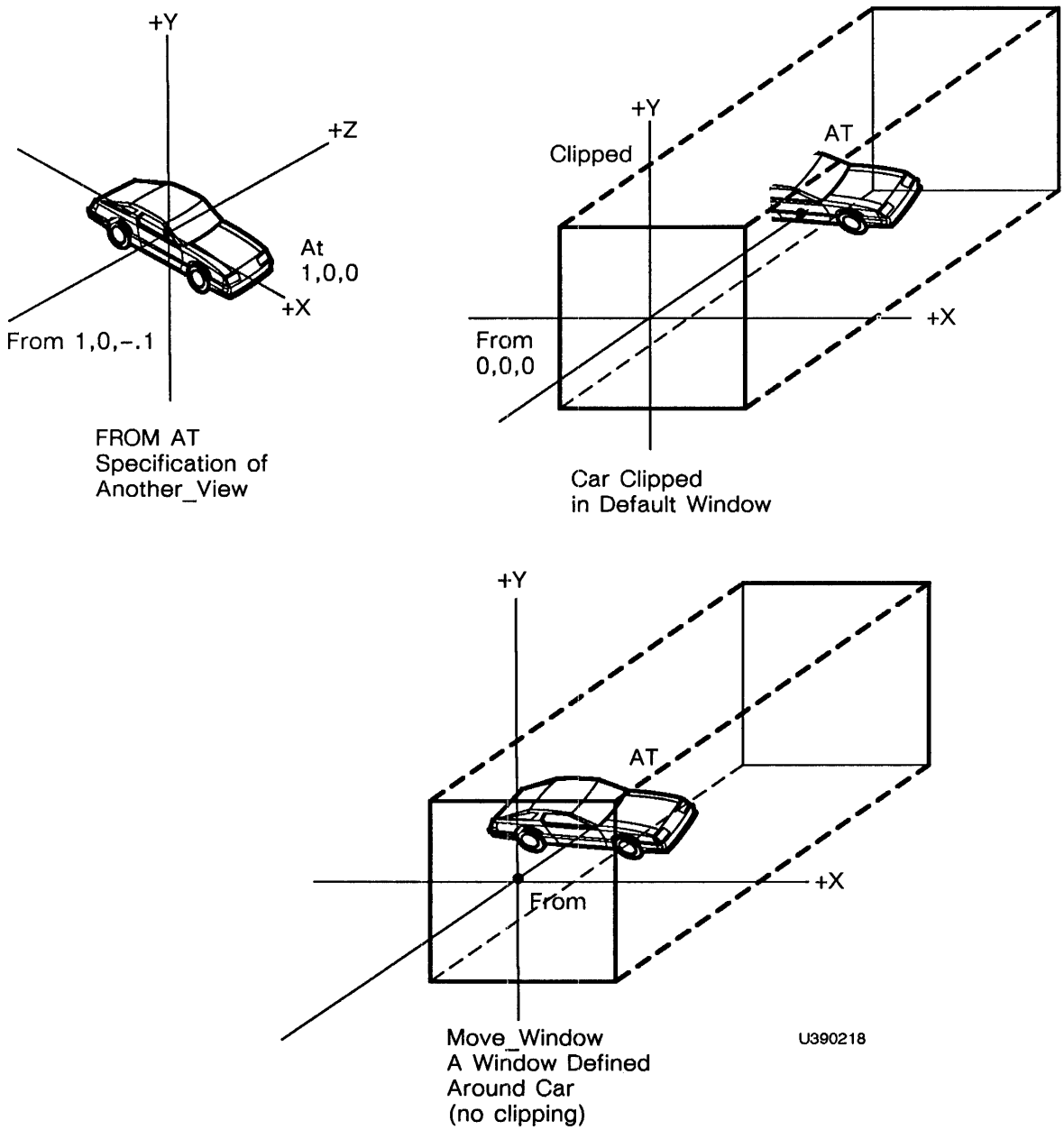


Figure 8-14. Interrelation of LOOK and WINDOW Transformations

2.3 Specifying Window Depth: Depth Clipping

So far you have redefined the X and Y dimensions of windows. The Z dimension of all the windows specified up to now has defaulted to 10^{-15} for the front boundary and to 10^{+15} for the back boundary. In this section, you will specify not only the X and Y boundaries of an orthographic window but the Z boundaries as well. The Z boundaries are specified as part of the WINDOW command.

The PS 390 automatically clips the top, bottom, right side, and left side of the window at the X and Y boundaries. However, clipping at the Z boundaries, known as depth clipping, does not automatically happen when you define Z boundaries for a displayed window. Portions of an object that fall in front of or in back of the Z boundaries are not clipped until depth clipping is enabled. Depth clipping is enabled by using the SET DEPTH CLIPPING command.

In an orthographic window, depth clipping can occur anywhere in positive and negative Z.

The SET DEPTH CLIPPING command is an operation node in the display structure. The node can be placed above the 4x4 WINDOW matrix because depth clipping operations are not matrix transformations (they are not overridden by a 4x4 matrix).

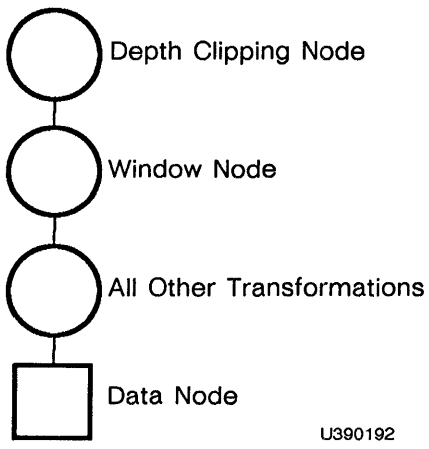


Figure 8-15. Set Depth Clipping Display Structure

2.3.1 Exercise 1

Include Z boundaries in an orthographic window by entering:

```
Change_Z :=  WINDOW   X=-1:1   Y=-1:1   FRONT=3 BACK=5
             APPLIED TO Car;

DISPLAY Change_Z;
```

The X and Y dimensions of Change_Z are the same as in the default window, but the Z dimensions define front and back boundaries at 3 and 5. Since the car extends from about -1 to about 1 in Z, none of it falls within the Z boundaries of Change_Z. However, you still see the car because depth clipping (set to OFF in default mode) is not in effect.

2.3.2 Exercise 2

To see only what is in the window, in this case from 3 to 5 in Z, enable depth clipping by entering:

```
REMOVE Change_Z;

Z_Clip := SET DEPTH_CLIPPING ON APPLIED TO Change_Z;

DISPLAY Z_Clip;
```

Now nothing appears on the screen because the car is outside the the Z dimensions of the window. The entire car has been clipped from view.

```
REMOVE Z_Clip;
```

2.4 Optimizing Depth Cueing

One of the ways the PS 390 gives the illusion of depth to an object is to vary the intensity between parts of the object that are near and those that are farther away. Near portions are brighter; portions farther away are gradually dimmed. This is called depth cueing. Refer to Figure 8-16.

The brightest intensity occurs at the front Z boundary (or clipping plane) and the dimmest intensity occurs at the back Z boundary. So maximum contrast in depth cueing is achieved when the Z boundaries are set close to the object in the window.

With depth clipping on, data between the eye and the front clipping plane will be clipped, data between the front clipping plane and back clipping plane will appear with an intensity gradient, and data behind the back clipping plane will be clipped.

With depth clipping off, data between the eye and front clipping plane will appear at full intensity, data between the front clipping plane and back clipping plane will appear with an intensity gradient, and data behind the back clipping plane will appear at minimum intensity.

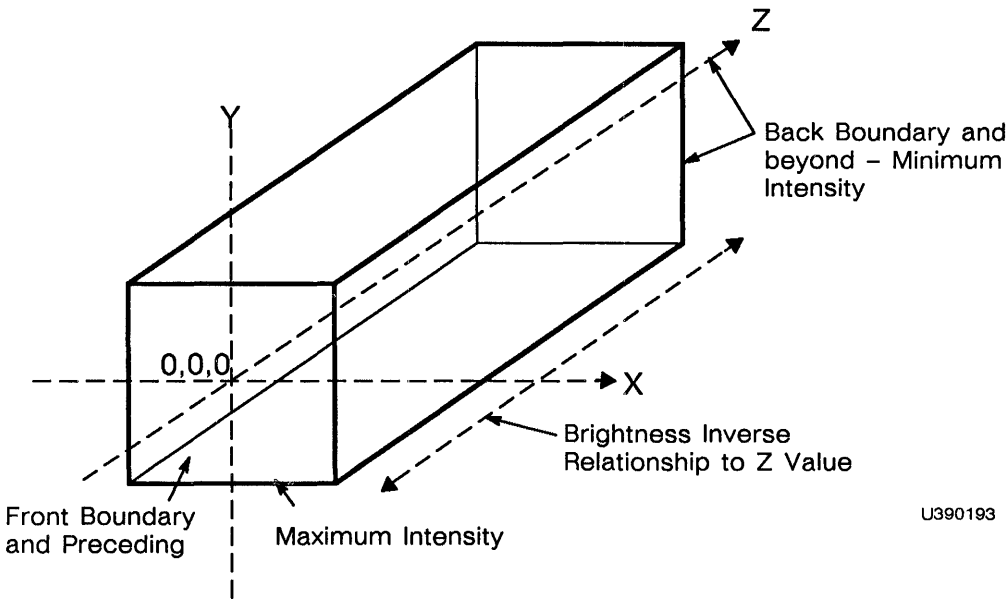


Figure 8-16. Intensity as a Function of Z Location

2.4.1 Exercise 1

Change the Z boundaries of the default WINDOW to see a change in depth cueing for the Car. First display the sports car in the default WINDOW, with Z boundaries at 10^{-15} and 10^{+15}. To make this easier to see, first rotate the car.

```
Rot_Car := ROTATE IN Y 110 APPLIED TO Car;

DISPLAY Rot_Car;
```

Depth cueing is apparent enough to make it difficult to see the back of the car. Now close in the Z boundaries around the car and display the new window.

```
Close := WINDOW   X=-1:1   Y=-1:1   FRONT=-.5  BACK=5
         APPLIED TO Rot_Car;

DISPLAY Close;
```

In Close, the front Z boundary is placed in negative Z (a placement that is legal only for orthographic windows).

```
REMOVE Close;

REMOVE Rot_Car;
```

2.4.2 Exercise 2

Refer to Section *GT3 PS 390 Tutorial Demonstrations* and run the WINDOW demonstration program.

2.5 Using a 4x4 Matrix to Specify an Orthographic Window

You can build your own 4x4 matrix in lieu of the one created by the WINDOW command by using the following MATRIX_4x4 command below. (The operation node this creates should be placed above all other matrix operations in a display structure branch, because a current matrix is overridden whenever a 4x4 matrix is encountered.)

```
Name := MATRIX_4X4
            m11,m12,m13,m14
            m11,m12,m13,m14
            m11,m12,m13,m14
            m11,m12,m13,m14

APPLIED TO Another_Name;
```

(For more details, refer to Section *RM1 Command Summary*.)

3. Defining Perspective Windows

The orthographic window is one of three possible ways to define a viewing area. With the orthographic window, the illusion of depth is created only by depth cueing.

The two other ways to define a viewing area employ perspective as well as depth cueing. In a perspective view, lines that go back from your eye point appear to be converging. So objects viewed in a perspective window appear smaller as they recede into the distance, further enhancing the illusion of depth and realism. The PS 390 defines perspective windows two ways: using the FIELD_OF_VIEW command and using the EYE BACK command.

Perspective windows are not box-shaped like orthographic windows. They are shaped like a pyramid, with your eye at the apex, extending into world coordinate space. The section of the pyramid in which objects are visible, called a frustum, is defined using front and back boundaries.

Figure 8-17 shows how a perspective window differs from an orthographic window:

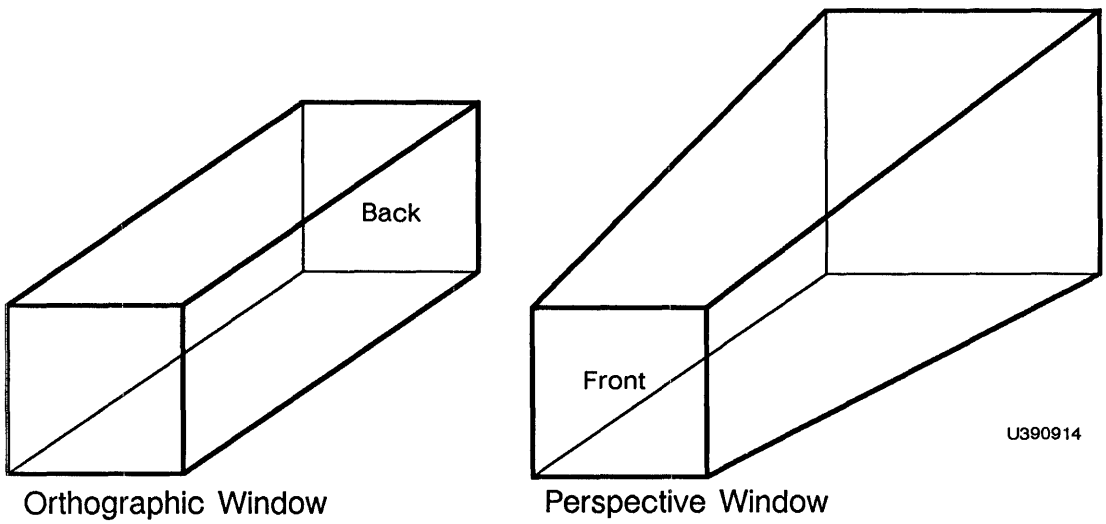


Figure 8-17. Orthographic Window Compared to Perspective Window

In a perspective window, the X,Y size of the front and back boundaries is not specified directly. Boundary size is determined by two factors.

The first factor is the size of the viewing angle—the angle between opposing sides of the viewing pyramid. As the viewing angle widens, the frustum of view encompasses more and more of the world coordinate system. So the wider the angle, the smaller an object appears relative to the viewing area. Also, since the angle opens equally in height and in width, the aspect ratio of perspective windows is always 1, width equal to height.

The second factor determining the size of a perspective window is the distance from the apex of the viewing pyramid (located at 0,0,0) to the front and back boundaries of the frustum and the distance between the front and back boundaries. See Figure 8-18.

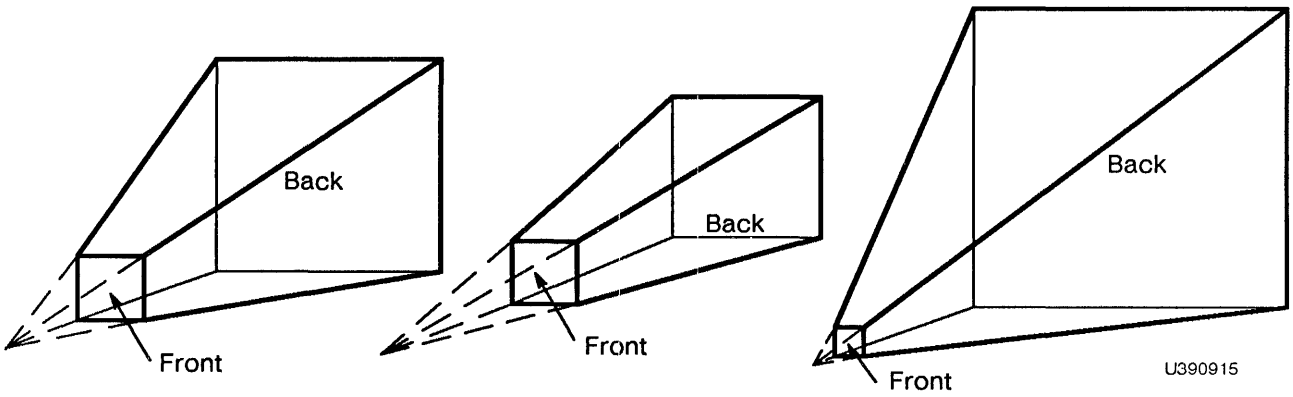


Figure 8-18. Angles Between Opposing Sides of the Pyramid

Unlike in an orthographic window, the front boundary of a perspective window cannot be placed behind your eyepoint (behind the LOOK FROM location). In perspective views, the front boundary cannot be at a location behind 10^{-15} in Z.

3.1 Using FIELD_OF_VIEW

The easiest way to define a perspective viewing area is using the FIELD_OF_VIEW command. A field of view is specified in terms of the viewing angle and the distance of the front and back boundaries from the eyepoint. This command imposes a perspective view on objects within the frustum of vision (the perspective window) it creates.

A field of view is like an orthographic window in that depth clipping does not occur in a field of view unless you set depth clipping on. And also, the intensity for depth cueing in a field of view is brightest at the front boundary and dimmest at the back boundary.

Lastly, like the orthographic window transformation, the field of view transformation is performed by a 4x4 matrix. This matrix is represented by an operation node, which must be above all other matrix transformation nodes in a display structure (see Figure 8-19).

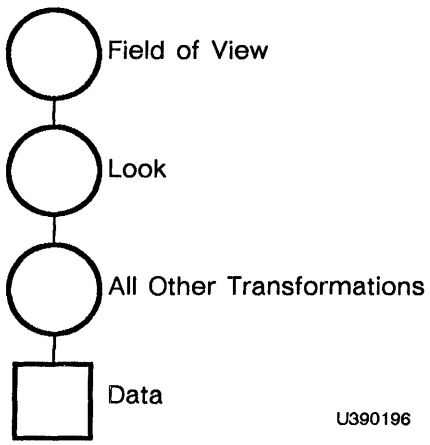


Figure 8-19. Display Structure With FIELD-OF-VIEW Node

For maximum depth cueing effects in a field of view, you must set the front and back boundaries close to the object. To do this, determine the distance from the eyepoint to the object being viewed and also how large the object is. If you place the AT point in the center of a large object and then position the front and back boundaries too close to it, parts of that object may be clipped from view.

If no LOOK transformation has been applied to the view, the distance to the object is its location along the positive Z axis—the default line of sight. If you have defined a line of sight with a LOOK transformation, you must calculate the distance between the AT and FROM points so you will know where to place the front and back boundaries. To calculate this distance, find the differences between the X, Y, and Z values of the FROM point and the AT point, square those differences, add them, and find the square root of that sum.

For example, if you are looking from (–2,2,0) at a one-unit radius sphere centered at (3,–2,–1), the FROM/AT distance is the square root of: 5 squared, plus 4 squared, plus 1 squared, or 6.48. For maximum depth cueing, place the near boundary (zmin) at 5.48 and the zmax boundary at 7.48 (see Figure 8-20).

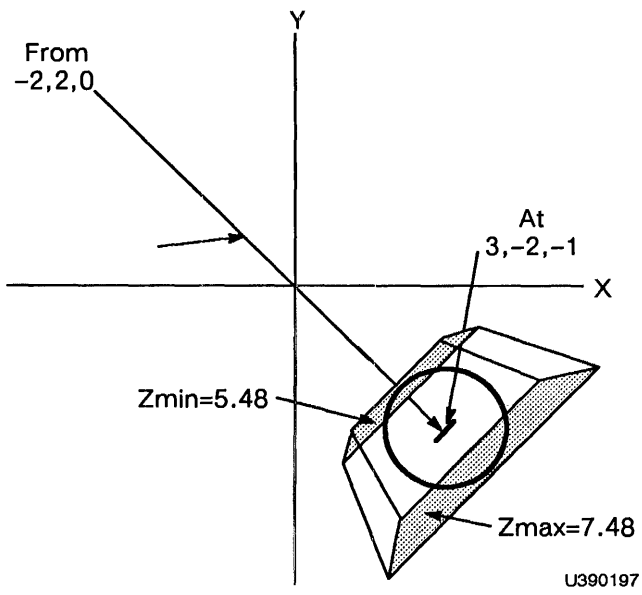


Figure 8-20. Setting Z Boundaries for Maximum Depth Cueing

The result of the LOOK command is, of course, to place FROM at 0,0,0 and AT on the positive Z axis; thus, the Zmax, Zmin designations.

3.1.1 Exercise 1

Position the sports car in a perspective window by specifying a FIELD_OF_VIEW and position the car within the frustum of vision using a LOOK command.

```
Perspective := FIELD_OF_VIEW 28 APPLIED TO Look;

Look :=  LOOK AT 0,0,0 FROM 0,0,-5
         APPLIED TO Car;

DISPLAY Perspective;
```

No front or back (Z) boundaries are specified. Because their default value is 10^{-15} and 10^{+15}, the car appears to be dim.

The 28 in the command is the number of degrees in the angle between opposing sides of the viewing pyramid. Twenty-eight degrees is approximately the actual viewing angle from your eye to the edges of the PS 390 screen at a comfortable viewing distance.

The LOOK (named Look) has the effect of translating the car forward 5 degrees in Z and placing the FROM point at the same location as the apex of the viewing pyramid (0,0,0). The Z axis runs down the center of the pyramid (Figure 8-21).

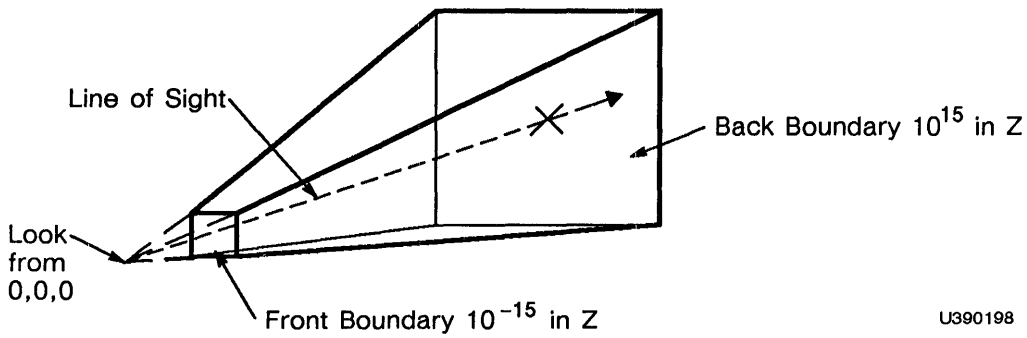


Figure 8-21. Using FIELD_OF_VIEW with LOOK

3.1.2 Exercise 2

Change Perspective to specify different front and back boundaries by entering:

```
Perspective := FIELD_OF_VIEW 28
               FRONT = 4.5
               BACK = 7
               APPLIED TO Look;
```

Since the LOOK (named Look) moves the car forward so that it is centered around 5 in Z, placing the front and back boundaries at 4.5 and 7 in Perspective closes the boundaries around Car, maximizing depth cueing. The part of the car nearest to the front boundary appears brighter. Figure 8-22 shows the car in the frustum of vision just created.

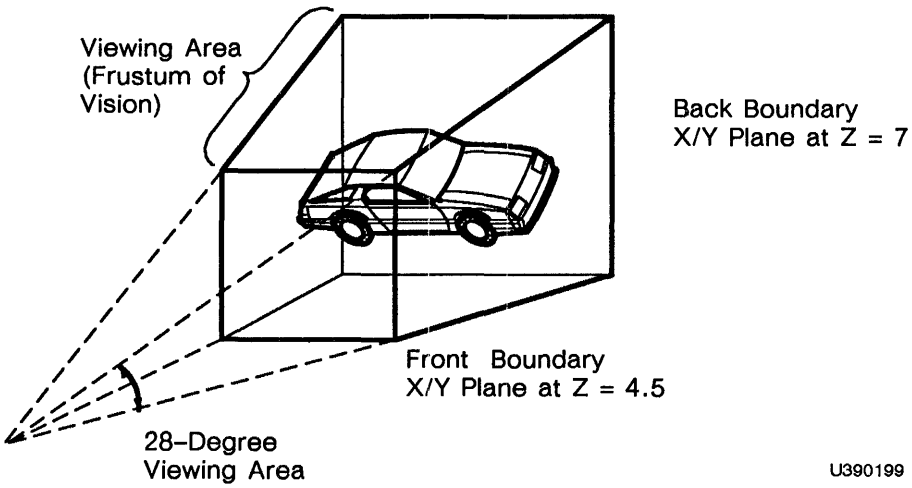


Figure 8-22. Setting Front and Back Boundaries

3.1.3 Exercise 3

Refer to Section *GT3 PS 390 Tutorial Demonstrations* and run the FIELD_OF_VIEW demonstration program. Before you begin, remove Perspective. Enter:

```
REMOVE Perspective;
```

3.2 Using the EYE BACK Command

In addition to FIELD_OF_VIEW, there is another command that creates a perspective window. Like FIELD_OF_VIEW, the EYE BACK command specifies a pyramid-shaped viewing area with front and back clipping planes.

In addition, it allows you to move the eyepoint back from, above, below, and to the side of screen center. This also moves the line of sight established by the LOOK transformation, keeping the line of sight parallel to a line straight through the center of the screen (where most lines of sight are situated). This effect means that you may not see what you are LOOKing AT. The EYE BACK command is the only viewing command that has the effect of moving the line of sight, established by the LOOK transformation, somewhere other than directly through the center of the screen.

Imagine yourself in a room looking out through a porthole. The EYE BACK command simulates a view from any position in the room through this porthole and into the world coordinate system. Distance and location through the porthole (that is, FRONT and BACK BOUNDARIES) are measured in the usual PS 390 coordinate system units. Inside the room, distance is measured in relative room coordinates. These relative room coordinates are used to create the proper proportions for the viewing pyramid in the world coordinate system.

What you see—the viewing area—is determined by the line of sight established in the LOOK transformation, the size of the porthole, your distance back from it, and your position in the room with respect to its center. The closer you are to the porthole, the larger the viewing area. The EYE BACK command allows you to adjust how far back and/or off-center you are from the center of the porthole. As with all windowing commands, you may also specify front and back boundaries.

From where you stand in the room, distance and screen width are specified in terms of relative room coordinates. These coordinates are important in terms of the ratios they establish, which determine the viewing angle. For example, in Figure 8-23 the ratio of screen width to eyeback distance is 2:2. A screen width of 4 and eyeback distance of 4 would establish the same ratio (2/2=1; 4/4=1) and so the same view. (Figure 8-23).

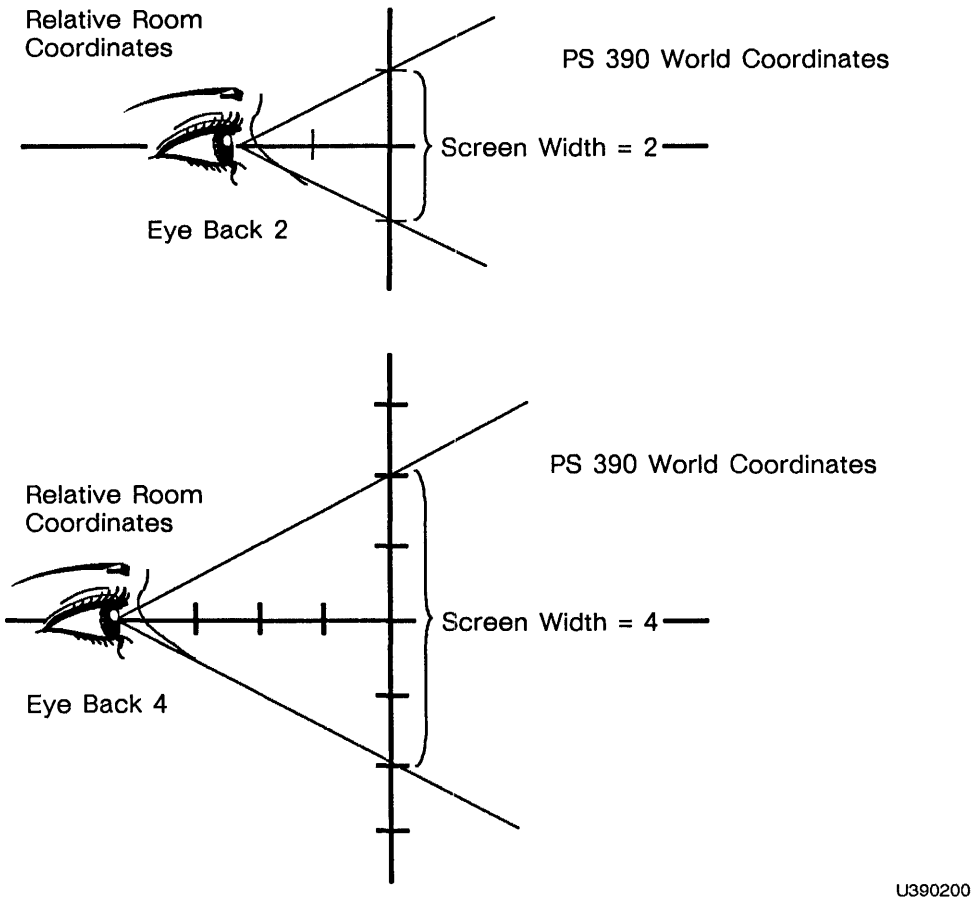


Figure 8-23. Relative Room Coordinates

The line of sight established by the LOOK transformation may not point at what you are looking at when you use the EYE BACK transformation. The eye transformation creates its own sightline relative to the line of sight established by the LOOK transformation. As shown in Figure 8-24, the LOOK transformation establishes a line of sight to the viewed object. With EYE BACK, however, the new line of sight may be different. So, you may not see what you are "LOOKing AT." (You may be LOOKing AT Car 1, but see Car 2.)

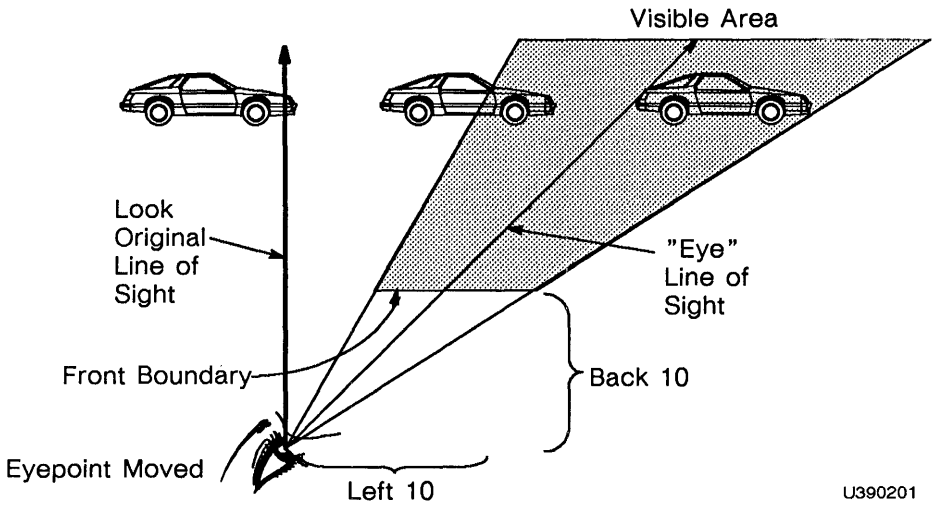


Figure 8-24. Line of Sight for LOOK and EYE BACK

In the simplest instance of using the EYE BACK command, you specify only the distance from the screen (back) and the screen width (wide). The ratio of these two determines how much of the world coordinate system is viewable (viewing angle) and the orientation of the viewing pyramid. (This is effectively another way to specify a view that can be specified using FIELD_OF_VIEW.) In such a view, the line of sight established by the LOOK transformation would aim through the center of the screen toward the AT point.

In part A of Figure 8-25, at least part of all four cubes appears in the viewing area. When the eyepoint is moved further back in part B, only two of the cubes are viewable, but they appear to be larger than in part A.

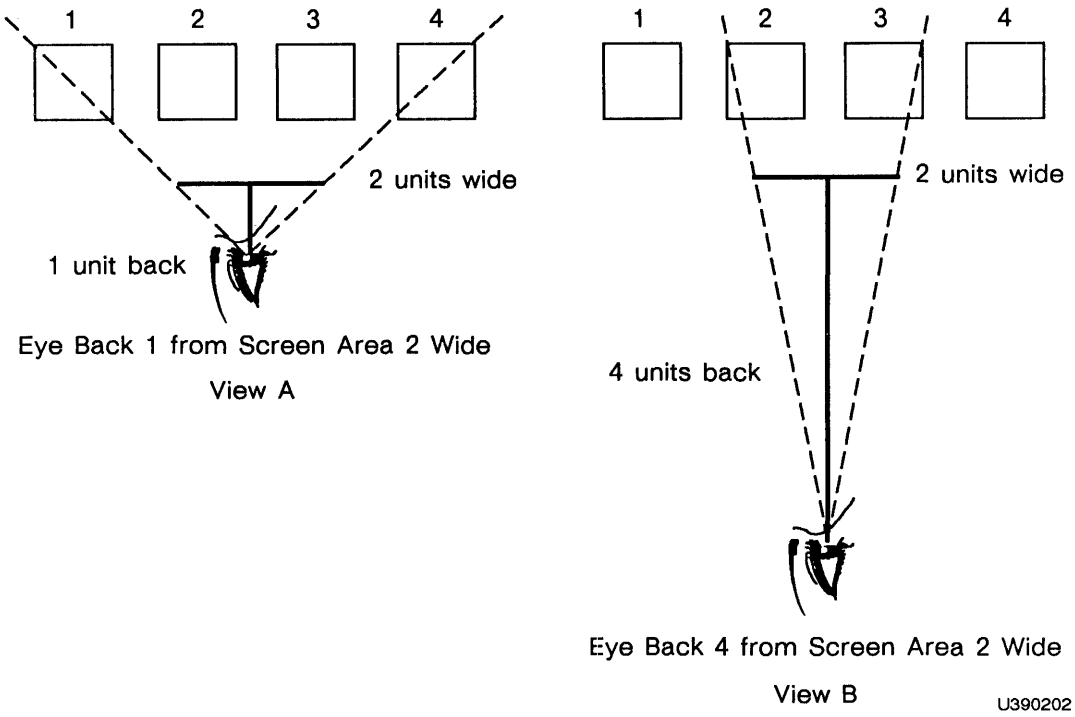


Figure 8-25. Specifying the Viewing Angle

Moving the eyepoint so that it is not directly over the center of the screen, results in a different portion of the world coordinate system coming into view. For example, in Figure 8-26, moving the eyepoint back 1 unit and left 2 units has shifted the viewing so that no part of cube 1 is visible and most of cube 4 has come into view.

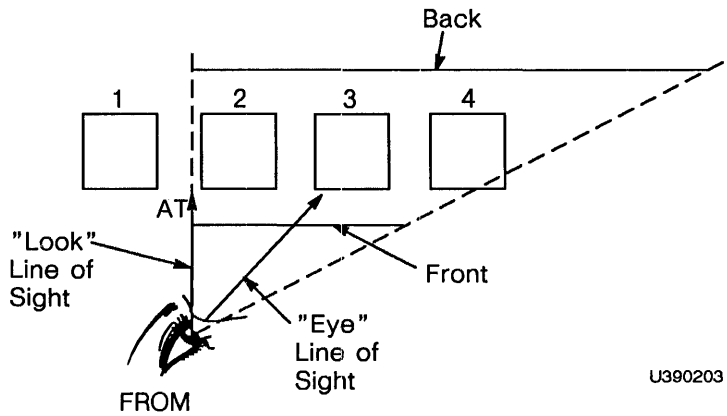


Figure 8-26. Moving Eyepoint Back and Left

As with FIELD_OF_VIEW, you must set boundaries correctly with the EYE BACK command to have an object appear. As expected, if depth clipping is not in effect, any object in front of the front boundary appears at full intensity; anything between boundaries diminishes in brightness as it approaches the back boundary; and everything behind the back boundary appears at minimum brightness.

As with the FIELD_OF_VIEW, boundaries are specified in world coordinate system units measured from 10^{-15} in Z (the center of the screen after the LOOK transformation is applied).

Note that with the EYE BACK command, Z boundaries remain orthogonal to the Z axis. For example, in Part A of Figure 8-27, though the eyepoint has been moved farther back, the boundary is still placed 6 units from the original FROM point (0,0,0) at the center of the screen. This is also the case in Part B, where the eyepoint has been moved back and to the left. Even when EYE BACK changes the line of sight, the boundaries do not shift. Instead, the viewing area, the frustum of vision, becomes skewed.

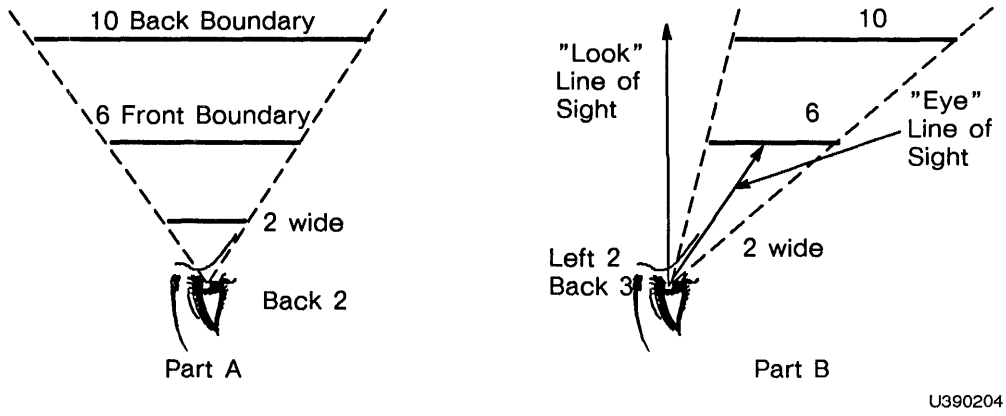


Figure 8-27. Boundaries Using the EYE BACK Command

3.2.1 Exercise 1

Run the LOOK demonstration program. (Refer to Section *GT3 PS 390 Tutorial Demonstrations.)*

3.2.2 Exercise 2

Create instances of Car to the right and the left of the original sports car and group all three instances under the name Three_Cars.

```
Car2 := TRANSLATE BY 3,0,0  APPLIED TO Car;

Car3 := TRANSLATE BY -3,0,0  APPLIED TO Car;

Three_Cars :=  INSTANCE OF Car, Car2, Car3;
```

View Three_Cars using the LOOK and EYE BACK commands. First, establish a line of sight (Look1).

```
Look1 := LOOK AT 0,0,0 FROM 0,0,-10
         APPLIED TO Three_Cars;
```

This places the three cars 10 units away from your eyepoint. Now apply an EYE BACK command to view the cars through a porthole 1 room unit wide from a distance of 2 room units.

Notice the following three commands include values for the front and back boundaries. The sports cars have been placed in front of the front boundary (depth clipping is off by default) to appear at maximum intensity.

```
Eye_Locate := EYE
              BACK 2
              RIGHT 0      {default}
              UP 0         {default}
              SCREEN 1 WIDE
              FRONT = 9.5
              BACK = 10.5
              THEN Look1;

DISPLAY Eye_Locate;
```

You can see the original Car, but Car2 and Car3 are partially clipped on the right and the left sides, respectively, of the window. See Figure 8-28.

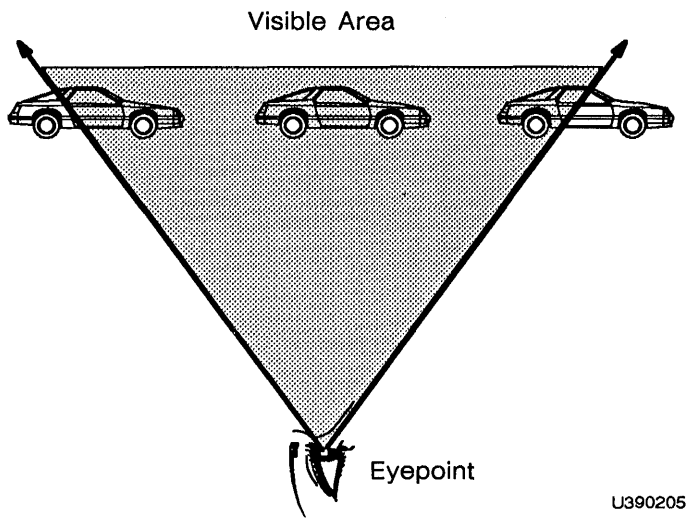


Figure 8-28. EYE BACK View of Cars

Now move your eyepoint to the left far enough to see all of Car2 (which is partially visible to the right of the present window).

```
REMOVE Eye_Locate;

New_Eye :=  EYE
            BACK 2
            LEFT .5      {or RIGHT -.5}
            UP 0         {default}
            SCREEN 1 WIDE
            FRONT = 9.5
            BACK = 10.5
            THEN Look1;

DISPLAY New_Eye;
```

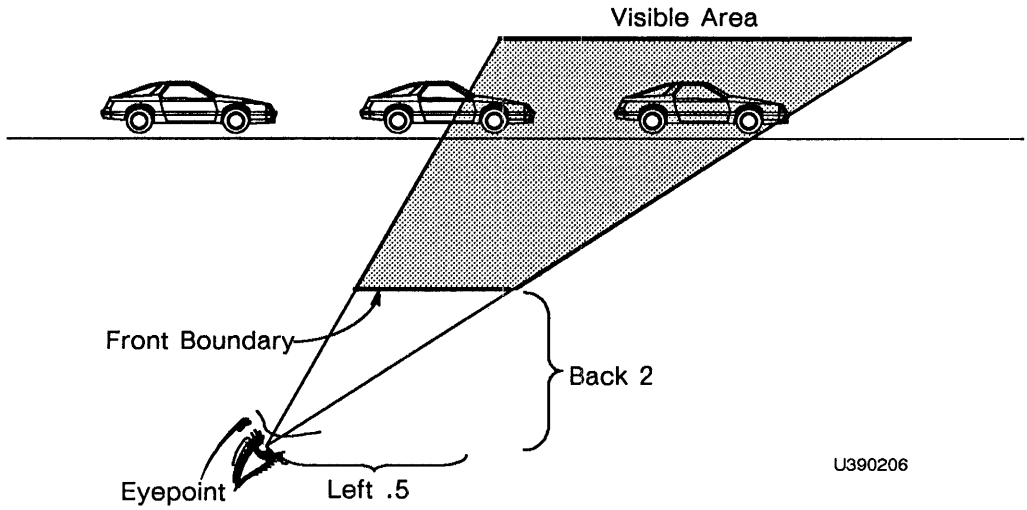


Figure 8-29. EYE BACK View of Car2

Now, look at Car3 (which is partially visible to the left of the present screen) by moving your eye to the right.

```
REMOVE New_Eye;

Last_Eye := EYE
            BACK 2
            RIGHT .5     {or LEFT -.5}
            UP 0
            SCREEN 1 WIDE
            FRONT = 9.5
            BACK = 10.5
            THEN Look1;
DISPLAY Last_Eye;
```

What you see on the screen is in correct perspective only if your actual position in the room is approximately where you specified your eye location to be in the EYE BACK command. In the last example, the values in the EYE BACK command are .5 right, back 2 from a screen 1 wide. You would need to move your head right one-half of a screen width and back two widths from the center of the PS 390 screen to view the cars in correct perspective. (Note in this case, you will not be able to see the AT point specified by the LOOK command.)

If you remain seated at the PS 390 looking into the center of the screen, displayed objects may appear distorted or skewed when the eyepoint is changed. This is because you are looking at what should be an oblique view from a position that would not normally create an oblique view.

3.3 Using a 4x4 Matrix to Specify a Perspective Window

The EYE BACK transformation is a 4x4 matrix operation that is represented by an operation node. This node must be above all other transformation nodes in a display structure. The EYE BACK operation node should also be directly above the LOOK operation node in the display structure.

You can build your own customized 4x4 matrix in lieu of the one created by the FIELD_OF_VIEW or EYE BACK command by using the following MATRIX_4x4 command:

```
MATRIX_4x4:= m11,m12,m13,m14
             m11,m12,m13,m14
             m11,m12,m13,m14
             m11,m12,m13,m14  APPLIED TO Another_Name;
```

(For more details, refer to the Section *RM1 Command Summary.*)

4. Specifying a Viewport

In addition to the two types of viewing transformations, establishing a line of sight and specifying a viewing window, the PS 390 lets you specify portions of the full screen in which windows are displayed. The PS 390 raster screen allows display of both antialiased wireframe models, and shaded renderings. Because the commands and operations governing the display of each kind of model are inherently different, there are two types of viewports used for display. The dynamic viewport is used for the display and manipulation of wireframe models, while the static viewport allows display of hidden-line images and shaded renderings.

Either an orthographic or perspective window can be displayed within either a dynamic or a static viewport. Up to this point, all windows specified in examples have been projected onto the full dynamic screen of the PS 390. The PS 390 maps a window to the full dynamic screen by default if no smaller portion of the screen is specified. The area of the screen that has the window mapped to it is called a viewport.

The process of mapping a window to a viewport is not a matrix operation. Because of this, the viewport specification can be placed virtually anywhere in relation to matrix operations in a display structure. A logical placement, though, is above the windowing transformation.

4.1 Dynamic Viewport Operations

The following operations are performed in the dynamic viewport:

- Real-time manipulation of vector or wireframe representations of polygonal models.
- Cross-sectioning defined by the sectioning plane (solid wireframe polygonal model).
- Sectioned rendering (wireframe polygonal model).
- Backface removal (solid wireframe polygonal model).

Operations involving polygonal models are discussed fully in Section *GT13 Polygonal Rendering.*

4.2 Specifying a Dynamic Viewport

Dynamic viewports are specified using the VIEWPORT or the LOAD_VIEWPORT command. The VIEWPORT command defines a dynamic viewport in terms of the current viewport. Values of the new viewport must be within the -1 to 1 range of the current viewport, implying that each viewport may be no larger than its predecessor. The LOAD_VIEWPORT command however, defines a dynamic viewport relative to the full PS 390 screen.

The dimensions of the current viewport are always -1 to 1 in width and -1 to 1 in height, with the center of the viewport corresponding to 0,0. (See Figure 8-30.)

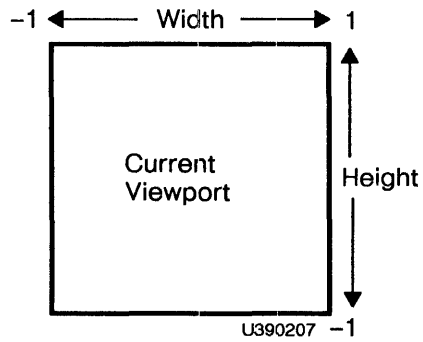


Figure 8-30. Current Viewport Dimensions

This holds true with both the VIEWPORT and LOAD_VIEWPORT commands. The default intensity range available for any dynamic viewport is from 0 to 1, or from minimum to maximum intensity. This intensity is spread over the range from the front boundary to the back boundary of the window being displayed in the viewport. The values for viewport dimensions and intensity ranges have nothing to do with world coordinate values.

4.2.1 Exercise 1

First display the Car in the default full-screen dynamic viewport by entering:

```
INITIALIZE DISPLAY;

DISPLAY Car;
```

The car is now displayed in the current viewport, which is -1 to 1 in height and in width.

4.2.2 Exercise 2

Using the VIEWPORT command, define a viewport to be the upper right corner of the default full-screen viewport by entering;

```
Port2 := VIEWPORT
         HORIZONTAL=0:1
         VERTICAL=0:1    APPLIED TO Car;

DISPLAY Port2;

REMOVE Car;
```

By using the VIEWPORT command, Port2 was defined in terms of the current viewport. Now the upper right corner of the screen becomes the current viewport and the default window is mapped to it (Figure 8-31).

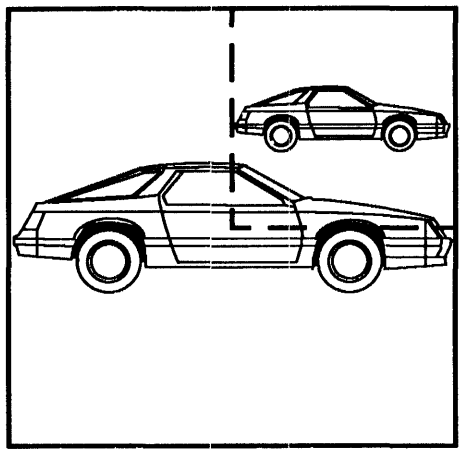


Figure 8-31. Port2 – Upper Right Quadrant

The display structure for this viewport applied to Car is shown in Figure 8-32.

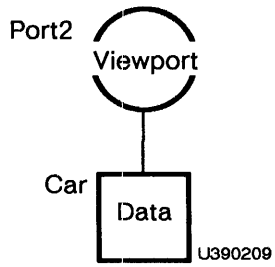


Figure 8-32. Display structure for Port2

4.2.3 Exercise 3

Define another viewport in terms of the now current viewport (Port2).

```
Port3 := VIEWPORT
         HORIZONTAL=0:1
         VERTICAL=0:1    APPLIED TO Port2;

DISPLAY Port3;

REMOVE Port2;
```

Port3 is now the upper right quadrant of Port2, which is the upper right quadrant of the default full-screen viewport. Figure 8-33 shows the associated display structure.

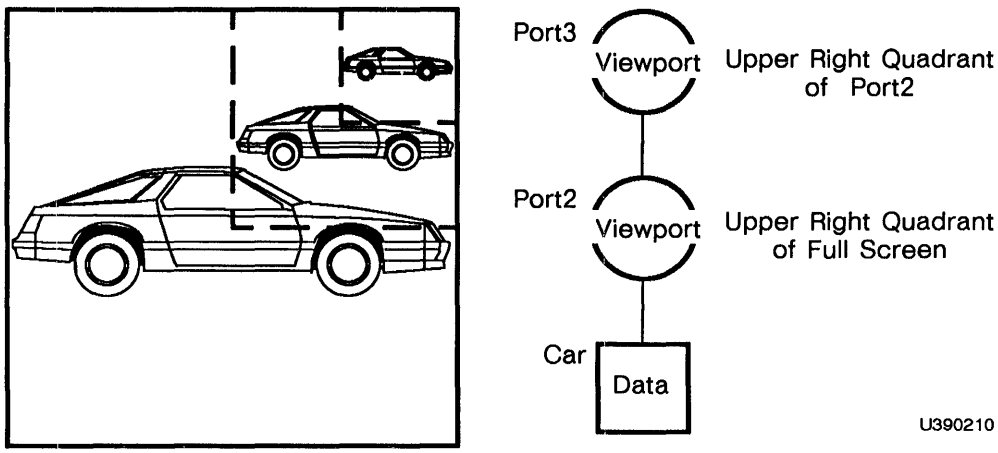


Figure 8-33. Port3 and Associated Display Structure

To define a viewport independent of the current viewport, the LOAD_VIEWPORT command is used. This specifies a viewport relative to the entire PS 390 screen. Viewport specification using the LOAD_VIEWPORT command does not restrict the user from making a larger viewport after making a smaller one, as is the case with the VIEWPORT command.

4.2.4 Exercise 4

Using the LOAD_VIEWPORT command define the current viewport to be the full dynamic screen (Full_View).

```
Full_View := LOAD_VIEWPORT
             HORIZONTAL= -1:1
             VERTICAL = -1:1   APPLIED TO Port3;

DISPLAY Full_View;

REMOVE Port3;
```

The display structure for this is shown in Figure 8-34. Note that the LOAD_VIEWPORT command overrides the previous viewports specified with the VIEWPORT command.

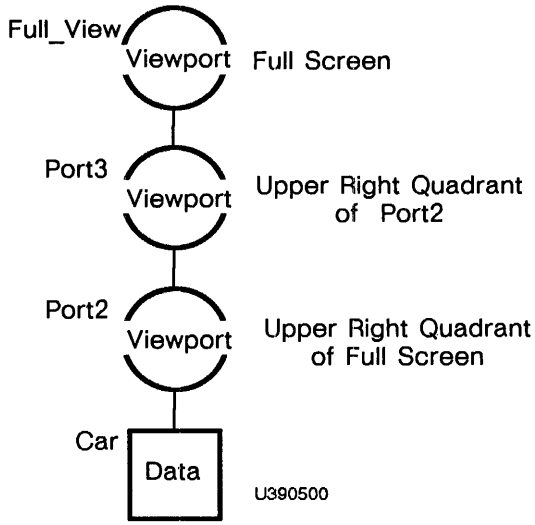


Figure 8-34. Display Structure for Full_View

4.2.5 Exercise 5

Define the equivalent of viewport Port3 using the LOAD_VIEWPORT command (Port4).

```
Port4 := LOAD_VIEWPORT
         HORIZONTAL = .5:1
         VERTICAL = .5:1   APPLIED TO Full_View;

DISPLAY Port4;

Remove Full_View;
```

Port4 is in the upper right of the screen. Figure 8-35 shows the display structure and equivalent viewport of Port4.

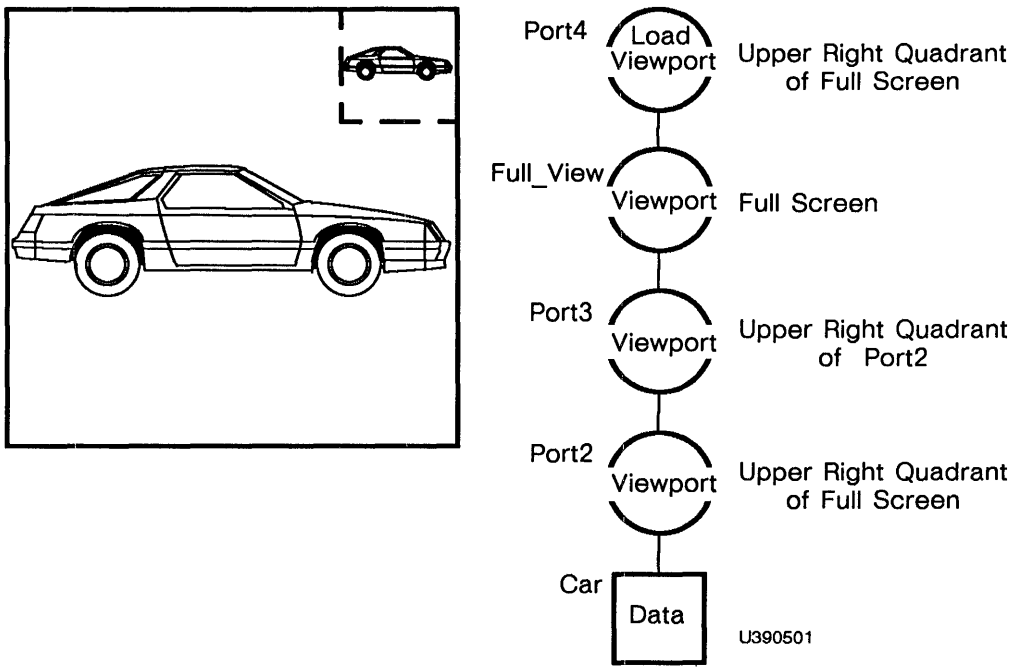


Figure 8-35. Port4 and Associated Display Structure

Before going on to the next section, remove the data structures from the display list. Enter the INITIALIZE DISPLAY command:

```
INITIALIZE DISPLAY;
```

4.3 Dynamic Viewport Considerations

Although the raster screen contains 1024 by 1024 addressable pixels, the actual displayable area on the raster screen is a rectangle, with pixel addresses going from 0 to 1023 in X and to 863 in Y, where the physical pixel address 0,0 is in the lower left corner. A PS 390 viewport which spans (-1,1) in both vertical and horizontal directions maps onto the full 1024 x 1024 screen so that a rectangular portion along the lower edge of the viewport is not displayed. To avoid this situation, all viewports in the display structure are initially concatenated with a default viewport in the top display structure which maps to a square of 864 x 864 (Figure 8-36).

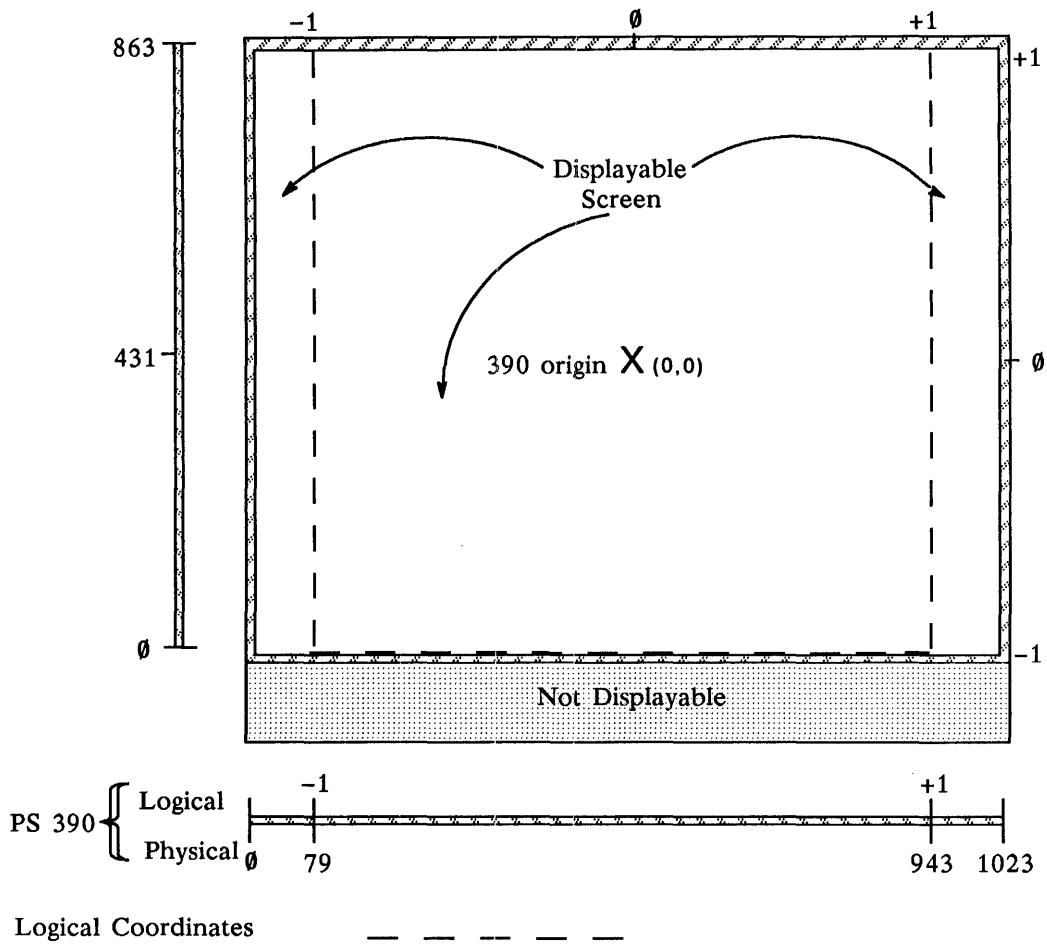


Figure 8-36. PS 390 Display

4.3.1 Overriding the Default Viewport

If you want to override the default viewport and use the entire displayable rectangular screen area as a non-square viewport, the following commands can be entered:

```
Configure A;
VPF1$ :=  Viewport Horizontal = -0.998:.998
                   Vertical = -0.685:1
                   Intensity = 0:1
Then HVP1$;
Finish Configuration;
```

This will cause all the subsequent VIEWPORT and LOAD_VIEWPORT commands in the structure to be concatenated with this rectangular viewport. In doing so, however, your data must account for the nonsquare viewport.

To re-establish the default viewport, use either the commands

```
Configure A;
VPF1$ := Viewport Horizontal = -0.8425:0.8425
         Vertical = -0.685:1
         Intensity = 0:1
Then HVP1$;
Finish Configuration;
```

or

```
Screensave := F:Screensave;
```

Note that the INITIALIZE command does not restore the original viewport. Also note that you cannot override the default viewport with the LOAD_VIEWPORT command.

4.4 Operations in the Static Viewport

The following types of rendering styles are displayed in the static viewport:

- Wash shading
- Flat shading
- Gouraud shading
- Phong shading
- Raster hidden-line removal

A complete description of rendering operations can be found in Section *GT13 Polygonal Rendering.*

4.5 Specifying a Static Viewport

There is no PS 390 specific command to specify static viewports. Specifying a static viewport boundaries is done by sending a 3D vector to input <3> of SHADINGENVIRONMENT. Refer to section *RM3 Initial Function Instances* for a complete description of the SHADINGENVIRONMENT function.

A 3D vector sent to input <3> of SHADINGENVIRONMENT specifies viewport pixel values. For example,

```
Send V3D(80,0,863) to <3>SHADINGENVIRONMENT;
```

would be a valid command to specify a square static viewport of 864 by 864 pixels on the PS 390. Static viewports cannot be nonsquare; they must always be specified as a square viewport. Specifying a viewport by sending to input <3> of SHADINGENVIRONMENT is completely independent from dynamic viewport specifications as it does not create a viewport node in the display structure.

4.6 Clearing Viewports to Static or Dynamic

It is also possible to clear either the current viewport or the entire PS 390 screen and specify if the viewport is to be treated as a dynamic or static viewport. This is done by sending an integer or Boolean to input <7> of SHADINGENVIRONMENT.

4.6.1 Clearing to Static

Sending either a True or a fix (0) to input <7> clears the entire screen to static and causes a screen wash with the current static background color. Sending a False or fix (1) to input <7> clears only the currently specified static viewport and causes the viewport to be filled with the current static background color. When requesting another rendering to be displayed in a current static viewport, it is not necessary to clear the viewport first, since this is accomplished by requesting a new rendering. Refer to Section *GT13 Polygonal Rendering* for more information.

4.6.2 Clearing to Dynamic

Sending a fix (2) to input <7> clears the entire screen to dynamic and causes a screen wash with the current dynamic background color. This must be done to clear either a shaded image or a dynamic image (from the entire screen or viewport), before displaying a new dynamic image. Sending a fix (3) to input <7> clears only the currently specified dynamic viewport with the current dynamic background color. To prevent images from being corrupted, do not display an image or use the terminal emulator in an area of the screen that already has an image without first clearing the screen (or viewport) with a dynamic wash. Refer to Section *RM3 Initial Function Instances* for more details on the SHADINGENVIRONMENT function.

4.7 Displaying Multiple Viewports

The PS 390 allows multiple combinations of dynamic and static viewports to be displayed simultaneously. Because of this flexibility in defining the usable screen space, it is necessary that the code be organized so that images or viewports do not overlap each other. The overlapping of dynamic images on static viewports will corrupt the static image.

To illustrate the capability of multiple viewports, the exercises that follow create four dynamic views that can be displayed simultaneously. The four views are:

- In the lower left quadrant, the Car is displayed as a side view in an orthographic window.
- In the lower right quadrant, the Car is displayed as a front view in an orthographic window.
- In the upper right quadrant, the Car is displayed as a top view in an orthographic window.
- In the upper left quadrant, the Car is displayed in a perspective window.

4.7.1 Exercise

Create the four views by applying the following VIEWPORT definitions:

```
DISPLAY Four_View;

Four_View := INSTANCE OF Side, Front, Top, Persp;

Side := VIEWPORT
          HORIZONTAL= -1:0
          VERTICAL= -1:0    APPLIED TO Car;

Front := BEGIN_STRUCTURE
           VIEWPORT
              HORIZONTAL= 0:1
              VERTICAL= 0:-1;
           LOOK
              AT 0,0,0
              FROM .1,0,0    APPLIED TO Car;
         END_STRUCTURE;
```

```
Top := BEGIN_STRUCTURE
         VIEWPORT
            HORIZONTAL= 0:1
            VERTICAL= 0:1;
         LOOK
            AT 0,0,0
            FROM 0,.1,0     APPLIED TO Car;
       END_STRUCTURE;

Persp := VIEWPORT
         HORIZONTAL= -1:0
         VERTICAL= 0:1     APPLIED TO Perspective;
```

If you have rebooted, changed modes, or initialized the system since you began this section, you will need to add the two following lines of code to the above listing:

```
Perspective := FIELD_OF_VIEW 28 FRONT=4.5 BACK=7
               APPLIED TO Look;

Look := LOOK AT 0,0,0 FROM 0,0,-5 APPLIED TO Car;
```

4.8 Using Nonsquare Viewports

Nonsquare viewports are applicable to dynamic viewports only. Sometimes a nonsquare viewport is needed. The dimensions of -1 to 1 in width and height apply to nonsquare viewports as well (see Figure 8-37).

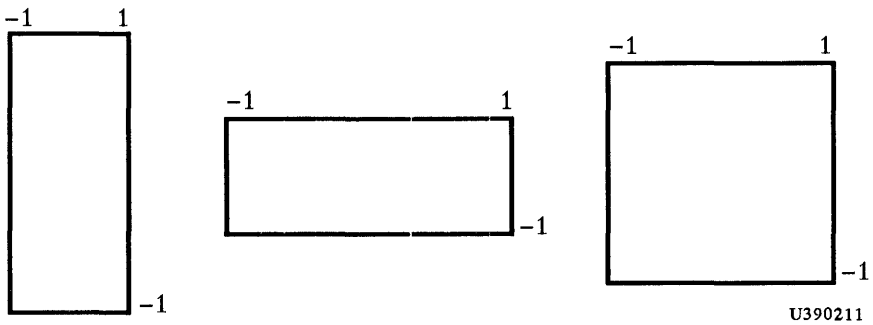


Figure 8-37. Dimensions of a Nonsquare Current Viewport

A nonsquare viewport can cause distortion of displayed data. To compensate for such distortion, objects can be viewed in a nonsquare window. This window must have the same height to width ratio (aspect ratio) as the viewport. For example, if the aspect ratio of a viewport is 1:2, half as wide as it is high, the window displayed in the viewport must also be half as wide as it is high to eliminate distortion that results from viewport mapping.

Orthographic windows are the only windows that can have nonsquare front boundaries. Perspective windows always have square front boundaries, so objects are distorted if a perspective window is displayed in a nonsquare viewport.

Unmatched aspect ratios can sometimes be used to advantage. A variety of effects can be achieved using this distortion. Cubes can become bricks in a viewport that is wider than it is high. Circles can become ellipses; econo-sedans can become sleek sports cars.

4.8.1 Exercise 1

Map a square window to a nonsquare dynamic viewport to observe the resulting distortion. First impose the PS 390 default orthographic (square) window by removing the previous perspective view:

```
REMOVE Four_View;
```

Then create the nonsquare viewport:

```
Nonsquare := VIEWPORT
             HORIZONTAL=-.5:.5
             VERTICAL=-1:1   APPLIED TO Car;

DISPLAY Nonsquare;
```

The default window around the Car is compressed to fit in the width of the narrow viewport. The result is distortion: a tall car (see Figure 8-38).

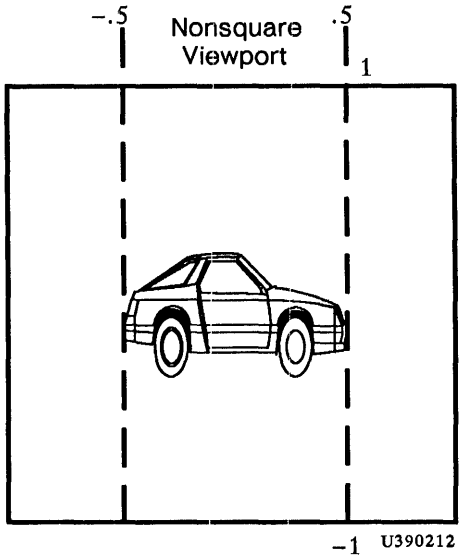


Figure 8-38. Square Window Mapped to a Nonsquare Viewport

4.8.2 Exercise 2

Compensate for the distortion by creating a nonsquare window for the nonsquare viewport by entering:

```
Nonsquare_Window :=  WINDOW
                     X = -1:1
                     Y = -2:2  APPLIED TO Nonsquare;

DISPLAY Nonsquare_Window;

REMOVE Nonsquare;
```

When this window is applied to the viewport, its aspect ratio is equivalent to the aspect ratio of the viewport, so the car appears in the nonsquare viewport without distortion (see Figure 8-39).

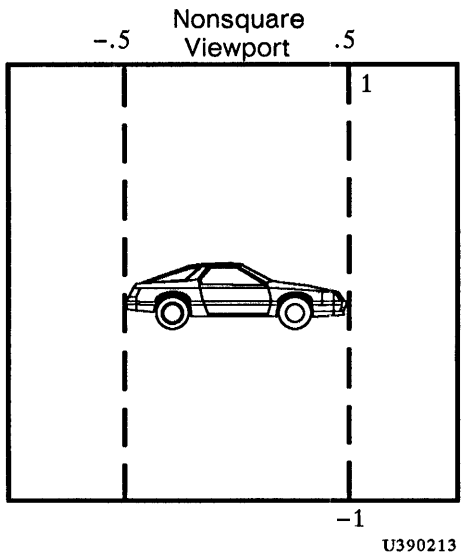


Figure 8-39. Nonsquare Window Mapped to a Nonsquare Viewport

To clear the display enter:

```
REMOVE Nonsquare_Window;
```

4.9 Setting an Intensity Range for a Window in the Dynamic Viewport

A dynamic viewport specification can also set an intensity range for the window displayed in the viewport. This intensity mapping is another facet of the window-to-viewport mapping process.

Remember that the maximum and minimum intensities for an orthographic or perspective window are anchored at the front and back boundaries of the displayed window. The default intensity range is from 0 (dimmest, back boundary) to 1 (brightest, front boundary).

Set the dynamic viewport boundaries to the upper right quadrant of the screen. To change the maximum and minimum intensities, compress the intensity range from .25 (quarter) to .75 (three-quarters). The car in the viewport will appear slightly dimmer.

```
Display Car;

New_Range :=  VIEWPORT
              HORIZONTAL=0:1
              VERTICAL=0:1
              INTENSITY=.25:.75
              APPLIED TO Car;

Display New_Range;
```

The intensity ranges of nested viewports affect each other. If Viewport2, with a range of .25 to .75, is defined in terms of a current viewport having an identical intensity range of .25 to .75, Viewport2 will have an intensity range of .375 to .625.

```
REMOVE New_Range;
REMOVE Car;
```

5. Viewing Attributes

You are now familiar with viewing transformations, which let you create any number of views of objects ... and with viewports, which allow you to display objects anywhere on the screen. The last set of viewing operations you can specify add a further range of possibilities to the images that are displayed. These operations let you set attributes in the structure of a model to enhance its usefulness.

In particular, viewing attributes allow you to specify the:

- Intensity at which lines are drawn in the dynamic viewport
- Colors of lines that form the image in the dynamic viewport

Viewing attributes differ from viewing transformations (line of sight and windows) in that they are not matrix operations. Consequently, they can be placed above windows (WINDOW, FIELD_OF_VIEW, EYE BACK) and LOOK transformations in a display structure.

5.1 Setting Intensity

Remember that with the VIEWPORT and the LOAD_VIEWPORT commands, an intensity range can be specified which applies to the window being displayed in the current dynamic viewport. In addition to this method, dynamic viewport intensity can be manipulated using the SET INTENSITY attribute.

The SET INTENSITY attribute is a non-matrix operation that overrides and replaces the intensity range set in the viewport specification. In fact, SET INTENSITY can be switched on and off, allowing you to easily and directly switch intensities between the values in the viewport specification and the values in the SET INTENSITY node.

A set intensity node can be switched on and off via function networks. SENDing (or CONNECTing) a Boolean value to a SET INTENSITY node toggles the ON/OFF condition of the node. (Refer to Section *RM1 Command Summary* for details.)

In a series of SET INTENSITY commands, the last one ON determines the intensity range in effect. For example:

```
One := BEGIN_STRUCTURE
       a := VIEWPORT
              HORIZONTAL=-1:1
              VERTICAL=-1:1
              INTENSITY=.5:1;
       b := SET INTENSITY ON 0:1;
       c := SET INTENSITY ON 1:1;
       INSTANCE OF object;
       END_STRUCTURE;

DISPLAY One;
```

When One is displayed, the intensity range is 1:1, the last specified intensity range.

A SET INTENSITY OFF command does not cancel a previous SET INTENSITY ON command. For example:

```
Two := BEGIN_STRUCTURE
       a := VIEWPORT
              HORIZONTAL=-1:1
              VERTICAL=-1:1
              INTENSITY=.5:1;
       b := SET INTENSITY ON 0:1;
       c := SET INTENSITY OFF .8:1;
       INSTANCE OF object;
       END_STRUCTURE;

DISPLAY Two;
```

The intensity range in effect is 0:1 since that is the range specified in the last SET INTENSITY command to be ON in the series. You can set the intensity range to .8:1 by SENDing a TRUE to <1>Two.c.

Other operations and definitions can affect intensity. The VECTOR_LIST command lets you separately specify the intensity of each vector in the list. If this is done, those vector intensities are affected by the intensity range of the VIEWPORT. If the object has very bright vectors in the background and dim vectors in the foreground, the effect of depth cueing could bring them to a nearly equal intensity by brightening the near, dim vectors and dimming the far, bright vectors. (Refer to the Section *RM1 Command Summary* for information on assigning intensities to vectors using the VECTOR_LIST command.)

5.2 Setting Color

It is possible to display entire objects (vector lists or character strings) in the same color. Color is specified in terms of hue and saturation. The hue is a color, such as red or blue. The saturation is the amount of color versus the amount of white in the hue. Red at high saturation is full-toned; red at low saturation is pink. All hues are white at 0 saturation.

The intensity, or brightness, of any hue/saturation combination depends on factors other than the color specification. These factors include such things as the intensity range of the viewport, and the condition of a SET INTENSITY command.

The PS 390 lets you choose from 120 hues. Selectable hues correspond to the values on the color wheel shown in Figure 8-40, with blue at 0 and 360, red at 120, and green at 240.

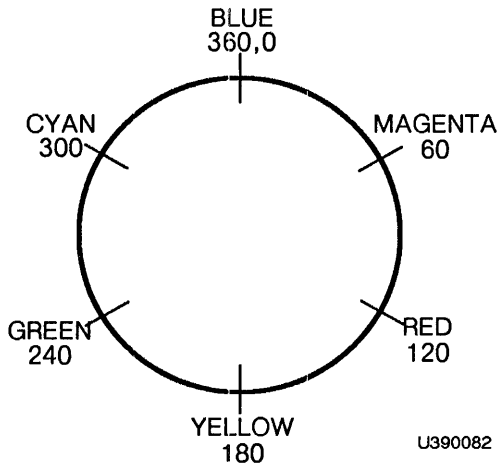


Figure 8-40. Color Wheel

In effect, then, color is specifiable in 3-degree increments around the color wheel. Hue values from 0 to 2 select the same hue; hue values from 3 to 5 select the same hue, etc.

The saturation of any hue is specified as a value from 1 to 0, or from full-color saturation to no color (white). The default saturation is full (1).

Color and saturation is set as follows:

```
Blue_Car := SET COLOR 0,1 APPLIED TO Car;
```

where 0 indicates the color (blue) and 1 is the saturation (full).

5.2.1 Exercise

Make the Car red, fully saturated.

```
Redcar := SET COLOR 120,1  APPLIED TO Car;

DISPLAY Redcar;
```

Change the color settings and watch what happens to the color of the car:

Same hue, less saturated:

```
Redcar := SET COLOR 120,.3  APPLIED TO Car;
```

The car appears to be light pink. For a new hue, full saturation enter:

```
Redcar := SET COLOR 240,1  APPLIED TO Car;
```

New hue midway between red and green—yellow, full saturation:

```
Redcar := SET COLOR 180,1  APPLIED TO Car;
```

Make the wheels of the yellow Car a different color than the car body by specifying a new color (green) for the tires only.

```
PREFIX Tire WITH SET COLOR 240,1;

INITIALIZE DISPLAY;
```

6. Viewing Summary

Viewing consists of placing an object in front of you by defining a line of sight (LOOK), defining a window (WINDOW, FIELD_OF_VIEW, EYE BACK), and setting up a portion of the PS 390 screen to display the window in (VIEWPORT).

If an object is viewed without specifying a line of sight, a window, or a viewport, defaults are supplied by the system. The default view has a line of sight from the origin (0,0,0) looking straight along the positive Z axis. In the default window, objects appear as orthographic views. The default viewport is the dynamic full screen.

The WINDOW command creates orthographic views. The FIELD_OF_VIEW and EYE BACK commands create perspective views. With FIELD_OF_VIEW, the line of sight is perpendicular to the front and back boundaries of the frustum of vision. With EYE BACK, the line of sight can be offset, creating a skewed frustum of vision.

Non-matrix viewing attributes may be used to set intensity in the dynamic viewport, to display entire objects in color in the dynamic viewport, and to enable and disable the display of objects on selected screens.

The following sections summarize concepts in this section.

Important Concepts for LOOK

- The LOOK transformation defines a line of sight in the world coordinate system in terms of a point to look from and a direction in which to look.
- If no LOOK is specified, the system defaults to a LOOK from 0,0,0 along the positive Z axis (AT 0,0,1).
- An UP direction can be specified as part of any LOOK transformation.
- If the line of sight coincides with the UP direction, the system defines positive Y relative to the LOOK AT point to be up in the new view.

- The command format to specify a LOOK is:

```
name := LOOK AT X,Y,Z FROM X,Y,Z   APPLIED TO Name2;
```

or

```
name := LOOK FROM X,Y,Z AT X,Y,Z   APPLIED TO Name2;
```

- The LOOK transformation is done in a 4x3 matrix. To work correctly, a LOOK transformation should be placed above all modeling transformations (ROTATE, TRANSLATE, SCALE) in the display structure and immediately below the windowing transformation (WINDOW, FIELD_OF_VIEW, EYE BACK).

Important Concepts for WINDOW

- Orthographic windows are specified in terms of X and Y and optionally Z.
- WINDOWs can be defined to be not centered around the X/Y axis.
- WINDOWs can be specified to be larger or smaller than the default window. Large windows encompass more, and therefore make objects appear smaller than they appear in smaller windows.
- Objects or parts of objects within a window are displayed when the window is displayed.
- Objects or parts of objects outside a window are clipped from view.
- Depth clipping at Z boundaries is not in effect unless you put it into effect.
- Depth cueing, the variation of intensity that imparts an illusion of depth to displayed objects, is anchored at the front and rear (Z) boundaries of the window. Brightest intensity occurs at the front boundary and dimmest occurs at the back boundary.
- WINDOWs are usually square.

- The command format to specify a WINDOW is:

```
name := WINDOW X=xmin:xmax  Y=ymin:ymax [FRONT boundary = zmin
        BACK boundary = zmax] APPLIED TO name1;
```

- The WINDOW transformation is done in a 4x4 matrix. To work properly, the WINDOW transformation must be the topmost matrix node in a display structure.

Important Concepts for FIELD_OF_VIEW

- FIELD_OF_VIEW is specified in terms of a viewing angle and front and back boundaries.
- The FIELD_OF_VIEW is always centered about the positive Z axis. The apex of the pyramid (your eyepoint) is always at 0,0,0.
- Since the eyepoint is always at 0,0,0, objects must be located on the positive Z axis, far enough out to be within the frustum of vision if they are to be seen. Usually a LOOK transformation is used to do this.
- The size of the viewing angle in no way distorts the perspective imposed on viewed objects. However, the larger the viewing angle, the larger the area included in the frustum of vision. Larger angles have the effect of making a viewed object appear smaller.
- Depth clipping is not in effect unless you put in effect with a SET DEPTH CLIPPING ON command.
- Depth cueing is anchored at the front and back boundaries. Brightest intensity occurs at the front boundary and dimmest occurs at the back boundary.
- The face of a window created using FIELD_OF_VIEW is always square. That is, it has an aspect ratio of 1.
- The command format to specify a FIELD_OF_VIEW is:

```
name := FIELD_OF_VIEW angle [FRONT boundary = zmin]
        [BACK boundary = zmax] APPLIED to name1;
```

- The FIELD_OF_VIEW transformation is performed by a 4x4 matrix. The FIELD_OF_VIEW operation node must be the topmost matrix node and be directly above the LOOK node in the display structure.

Important Concepts About the EYE BACK Command

- EYE BACK is specified in relative room coordinates to position the eye relative to the center of the viewport. Front and back boundaries are specified in world coordinates.
- The face of a window created using EYE BACK is always square.
- With the EYE BACK transformation, the line of sight is not necessarily collinear with the from/at line in LOOK.
- If the eye position is not collinear with the from/at line in LOOK, the viewing pyramid is skewed. Front and back boundaries remain perpendicular to the line of sight established in the LOOK specification.
- The larger the viewing angle, the larger the area included in the frustum of vision. Larger angles have the effect of making a viewed object appear smaller.
- The command format to specify EYE BACK is:

```
name := EYE BACK Z [option 1] [option 2] from SCREEN area w
        WIDE [FRONT boundary = zmin] [BACK boundary = zmax]
        APPLIED TO name1;
```

- The EYE BACK transformation is performed by a 4x4 matrix. To work properly, the EYE BACK operation node must be above all other transformation nodes and directly above the LOOK operation node in the display structure.

Important Concepts About Viewports

- A viewport is the area of the PS 390 screen to which a window is mapped.
- A viewport may be defined in terms of a current viewport by using the VIEWPORT command, or in terms of the full PS 390 screen by using the LOAD_VIEWPORT command.
- The dimensions of any current viewport are –1 to 1 in X and in Y.
- Multiple viewports can be displayed simultaneously.
- Nonsquare dynamic viewports distort displayed objects unless the viewed window has the same aspect ratio as the nonsquare viewport.

- An intensity range for a window (WINDOW, EYE BACK, etc) can be specified for a dynamic viewport.
- The command format to specify a dynamic viewport is either:

```
name := VIEWport HORizontal hmin:hmax VERtical = vmin:vmax
        [INTENsity = imin:imax] APPLIED TO name1;
```

or

```
name := LOAD_VIEWport HORizontal hmin:hmax
                      VERtical = vmin:vmax
        [INTENsity = imin:imax] APPLIED TO name1;
```

- Static viewports are always square.
- The command format to specify a static viewport is:

```
Send V3D (x,y,z) to <3> SHADINGENVIRONMENT;
```

- Mapping a window to a viewport is not a matrix operation, so viewport specifications can be placed anywhere in relation to matrix operations in a display structure.

Important Concepts About Viewing Attributes

- Viewing attributes differ from viewing transformations in that they are non-matrix operations. They can be placed above windows (WINDOW, FIELD_OF_VIEW, EYE BACK) and LOOK transformations in a display structure.
- The SET INTENSITY attribute manipulates viewport intensity. SET INTENSITY can be switched on and off, varying intensities between values in the viewport specification and values in the SET INTENSITY command.
- In a series of SET INTENSITY commands, the last one ON determines the intensity range in effect.
- A SET INTENSITY OFF command does not cancel a previous SET INTENSITY ON command.
- The SET COLOR attribute allows you to display entire objects as a single color. Color is specified in terms of hue and saturation. Hue is specifiable in 3-degree increments around a color wheel. Saturation is specified as a value from 1 to 0.

GT9. CONDITIONAL REFERENCING

GT9. CONDITIONAL REFERENCING

SELECTING PORTIONS OF A MODEL FOR DISPLAY

CONTENTS

ILLUSTRATIONS

Section GT9

Conditional Referencing

Selecting Portions Of A Model For Display

Introduction

Conditional referencing is the referencing of data only when certain conditions are met. It is a way to display selected branches of a display structure without displaying other branches. It is useful, for example, if you have a model that you would like to add parts to or take parts from, showing various stages of development or assembly.

There may be layers of detail in your model that you would like to be able to overlay or strip off. An example of adding detail might start with an outline map of the United States, then sequentially add major rivers, mountain ranges, state borders, major cities, county borders, etc.

You might also want to display different views of an object at different times to animate an object, or alternately display and blank an object at a selectable rate (blinking).

These kinds of operations are achieved with conditional referencing, using three methods: conditional-bit settings, level-of-detail settings, and rate settings.

To use conditional referencing, a minimum of two nodes must be placed in a display structure. The first node (called a SET node) sets a condition. A hypothetical PS 390 command to do this might be:

```
THE CONDITION IS 1
```

The second node (called an IF node) tests the condition and makes the traversal of the branch (and therefore the display of data indicated by that branch) dependent on the condition in the SET node:

```
IF THE CONDITION IS 1 THEN DISPLAY Object1
IF THE CONDITION IS 2 THEN DISPLAY Object2
```

Figure 9-1 shows these nodes in a display structure. These nodes are attribute nodes and follow the same rules of placement and of use as operation nodes.

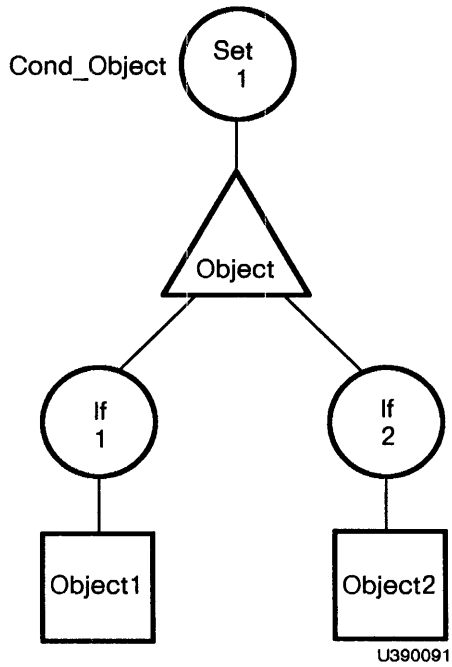


Figure 9-1. Display Structure Including Conditional Referencing Nodes

In the above example, displaying the SET node (Cond_Object) will result in Object1 being displayed and Object2 not being displayed. This is because the condition is not satisfied for the branch with Object2. By changing the condition from 1 to 2 in the SET node, Object2 will be displayed and Object1 will not be displayed.

The values in both the SET node (Cond_Object) and the IF nodes (Object1, Object2) can be changed interactively. For example, the two branches could be alternately displayed by toggling the numbers in the SET node between 1 and 2.

Objectives

In this section, you will learn to display selected parts of your display structure using:

- Conditional-bit attribute settings
- Level-of-detail attribute settings
- Rate attribute settings

Prerequisites

Before reading this section, you should be familiar with the rules for using operation nodes in display structures (Section *GT4 Modeling*), and the differences between matrix operations and attribute operations (Section *GT2 Graphics Principles.*)

1. Using Conditional-Bit Attribute Settings

Conditional bits are used to display selected branches of a display structure, independent of whether other branches are displayed. Branches of a display structure that have IF nodes that are not satisfied by the condition are not traversed by the display processor and are therefore excluded from displayed data.

The SET CONDITIONAL_BIT node is used to set any of 15 conditional bits (0–14). By placing the SET CONDITIONAL_BIT node above an instance node, bit settings affect all branches under the instance node. The SET node is created with the SET CONDITIONAL_BIT command. The syntax is as follows:

```
name := SET CONDITIONAL_BIT n switch APPLIED TO name1;
```

where:

n is an integer from 0 to 14, corresponding to the conditional bit to be set ON or OFF.

switch is either ON or OFF. All bits default to OFF.

name1 is the descendent node of the conditional bit node.

For example, the following command creates a SET node and sets BIT 2 ON APPLIED TO Car.

```
Pattern := SET CONDITIONAL_BIT 2 ON APPLIED TO Car;
Car := INSTANCE OF Body, Wheels;
```

When you create a SET node, you explicitly set one bit ON or OFF. However, all 14 bits default to OFF. So if you enter the command:

```
name := SET CONDITIONAL_BIT 1 ON APPLIED TO name1;
```

then bit 1 is ON, and bits 2–14 are OFF. All bits can be changed by sending values to an input of the SET node.

Inputs to the SET CONDITIONAL_BIT node are as follows:

```
Boolean-----> <1>  Sets the original bit (n)
                   to be ON (T) or OFF (F).

Integer-----> <2>  Sets bit number input (0-14)
                   ON.

Integer-----> <3>  Sets bit number input (0-14)
                   OFF.

Integer-----> <4>  Disables bit number input
                   (0-14) from being affected
                   by this node.

Integer-----> <5>  Toggles bit number input (0-14).
```

The SET node controls the states of the conditional bits and it is only through the SET node that the conditions of all 15 bits are changed. If bit 5 was originally set to ON and then you want to set it to OFF, it could be done in any of the following three ways:

- Sending the integer 5 to input<3> of the SET node.
- Sending a FALSE to input<1> of the SET node.
- Sending the integer 5 to input<5> of the SET node.

Of course, the SET node is useless unless you have an IF node that tests the condition set by the SET node. The IF node tells under which condition a branch will be traversed for display.

IF nodes are created with the IF CONDITIONAL_BIT command. The syntax is as follows:

```
name := IF CONDITIONAL_BIT n switch APPLIED TO name1;
```

where:

n is an integer from 0 to 14, indicating which bit to test.

switch is the setting to be tested, ON or OFF.

name1 is the descendent of the IF node.

The IF CONDITIONAL_BIT node has one input that accepts an integer (0–14) to change the bit number in the node.

In the following command sequence, when Car is displayed, Wheels would also be displayed.

```
Set := SET CONDITIONAL_BIT 4 ON APPLIED TO Car;
PREFIX Wheels WITH IF BIT 4 IS ON;
```

If bit 4 of Car is set to OFF or the condition in Wheels is changed to OFF, then the test in Wheels would fail and Wheels would not be displayed.

The display structure for Car that this command sequence creates in shown in Figure 9-2.

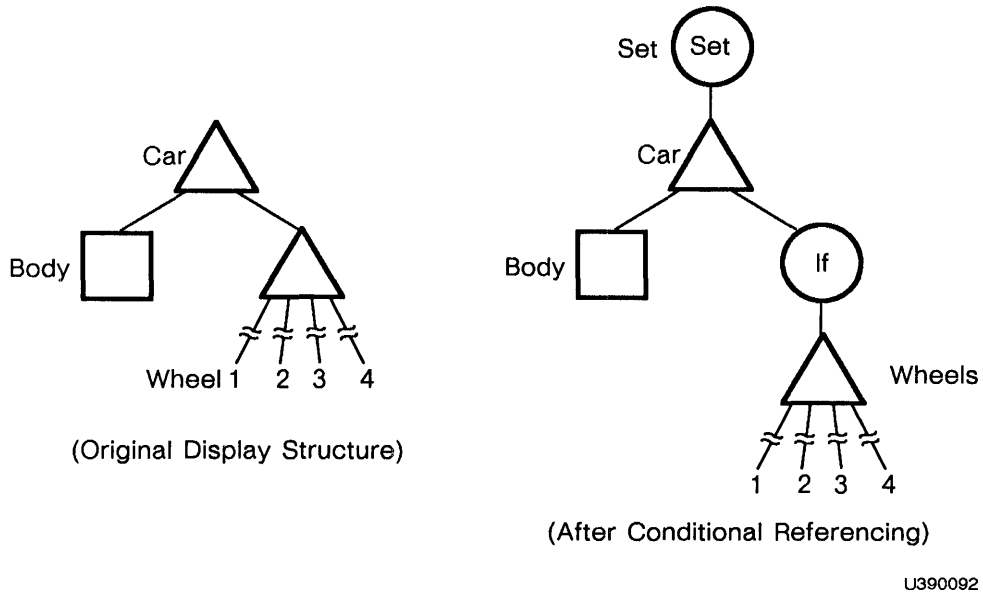


Figure 9-2. Car Display Structures

Figure 9-3 is a display structure for a molecule for which conditional referencing will be implemented.

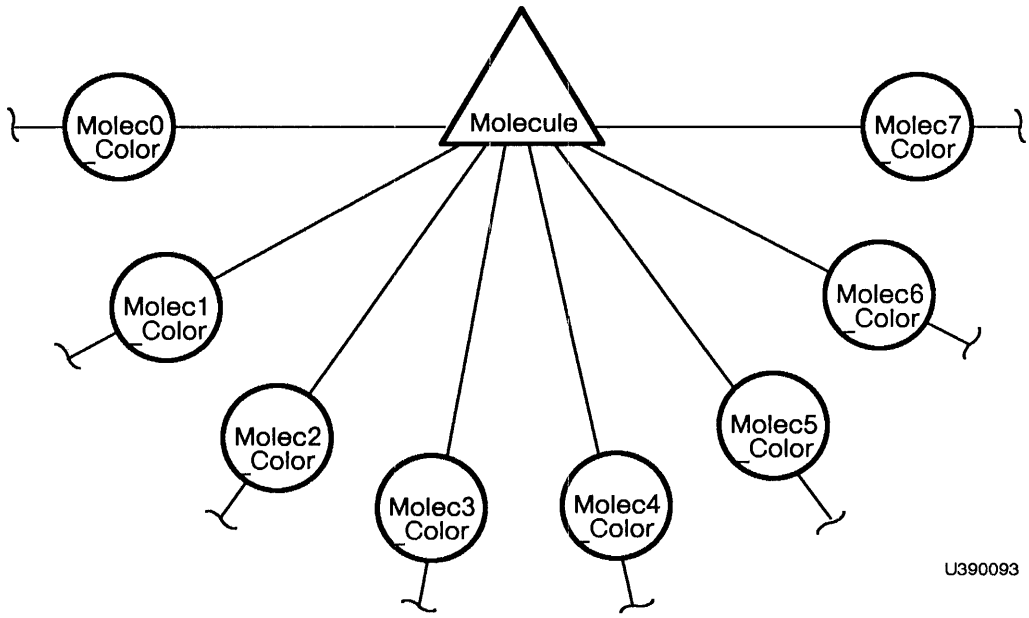


Figure 9-3. Molecule Display Structure

In Figure 9-3 notice that Molecule is made up of an instance node pointing to 8 SET COLOR nodes for parts of the molecule. The eight parts can be controlled separately for display by placing a SET node and eight IF nodes in the structure.

The molecule will be set with the following conditions.

Bit No.	Condition	Result
1	OFF	Branch 1 (Molec0_Color) will be displayed
2	OFF	Branch 2 (Molec1_Color) will be displayed
3	OFF	Branch 3 (Molec2_Color) will be displayed
4	OFF	Branch 4 (Molec3_Color) will be displayed
5	OFF	Branch 5 (Molec4_Color) will be displayed
6	OFF	Branch 6 (Molec5_Color) will be displayed
7	OFF	Branch 7 (Molec6_Color) will be displayed
8	OFF	Branch 8 (Molec7_Color) will be displayed

The display structure to implement this is shown in Figure 9-4.

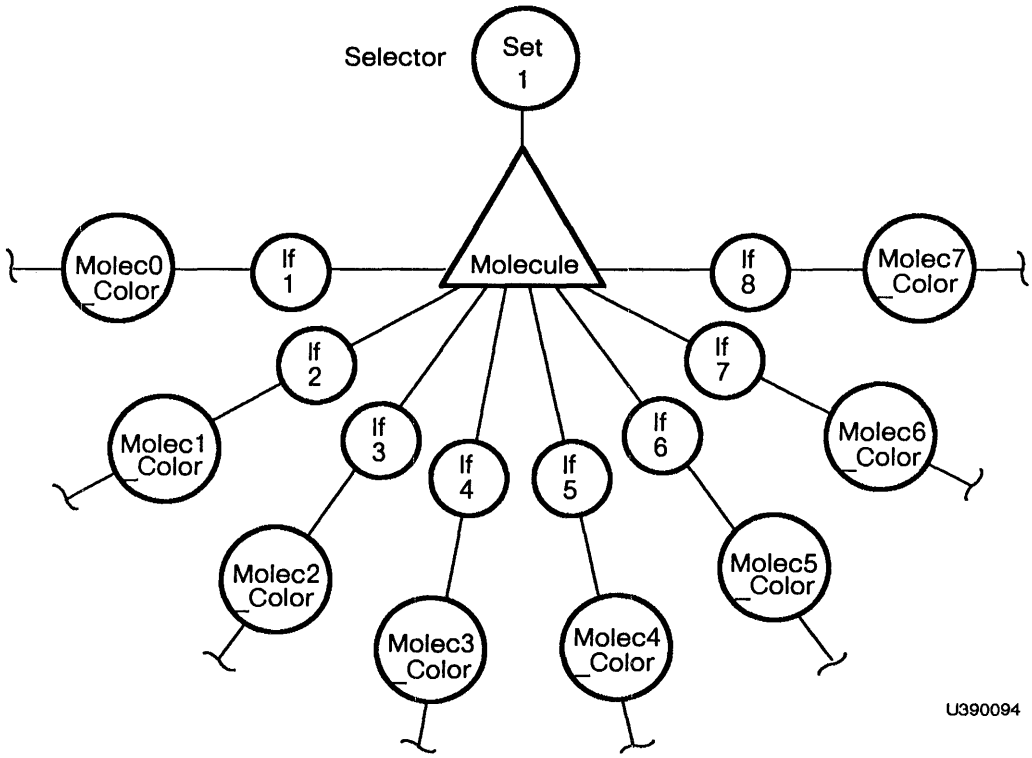


Figure 9-4. Display Structure for Conditional Referencing in Molecule

1.1 Exercise

Add conditional-bit referencing to the display structure for Molecule. The first step is to place a SET node above the instance node Molecule. Do this by entering:

```
Selector := SET CONDITIONAL_BIT 1 OFF APPLIED TO Molecule;
```

Remember, even though the command says to set only conditional bit 1 OFF, this one node may be used to separately control the ON/OFF condition of all 15 conditional bits. Also, note that the condition of the other 14 bits defaults to OFF.

Next place nodes at the top of each branch under the instance node so that the branches will be separately selectable for display. To do this, redefine Molecule as follows.

```
Molecule := BEGIN_STRUCTURE

                    IF BIT 1 IS OFF THEN Molec0_Color;
                    IF BIT 2 IS OFF THEN Molec1_Color;
                    IF BIT 3 IS OFF THEN Molec2_Color;
                    IF BIT 4 IS OFF THEN Molec3_Color;
                    IF BIT 5 IS OFF THEN Molec4_Color;
                    IF BIT 6 IS OFF THEN Molec5_Color;
                    IF BIT 7 IS OFF THEN Molec6_Color;
                    IF BIT 8 IS OFF THEN Molec7_Color;

                END_STRUCTURE;
```

You have built the display structure that allows conditional-bit referencing in Molecule. Notice that the molecule is displayed because all conditional bits are set OFF. To remove parts of the molecule from display, bits must be set ON.

To control the ON/OFF condition of the eight bits that affect the branches of this display structure, a function network can be used to connect the function keys to the SET node named Selector. That network is shown in Figure 9-5.

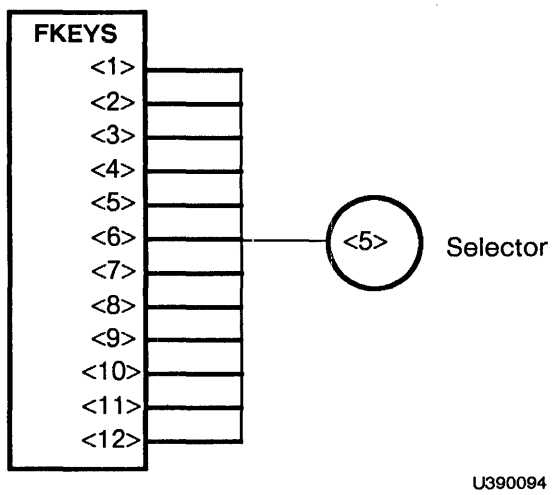


Figure 9-5. Function Network for Conditional-Bit Control

FKEYS will output integers corresponding to the number of the pressed function key. Input<5> to the SET CONDITIONAL_BIT node toggles the setting of the bit corresponding to the integer received. For example, if bit 6 is OFF, pressing function key F6 will turn bit 6 ON.

Enter the following commands to build the network.

```
CONNECT FKEYS<1>:<5>Selector;
```

The display structure is now designed to allow conditional display of parts of the molecule (Molec0_Color through Molec7_Color). Also, the function keys have been connected to control this display.

One step remains in this particular case. The values used to define the molecule are large. The molecule has a diameter of some 45,000 units. To see the molecule, put a window around it and disable depth cueing by entering:

```
Molecule_View := WINDOW
                 X=-22500:22500
                 Y=-22500:22500
                 FRONT BOUNDARY = -22500
                 BACK BOUNDARY = 22500 APPLIED TO Intensity;
Intensity := SET INTENSITY ON 1:1 APPLIED TO Selector;
```

now,

```
DISPLAY Molecule_View;
```

Press SHIFT/LINE LOCAL (PS 300-style keyboard) or press CTRL/LOCAL (PS 390-style keyboard) to enable the function keys. Use keys F1 through F8 to toggle the display of the parts of the molecule.

When you are finished enter:

```
REMOVE Molecule_View;
```

2. Using Level-of-Detail Conditional Referencing

The conditional-bit method shown for the molecule is usually used when you need to separately control the display of branches of your display structure in a variety of sequences. In the level-of-detail method, the parts of a model are always displayed and removed in a predetermined sequence.

Level-of-detail is usually used to overlay detail on your picture. For example, progressive detail could be added to an outline of a sphere (world) to add continents, mountain ranges, states, etc. Level-of-detail can also be used to run animation sequences comprised of a series of separate picture definitions.

Unlike conditional-bit referencing where 15 variables (bits) are set, only one variable is set using the level-of-detail method. All IF nodes are tested against that one variable in the SET node.

The command to create a SET LEVEL_OF_DETAIL node is as follows.

```
name := SET LEVEL_OF_DETAIL TO n APPLIED TO name1;
```

where:

n is an integer from 0 to 32767 indicating the level of detail value. (The default **n** is 0.)

name1 is the descendent of the SET node.

The input for updating the SET LEVEL_OF_DETAIL node is:

```
Integer-------->   <1> Changes the level of detail
                       (0-32767) to the value of the
                       received integer.
```

2.1 Determining the Order for Overlaying Detail

Because level-of-detail controls the display of branches in a determined order, the conditional statements are expressed as relationships rather than the two-state (ON/OFF) type used in conditional-bit references.

These relationships are:

```
Less Than                 <
Less Than Or Equal To     <=
Equal To                  =
Not Equal To              <>
Greater Than Or Equal To  >=
Greater Than              >
```

and are specified in the IF LEVEL_OF_DETAIL node. The command to create this IF node is as follows.

```
name := IF LEVEL_OF_DETAIL relationship n THEN name1;
```

where:

relationship is the relationship to n to be tested (<, <=, =, <>, >=, >).

n is an integer from 0 to 32767 indicating the number (along with the previous relationship) to compare against the current level of detail setting (the default **n** is 0).

name1 is the descendent of the IF LEVEL_OF_DETAIL node.

The IF LEVEL_OF_DETAIL node has one input that accepts an integer (0–32767) to change the value in the node.

With the following command sequence,

```
A := SET LEVEL_OF_DETAIL TO 3 THEN B;
B := IF LEVEL_OF_DETAIL = 3 THEN C;
C := VECTOR_LIST .....;
```

initially when A is displayed, C is also displayed. If the level-of-detail is changed to something other than 3, then the test in B fails and C is not displayed.

An example of adding detail is to start with a sphere and add continents, mountain ranges, and countries. To display the parts of the world in this order (and turn them OFF in the reverse order):

Sphere
Continents
Mountain Ranges
Countries

the sphere needs to be displayed first and remain on while all subsequent parts are displayed.

The continents need to be added next, the mountain ranges and then the countries. If the sphere is displayed whenever there is a value of 1 or greater in the SET node, and the subsequent parts are displayed for values equal or greater than 2, 3, and 4, respectively, the desired effect is achieved.

The display structure that sets up such a level-of-detail condition is shown in Figure 9-6.

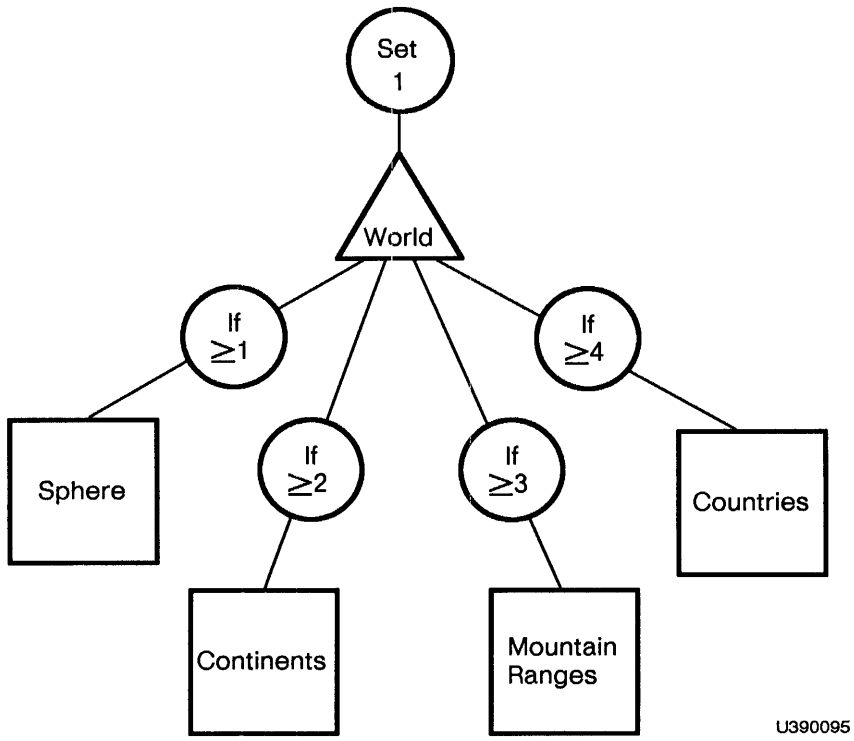


Figure 9-6. Level-of-Detail Structure for the World

By changing the value of the integer in the SET node, the parts of the sphere can be laid on and stripped off. If the integer 2 is sent to the SET node, then the sphere and the continents are both displayed because both branches of the display structure meet the condition tested against the SET node. If the integer 3 is sent to the SET node, the sphere, the continents, and the mountain ranges are all displayed. If the integer 4 is sent to the SET node, the entire structure is displayed. The details of the sphere can be stripped off by decreasing the value in the SET node.

2.2 Using Level-of-Detail Settings to Animate An Object

An example of using level-of-detail settings for animation is in the turbine blade portion of the PS 390 Demonstration Package. The turbine blade is defined as a sequence of turbine blades in slightly different positions. A clock is used to advance the level of detail settings resulting in the display sequence and the apparent motion of the turbine blade. The structure that sets this up is similar to the one shown in Figure 9-7.

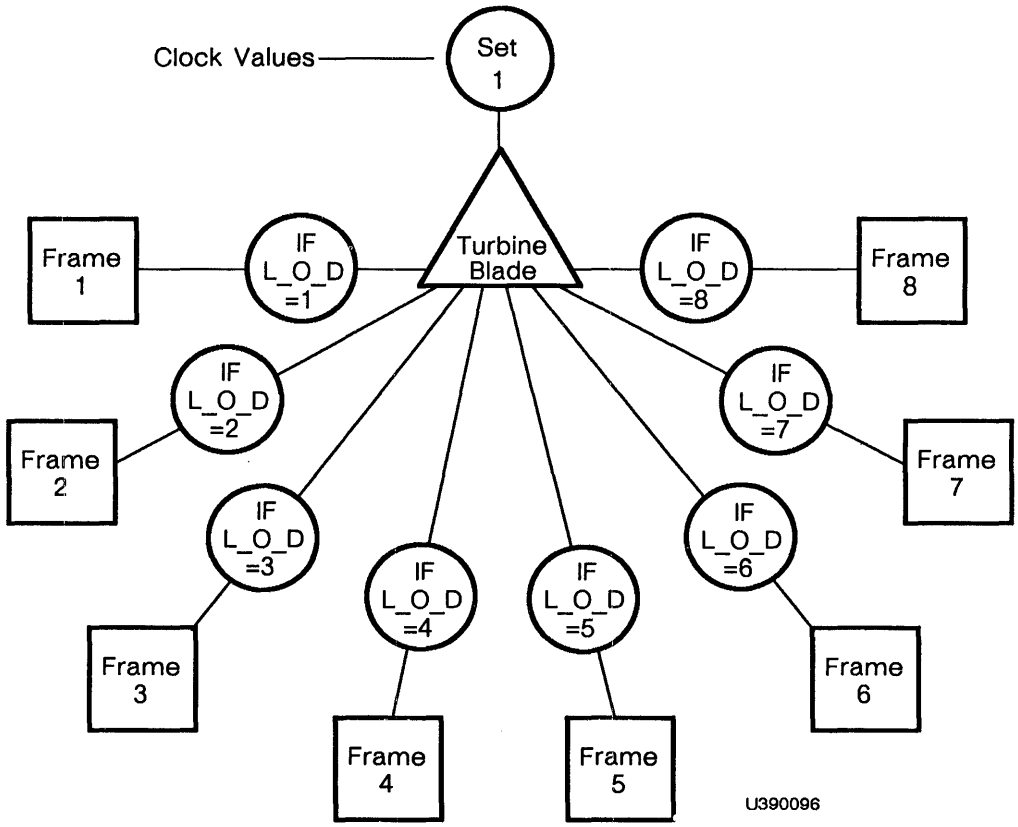


Figure 9-7. Turbine Blade Structure

The topmost node is the one supplied with clock values through a function network to step through the sequence of pictures corresponding to the referenced branches in the display structure. Note that in animation, detail is not laid over a displayed picture. Instead, sequences of pictures are displayed.

2.3 Exercise

Using the Tutorial Demonstrations select the Level-of-Detail Program from the menu.

The ANIMATED_CYLINDER in this demonstration is a good example of how level-of-detail settings can be used for local animation.

3. Using Rate Attribute Settings

The third type of conditional referencing allows you to blink an object or display structure branch under control of the refresh rate of the PS 390 display, an internal PS 390 clock, or an external clock. This type of conditional referencing can cause an object to blink or to be displayed alternately with another object. (For example, one part might be displayed for one second, then that part is removed while another part is displayed for a second, etc.)

Like the other types of conditional referencing, blinking requires two nodes. One node sets a blink rate in terms of phase ON and OFF durations. The other if node tells tell whether an object or branch will be displayed during the ON phase or the OFF phase.

3.1 Creating the SET RATE Node

The command to create the SET RATE node is:

```
name :=  SET RATE phase_ON phase_OFF [initial_state] [delay]
         APPLIED TO name1;
```

where:

phase_ON phase_OFF are integers designating the durations of the ON and OFF phases, respectively, in refresh frames.

initial_state is either ON or OFF, indicating the initial phase (the default initial_state is OFF).

delay is an integer designating the number of refresh frames in the initial state.

name1 is the descendent of the SET RATE node.

Inputs for updating the SET RATE node are as follows:

```
INTEGER----> <1>   Changes the phase_on value.

INTEGER----> <2>   Changes the phase_off value.

BOOLEAN----> <3>   Changes the initial_state ON(T)/OFF(F).

INTEGER----> <4>   Changes the delay.
```

A command similar to SET RATE, SET RATE EXTERNAL allows you to alter the PHASE attribute via an external source such as a function network or a message from the host computer. Refer to Section *RM1* for specific details of this command.

3.2 Creating the IF PHASE Node

The command to create the IF node to test the ON/OFF state of the phase is as follows:

```
name := IF PHASE IS state THEN name1;
```

where:

state is the phase setting under which name1 is displayed (ON or OFF).

name1 is the descendent of the IF PHASE node.

If there is no SET RATE node or SET RATE EXTERNAL node higher in the structure, the "state" of the PHASE node will always be OFF.

For example, with the command sequence

```
Shape := SET RATE  10  15 APPLIED TO Blink_Shape;
Blink_Shape := IF PHASE ON THEN Sphere;
Sphere := VECTOR_LIST ....;
```

If Shape is displayed, Sphere will be displayed for 10 refresh frames and not displayed for 15 refresh frames repeatedly.

If the command sequence is

```
Shape := SET RATE  10  15 APPLIED TO Blink_Shape;
Blink_Shape := IF PHASE OFF THEN Sphere;
Sphere := VECTOR_LIST .... ;
```

If Shape is displayed, Sphere will be displayed for 15 refresh frames and not displayed for 10 refresh frames repeatedly, since the condition is to display the vector list when the phase is OFF.

3.3 Exercise

This exercise uses the robot created in Section *GT5*.

To demonstrate the effects of blinking, add blinking nodes above Robot. The blink rate in this exercise will be based on the PS 390 refresh rate. First, define a node that sets the rate by entering:

```
Blink_Robot := SET RATE 120  60 APPLIED TO If_Robot;
```

This sets the ON phase to 120 refreshes and the OFF phase to 60 refreshes.

Now place a node that determines whether the robot will be displayed in the ON phase (and blanked in the OFF phase) or displayed in the OFF phase (and blanked in the ON phase). Display robot in the ON phase, by entering:

```
If_Robot := IF PHASE IS ON THEN Robot;
```

Robot should now blink at a rate of about 2 seconds on and one second OFF, when you:

```
DISPLAY Blink_Robot;
```

Then:

```
REMOVE Blink_Robot;
```

3.4 Some Uses for Timed Blinking

One practical use of the rate setting commands, other than the visual effects produced, is that they can synchronize the refresh rate of the display to a movie camera to make sure that the frame rate of the camera matches the frame refresh rate of the screen, allowing the camera to always be taking a frame as the picture is refreshed.

Stereo views can be created using a split screen (two viewports side by side), each half containing the same image and viewed with the EYE BACK projection (refer to Section *GT8 Viewing Operations*). Then each viewport can be displayed alternately with the other viewport. By placing an opaque divider between the viewports so each eye can see only one viewport, a 3D effect can be generated.

4. Summary

The flexibility and ease of use of conditional referencing within the display structure makes what is often a difficult operation on other graphics machines easy on the PS 390.

Conditional referencing allows you to display selected branches of a display structure without displaying other branches. These kinds of operations are achieved using three methods: conditional-bit settings, level-of-detail settings, and rate settings.

To use conditional referencing, a minimum of two nodes must be placed in a display structure. The first node sets up the condition on which all subsequent references are tested. The second tests the condition and makes traversal of the branch (display of the data) dependent on the condition in the set node.

Using Conditional Bit Settings

The conditional-bit method shown is used when you need to separately control the display of branches of your display structure in a variety of sequences.

The SET CONDITIONAL_BIT node sets any of 15 conditional bits (0–14). By placing the set conditional bit node above an instance node, then bit settings affect all branches under the instance node.

This node is created with the SET CONDITIONAL_BIT command. The syntax is as follows.

```
name := SET CONDITIONAL_BIT n switch APPLIED TO name1;
```

where:

n is an integer from 0 to 14, corresponding to the conditional bit to be set ON or OFF.

switch is either ON or OFF (all bits default to OFF).

name1 is the descendent node of the conditional bit node.

IF nodes (to test the condition of the SET node) are created with the IF CONDITIONAL_BIT Command. The syntax is as follows:

```
name := IF CONDITIONAL_BIT n switch APPLIED TO name1;
```

where:

n is an integer from 0 to 14, indicating which bit to test.

switch is the setting to be tested, ON or OFF.

name1 is the descendent of the IF node.

Using Level-of-Detail Conditional Referencing

When using the level-of-detail method, the parts of the model are always displayed and removed in a set sequence. Level-of-detail is usually used to overlay detail on your picture.

Level-of-detail can also be used to run animation sequences comprised of a series of separate picture definitions.

Unlike conditional-bit referencing where 15 variables (bits) are set, only one variable is set using the level-of-detail method. All IF nodes are tested against that one variable in the SET node.

The command to create a SET LEVEL_OF_DETAIL node is as follows.

```
name := SET LEVEL_OF_DETAIL TO n APPLIED TO name1;
```

where:

n is an integer from 0 to 32767 indicating the level-of-detail value (the default **n** is 0).

name1 is the descendent of the SET node.

Determining Order for Overlaying Detail

Because level-of-detail controls the display of branches in a determined order, the conditional statements are expressed as relationships rather that the two-state (ON/OFF) type used in conditional-bit references.

These relationships are specified in the IF_LEVEL_OF_DETAIL node:

```
Less Than                 <
Less Than Or Equal To     <=
Equal To                  =
Not Equal To              <>
Greater Than Or Equal To  >=
Greater Than              >
```

The command to create this IF node is as follows.

```
name := IF LEVEL_OF_DETAIL relationship n THEN name1;
```

where:

relationship is the relationship to be tested (<, <=, =, <>, >=, >).

n is an integer from 0 to 32767 indicating the number (along with the previous relationship) to compare against the current level-of-detail setting (the default **n** is 0).

name1 is the descendent of the IF LEVEL_OF_DETAIL node.

Using Level-of-Detail Settings to Animate an Object

An example of using level-of-detail settings for animation is in the turbine blade portion of the PS 390 Demonstration Package. The turbine blade is defined as a sequence of turbine blades in slightly different positions. A clock is used to advance the level-of-detail settings resulting in the display sequence and the apparent motion of the turbine blade.

Blinking and Alternately Displaying Parts of an Object

The third type of conditional referencing, rate attribute settings, allows you to blink an object or display structure branch under control of the refresh rate of the PS 390 display, an internal PS 390 clock, or an external clock. This type of conditional referencing can cause an object to blink or to be displayed alternately with another object. (For example, one part might be displayed for one second, then that part is removed while another part is displayed for a second, etc.)

Like the other types of conditional referencing, blinking requires two nodes. One node sets a blink rate in terms of phase ON and OFF durations. The other IF node tells whether an object or branch will be displayed during the ON phase or the OFF phase.

Creating the SET RATE Node

The command to create the SET RATE node is:

```
name := SET RATE phase_ON phase_OFF [initial_state] [delay]
        APPLIED TO name1;
```

where:

phase_ON phase_OFF are integers designating the durations of the ON and OFF phases, respectively, in refresh frames.

initial_state is either ON or OFF, indicating the initial phase (the default **initial_state** is OFF).

delay is an integer designating the number of refresh frames in the initial state.

name1 is the descendent of the SET RATE node.

Similar to SET RATE, a command SET RATE EXTERNAL allows you to alter the PHASE attribute via an external source such as a function network or a message from the host computer. Refer to Section *RM1* for specific details of this command.

Creating the IF PHASE Node

The command to create the IF node to test the ON/OFF state of the phase is as follows:

```
name := IF PHASE IS state THEN name1;
```

where:

state is the phase setting to be tested (ON or OFF).

name1 is the descendent of the SET RATE node.

If there is no SET RATE node or SET RATE EXTERNAL node higher in the structure, the state of the PHASE node will always be OFF.

GT10 TEXT MODELING

GT10. TEXT MODELING AND STRING HANDLING

CONTENTS

ILLUSTRATIONS

Section GT10
Text Modeling and String Handling

Introduction

Text is handled by the PS 390 in the same way as any other graphical item. Characters are defined as data nodes consisting of a single string (a CHARACTERS node) or a block of several strings or labels (a LABELS node). Just like other graphical items, characters can be transformed through matrices. Because they are affected by 3x3 matrices, they can be transformed along with any three-dimensional object which includes them in its definition. Characters can also be rotated and scaled using commands that create 2x2 transformation matrices. These matrices transform text while leaving other 2D and 3D graphical data unaffected.

Strings can be created and manipulated with commands. They can also be manipulated interactively using function networks and interactive devices.

A standard character font comes with the PS 390. Commands exist which allow you to design and use an unlimited number of alternate character fonts. A graphical character font editor program, MAKEFONT, is also available for designing and modifying character fonts. Refer to Section *TT7 Character Font Editor,* for information about this program.

Text and text-handling nodes are included in display structures. Text strings are data nodes and text transformations are operation nodes. The current character font is an attribute node which points to a look-up table for the vectors which comprise the font in current use.

Objectives

In this section you will learn how to:

- Use commands to create character strings.
- Use commands to manipulate character strings.
- Use functions to manipulate characters and strings.
- Update characters and labels nodes.
- Create and use different character fonts.

Prerequisites

Be at a PS 390 and have access to PS 390 Tutorial Demonstration Programs. Be familiar with the concepts covered in Sections *GT2 Graphics Principles, GT4 Modeling,* and *GT5 Command Language.* Also have at hand *Reference Materials 1–4.*

1. Using Commands To Create Character Strings

Two PS 390 commands create character strings: the CHARACTERS command and the LABELS command.

1.1 The CHARACTERS Command

The CHARACTERS command lets you create a single string of up to 240 characters and specify the location of that string in the world coordinate system.

The simplest form of the command lets you create a string which starts at the origin (the default location). Put the PS 390 in command mode by pressing the CTRL/LINE_LOCAL (PS 300-style keyboard) or CTRL/CMND or ALT/CMND (PS 390-style keyboard) keys. Use the following command to assign the name *String* to a character string.

```
String := CHARACTERS 'The quality of mercy... ;
```

Now DISPLAY String. All you can see at the moment is a large **T** in the top-right quadrant and the vertical stroke of the **h**. This is because each character is defined in a square which, by default, is one unit on each side. The default starting point for any string is the origin. Since the default window is from –1 to 1 in X and Y, only the first letter is within the window. Figure 10-1 illustrates this.

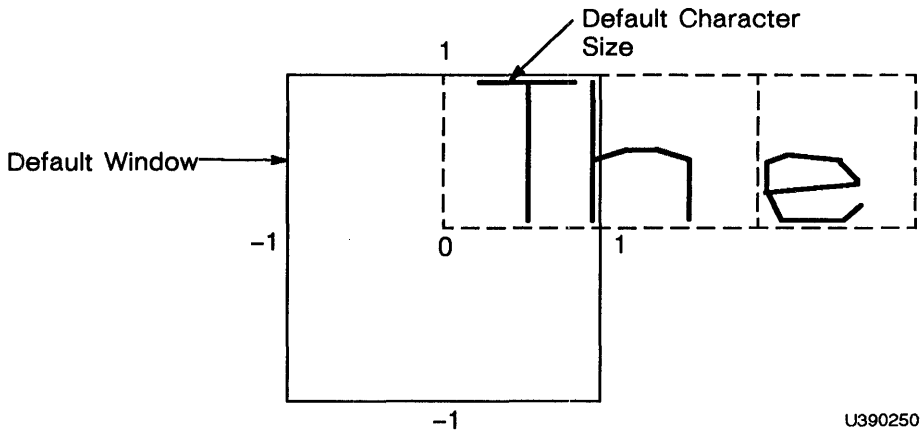


Figure 10-1. Default Window and Character Size

To scale the characters to fit the default window and display the string at its new size, enter the following commands.

```
Scale_String := SCALE BY .04 APPLIED TO String;
REMOVE String;
DISPLAY Scale_String;
```

The string should now appear in much smaller letters beginning at the center of the screen. Notice that the characters which form the string in the CHARACTERS command are enclosed in single quotation marks; however, when String is displayed, only the characters appear. If you want quotation marks in the text string, you must use three single quotation marks at the start and at the end of the string. Redefine String by entering the following command.

```
String := CHARACTERS '''The quality of mercy...''';
```

The character string should now appear in quotation marks. To get a single quote to appear in a string (as an apostrophe, for example) you must enter two single quotes. Redefine String with the following command.

```
String := CHARACTERS 'Love''s not time''s fool';
```

The string should appear with the contraction **Love's** and the possessive **time's**.

1.1.1 Changing Starting Position and Spacing

When the PS 390 displays a character string, the string is positioned by default with the lower-left corner of the unit square enclosing the first character at the origin of the world coordinate system. Characters are regularly spaced and follow each other horizontally. Optional parameters in the command let you specify the beginning coordinates of the string and change the horizontal and vertical spacing of the characters to create vertical and diagonal text strings. Enter the following command to redefine String as a new line of text positioned off the origin.

```
String := CHARACTERS 0,5,0 'Up a little';
```

This string starts at 0 on the X axis and 0 on the Z axis but 5 on the Y axis. The X,Y,Z coordinate of the starting point can always be specified in this way. The Z coordinate is optional and, if not supplied, defaults to zero.

The spacing between characters can be changed with a STEP clause. This clause lets you specify the spacing between characters in X and Y as a value from -1 to 1. The default spacing is 1,0 or one unit in X and zero in Y for regular horizontal spacing.

The vertical spacing can be changed by specifying the Y component of the STEP clause as a value other than zero. Enter the following command to create a string which descends diagonally from the origin to the right.

```
String := CHARACTERS STEP 1,-1 'Stepping down';
```

Now redefine the string as a diagonal which ascends from the origin to the upper-right.

```
String := CHARACTERS STEP 1,1 'Stepping up';
```

1.1.2 Exercise

Try different combinations of X and Y values to produce strings which descend and ascend vertically from the origin.

1.2 The LABELS Command

The LABELS command, like CHARACTERS, defines character strings for display. Whereas CHARACTERS defines a single string, LABELS combines any number of character strings into a single block. Each character string in the block is called a label. The command is quite straightforward to use. The following example combines some of the text strings created earlier in this section into a single label block.

```
String := LABELS  0,0   'The quality of mercy...'
                 -1,2   '''The quality of mercy...'''
                  4,5   'Up a little'
                  2,-5  'Love''s not time''s fool';
```

Diagonal and vertical strings could not be included in the block, however, because they specify different horizontal and vertical spacing between characters. The LABELS command is not able to accommodate this. The only clause in the command is the X,Y,Z coordinate of each label in the block.

1.3 When to Use CHARACTERS and LABELS

Both the CHARACTERS and the LABELS commands create data nodes in a display structure. Whenever several character strings are defined as a single LABELS node rather than as separate CHARACTERS nodes, there is a gain in display capacity. If you are displaying a lot of text, it is best defined using the LABELS command.

Character strings defined with the CHARACTERS command, however, are more versatile. In deciding which command to use, keep the following in mind.

- The CHARACTERS command lets you change the horizontal and vertical spacing between characters. The LABELS command does not.
- If text is created using CHARACTERS, you can manipulate any character in the text string. If the LABELS command is used, the smallest entity you can manipulate is a single text string.

2. Using Commands To Manipulate Character Strings

The CHARACTERS and LABELS commands create data nodes containing text. Like any other primitive data, text can be transformed by having a matrix applied to it. Text can be rotated and scaled using the ROTATE and SCALE commands which transform any two-dimensional or three-dimensional structure. In addition, characters can be transformed with their own rotate and scale commands: CHARACTER ROTATE, CHARACTER SCALE, and TEXT SIZE. These commands create 2x2 transformation matrices which only operate on text.

2.1 The CHARACTER ROTATE Command

The CHARACTER ROTATE command rotates a character string or label block around the Z axis. When you look in the positive direction of the axis, the rotation is counterclockwise.

To see the effect of this command, initialize the display, then rotate and display the scaled labels block.

```
INITIALIZE DISPLAY;
Rot_Text := CHARACTER ROTATE 90 APPLIED TO Scale_String;
DISPLAY Rot_Text;
```

Each string in the block should be rotated 90 degrees to the left. Notice that each label in the block is rotated around its own starting location. There is no single point in a labels block around which the whole block rotates.

A character rotate node can be updated interactively by any 2x2 matrix. The functions F:MATRIX2 and F:CROTATE (where C stands for character) are often used to supply the new matrix to the node.

2.2 The CHARACTER SCALE Command

Characters can be scaled like any other primitive data by a three-dimensional scale matrix using the SCALE command. There is also a CHARACTER SCALE command which creates a 2x2 scale matrix for transforming text only.

There are two forms of the CHARACTER SCALE command, one for uniform scaling and one for nonuniform scaling. Enter the following commands to initialize the display and to uniformly scale by .75 and then display the characters in the labels block.

```
INITIALIZE DISPLAY;
Char_Scale := CHARACTER SCALE .75 APPLIED TO Scale_String;
DISPLAY Char_Scale;
```

The scale factor is applied in both X and Y to the characters that compose Scale_String. A nonuniform scale can be applied by specifying separate scale factors in X and Y. Enter the following command to redefine Char_Scale and make tall characters.

```
Char_Scale := CHARACTER SCALE .5,3 APPLIED TO Scale_String;
```

Characters in the strings are made tall and thin with this command.

When several CHARACTER SCALE commands are used, each is concatenated with the next and a cumulative scaling matrix is applied to the characters. To see this effect, initialize the display and create and display a text string called Text.

```
INITIALIZE DISPLAY;
Text := CHARACTERS 'See Spot run.';
DISPLAY Text;
```

Since the characters are at the default size, only the capital **S** and one line of the first lowercase **e** are visible in the top-right quadrant of the screen. Now scale the string by prefixing it with a CHARACTER SCALE node.

```
PREFIX Text WITH CHARACTER SCALE .5;
```

The characters should now change to half their previous size, and the **S**, first **e**, and one line of the second **e** should be visible. The PREFIX command inserts a new node above the existing node and assigns the name of the existing node to the new node. Figure 10-2 shows the effect of the PREFIX command on the display structure.

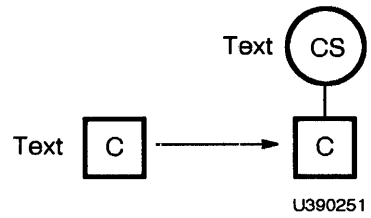


Figure 10-2. The Effect of the PREFIX Command

Use the PREFIX command again to create another scale node above the last one.

```
PREFIX Text WITH CHARACTER SCALE .1;
```

Notice that the size of the characters is now one tenth of what it was immediately before, not one tenth of the original default size. The actual size of the text is .5 times .1, which is .05 of the default size. The new display structure is as shown in Figure 10-3.

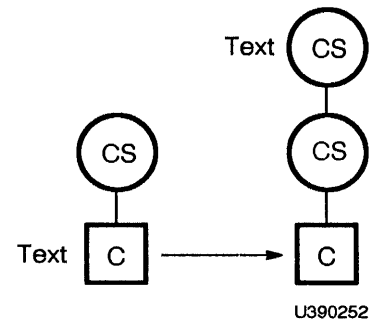


Figure 10-3. New Node Added with the PREFIX Command

The two character scales are concatenated and the combined scaling matrix is applied to the characters.

2.3 The TEXT SIZE Command

Character sizes can also be changed with the TEXT SIZE command. This command creates a text size which replaces the default size of 1. Text sizes are expressed as multiples or fractions of the default size.

Like the CHARACTER SCALE command, TEXT SIZE creates a 2x2 scaling matrix. However, this matrix is not concatenated with any other matrix. This means that the command creates a node which overrides any 2x2 matrix nodes above it in the same branch of the display structure.

To see the effect of the command, first remove the two CHARACTER SCALE prefixes of the string called Text, then prefix Text with a TEXT SIZE node.

```
REMOVE PREFIX OF Text;
REMOVE PREFIX OF Text;
PREFIX Text WITH TEXT SIZE .5;
```

As you remove the prefixes, the characters being displayed should get larger until they are back to the default size, and only the capital S is visible in the top-right quadrant. Prefixing with the TEXT SIZE command should make the letters half of the default size. The display structure for this structure is as shown in Figure 10-4.

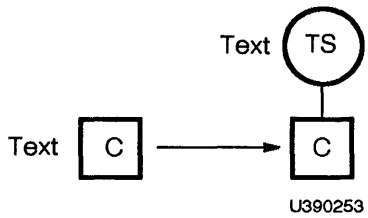


Figure 10-4. Display Structure with TEXT SIZE Node

Now prefix Text with a CHARACTER SCALE node to scale the characters by half again.

```
PREFIX Text WITH CHARACTER SCALE .5;
```

The text size does not change. This is because the effect of the CHARACTER SCALE node is overridden by the TEXT SIZE node below it in the structure. The display structure for the structure is shown in Figure 10-5.

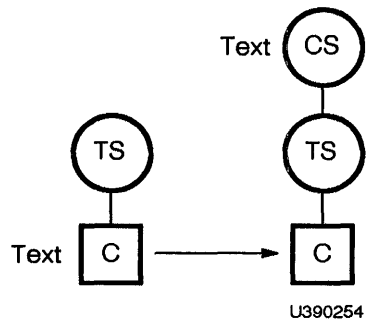


Figure 10-5. TEXT SIZE Node Prefixed with CHARACTER SCALE Node

Now prefix the CHARACTER SCALE node with a character rotation node.

```
PREFIX Text WITH CHARACTER ROTATE 90;
```

Again, nothing happens. The TEXT SIZE node overrides all 2x2 matrices above it. Since a CHARACTER ROTATE node is a 2x2 matrix node, it too is canceled out like the character scale. You should take this into account when structuring data.

2.3.1 Exercise

The TEXT SIZE node has no effect on 3x3 matrices, however. Try replacing the CHARACTER ROTATE node with a ROTATE node, and the rotation will be applied.

2.4 Character Orientation

If a transformation is applied to an object or part of an object which contains text in its structure, the default condition is that the text will be transformed too. Consider the display structure in Figure 10-6.

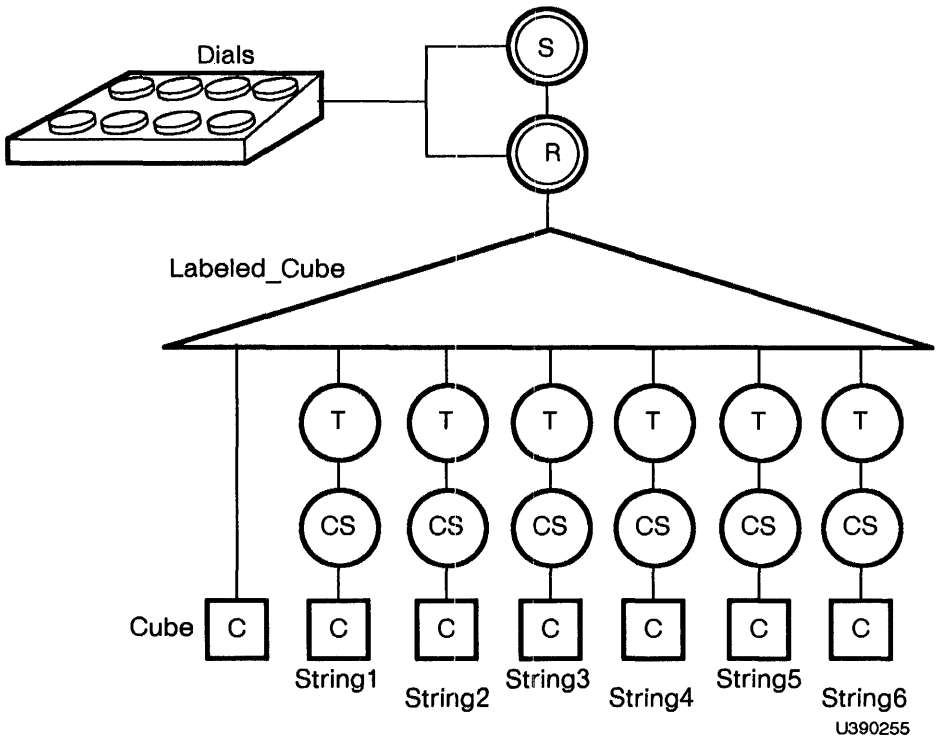


Figure 10-6. Display Structure for a Labeled Cube

An instance node called Labeled_Cube groups a vector list defining a cube and character strings which are scaled and positioned on each face to label the front, back, top, bottom, left, and right. A rotation node connected to three dials through a function network allows Labeled_Cube to be rotated interactively. A scale node is also connected to a dial to allow interactive scaling. Any rotation or scale that is applied to the cube is also applied to the character strings.

To display the cube represented by the display structure in Figure 10-6, go to the tutorial demonstration menu and select the program called CHARACTERS.

The cube with its faces labeled will be displayed in three viewports. The rotation node is connected to dials 1, 2 and 3 for rotations in X, Y, and Z. Dial 4 is connected to the scale node. Use the dials to manipulate the cube.

Notice that as you rotate and scale the cube, the character strings on the faces of the cube in viewport 1 are rotated and scaled also. Depth cueing is performed on the characters as well as on the lines that make up the cube.

As you manipulate the cube in viewport 1, the character strings which label its faces are unreadable much of the time. They may be backwards, upside-down, and too small to read. Notice that this is not the case with the characters in viewports 2 and 3. These characters are unaffected by rotations and scales while the object is being transformed. This is achieved by using the SET CHARACTERS command. This command determines the orientation of characters which are part of a model. It has an "orientation" clause with three options: WORLD_ORIENTED, SCREEN_ORIENTED, and SCREEN_ORIENTED/FIXED.

2.4.1 World-Oriented Characters

World-oriented characters are what you are seeing with the cube in viewport 1. The characters are transformed along with the object just like any other part of it. When an object is rotated, translated, or scaled, the characters undergo the same transformations. This is the default condition for any character string or label block you create.

The syntax for this command is as follows.

```
name := SET CHARACTERS WORLD_ORIENTED APPLIED TO name1;
```

2.4.2 Screen-Oriented Characters

Screen-oriented characters are unaffected by ROTATE and SCALE nodes. The SET CHARACTERS command can be used with the SCREEN_ORIENTED clause to maintain a readable orientation for character strings when an object is transformed. The cube in viewport 2 has a SET CHARACTERS SCREEN_ORIENTED node added. When this cube rotates, the names on the cube's faces stay readable. They rotate around the three axes but they stay parallel to the XY plane. When the cube is scaled, the character size remains unchanged.

The syntax for this form of the command is as follows.

```
name := SET CHARACTERS SCREEN_ORIENTED APPLIED TO name1;
```

2.4.3 Screen-Oriented Fixed Characters

Notice that with the screen-oriented characters in viewport 2, the intensity of the characters varies with depth. If the cube were being displayed in perspective projection, the size of the characters would vary too. In the initial position of the cube, the characters **BACK** on the back face of the cube would appear smaller and dimmer than the characters **FRONT**. You can use the SCREEN_ORIENTED/FIXED option of SET CHARACTERS to fix the size and intensity at which characters are displayed.

The cube in viewport 3 has a SET CHARACTERS node with the SCREEN_ORIENTED/FIXED option. Notice that when you rotate this cube, depth cueing is not performed on the characters, so they remain at full intensity.

The syntax for this form of the command follows.

```
name := SET CHARACTERS SCREEN_ORIENTED/FIXED APPLIED TO name1;
```

3. Using Functions to Manipulate Characters and Strings

There are several functions which are used for manipulating characters and strings. These functions convert characters and strings to other types of data, format and reformat strings, transform characters, and perform other miscellaneous character and string-handling operations.

Complete information on these functions is contained in Section *RM2 Intrinsic Functions*. The following sections summarize the functions and give a few examples of their use.

3.1 Character- and String-Conversion Functions

F:CHARCONVERT
Converts characters to integers. The function accepts a string and converts each byte of the string (i.e., each character) to an integer. For example, the string AB will be converted to 65 66, the ASCII decimal equivalent of A and B.

F:CHARMASK
Masks each character in a string by ANDing each byte with a constant integer. This is useful for converting one character or a string of characters to another, for example, from upper to lower case or from a nonprintable to a printable character.

F:PRINT
Converts any data type to a string. For example, a Boolean input will generate the string 'TRUE' or 'FALSE'; a 3D vector will generate a string such as '5,2,1' and so on.

F:TRANS_STRING
Translates one string into an output string using another string as a translation table. For example, prime the function by sending 'ABCDEFGHIJKLMONPQRSTUVWXYZ' as the translation table to input <3> of the function, and 97 (the ASCII decimal equivalent of **a**) to input <2>. If a string of lowercase letters of the alphabet is now sent to input <1>, the letters will be converted to uppercase on output <1>.

F:STRING_TO_NUM
Converts a string to a real number or an integer.

F:GATHER_STRING
Collects strings until a terminator arrives. It then packages them into one string which may or may not include the terminator.

3.2 String-Formatting and Reformatting Functions

F:CONCATENATE
Concatenates strings. The string on input <2> of the function is appended to the string on input <1>.

F:SPLIT
Compares two strings and splits them depending on the match. If a match occurs, characters in the string on input <1> that precede the match are output on output <1>. Matching characters are output on output <2>. Characters following the matching characters are output on output <3>. And a Boolean TRUE is output on output <4>. If no match is found, nothing is output on outputs <1>, <2>, and <3>, and a Boolean FALSE is output on output <4>.

F:PUT_STRING
Replaces characters in the string on input <1> with the string on input <3>, starting at the position specified by the integer on input <2>.

F:TAKE_STRING
Outputs a string consisting of the number of characters specified on input <3> taken from the string on input <1>, starting at the position given on input <2>.

F:LINEEDITOR
Accepts a stream of characters and simple editing commands, accumulates the characters in an internal line buffer, applies the commands to the contents of the line buffer as they are received, and outputs the edited line when a specified delimiter character is recognized.

F:LABEL
Creates a label to send to a LABELS node. A vector on input <1> of the function indicates the location of the label in the coordinate system. A string on input <2> is the text of the label. A Boolean value on input <3> indicates whether the label is to be displayed or not. The data type output by this function can only be used as input to a LABELS node.

3.3 Miscellaneous String-Handling Functions

F:LENGTH_STRING
Accepts a string and outputs its length.

F:FIND_STRING
Determines whether the string on input <2> is a substring of the string on input <1>. Outputs the starting location of the substring if it is found.

F:COMP_STRING
Compares two strings to determine if the string on input <1> is greater than, less than, or equal to the string on input <2>.

F:LBL_EXTRACT
Extracts information about a label in a LABELS node. An integer on input <1> is an index into the LABELS block. A string on input <2> is the name of the node. The function outputs the text of the label, its location in the coordinate system, and a TRUE or FALSE to indicate if the label is displayed or not.

3.4 Character-Transformation Functions

F:CROTATE
Uses an integer on input <1> which represents degrees of rotation to create a 2x2 Z-axis rotation matrix. This matrix can be used to update a CHARACTER ROTATE node.

F:CSCALE
Uses a real number or a two-dimensional vector to create a uniform or nonuniform 2x2 scaling matrix. The matrix can be used to update a CHARACTER ROTATE node.

F:MATRIX2
Accepts two-dimensional vectors on inputs <1> and <2> and creates a 2x2 matrix. This matrix can be used to update a CHARACTER SCALE or CHARACTER ROTATE node.

4. Updating Characters and Labels Nodes

Both CHARACTERS and LABELS nodes can have their contents updated using commands and functions.

4.1 Updating With Commands

The COPY and SEND commands can be used to change the contents of a CHARACTERS or LABELS node.

4.1.1 The COPY Command

Labels can be copied from one LABELS node to another using the COPY command. Note, however, that this command does not work with a CHARACTERS node.

The command has the following format:

```
name := COPY name1 [START=] i [,] [COUNT=] n;
```

The parameters for this command are:

name - The name of the LABELS node you are creating and copying into.

name1 - The name of the LABELS node you are copying from.

i - The number of the first label to be copied.

n - A count of the number of labels to be copied.

The command can be used as follows. First create a labels node called Limerick.

```
Limerick := LABELS  -1,.75   'What''s wrong with this PS 390?'
                    -1,.5    'The frustrated programmer thundered'
                    -1,.25   'I''ve entered commands'
                    -1,0     'With the carefulest of hands'
                    -1,-.25  'But somehow I seem to have blundered!';
```

To see the limerick, scale the labels block by .05 and display it.

```
Scale_Block := CHARACTER SCALE .05 APPLIED TO Limerick;
DISPLAY Scale_Block;
```

Now create a new labels block which starts at the third label and is three labels long.

```
New_Block := COPY Limerick START = 3, COUNT = 3;
```

The words START and COUNT and the equals signs are optional, so you could have typed "COPY Limerick 3,3;" instead. If one word is used, however, both must be used.

Now redefine Scale_Block so that it refers to New_Block.

```
Scale_Block := CHARACTER SCALE .05 APPLIED TO New_Block;
```

The last three lines of the Limerick should now be displayed on the screen.

4.1.2 The SEND Command

Several forms of the SEND command can be used to update a LABELS or CHARACTERS node. Both nodes have similar input queues. Figure 10-7 shows inputs to a CHARACTERS node and Figure 10-8 shows inputs to a LABELS node.

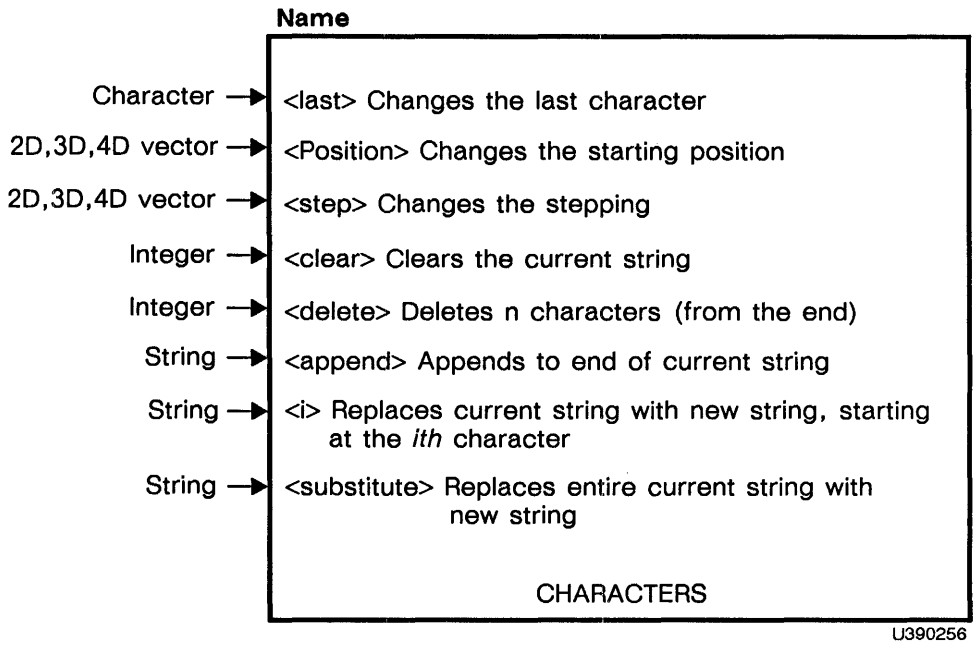


Figure 10-7. Inputs to a CHARACTERS Node

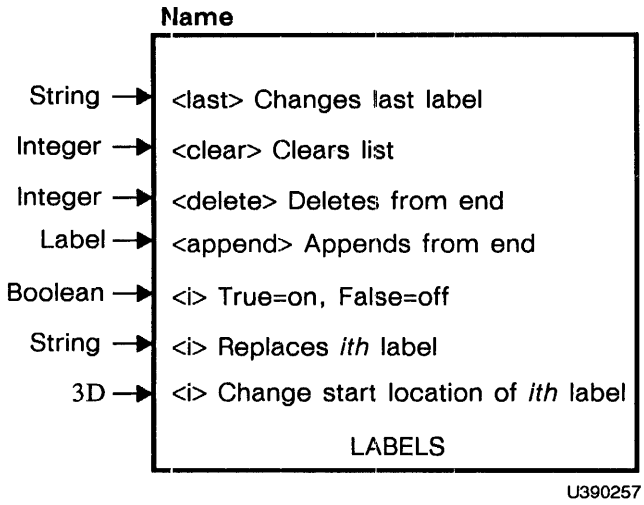


Figure 10-8. Inputs to a LABELS Node

Unlike most other nodes, these nodes have inputs with names as well as numbers. All data sent to these nodes are sent to a named input or to a numeric input which indicates the position of a character within a string or a label within a block.

The simplest form of the SEND command has the following format:

```
SEND option TO <n>name1;
```

The parameters in this command are as follows:

option - For sending to a LABELS node, this is a string enclosed in single quotes. For sending to a CHARACTERS node, the format is CHAR(number), where number is the ASCII decimal equivalent of a single character.

n - The name or number of the input to the LABELS or CHARACTERS node.

name1 - The name of the destination LABELS or CHARACTERS node.

You can use the command, for example, to send a new string to replace an existing one. Create a string called Quote.

```
Quote := CHARACTERS -1,0 'If we had world enough and time';
```

Now scale the string by .05 so it will fit the default window.

```
Scale_Quote := CHARACTER SCALE .05 APPLIED TO Quote;
```

Remove anything you are displaying and display Scale_Quote. Now use the SEND command to replace this string with the second line of John Donne's poem to his reluctant mistress.

```
SEND 'This coyness, mistress, were no crime' TO <substitute>Quote;
```

4.1.3 Exercise

Try SENDing to some of the other inputs of CHARACTERS and LABELS nodes. For more information, refer to Section *RM1 Command Summary*.

Two other forms of the SEND command can be used with LABELS but not with CHARACTERS: they are SEND VL and SEND number*mode. The SEND VL form allows you to overwrite or append a label in a LABELS block. The SEND number*mode form allows you to send a P or L identifier to a label to indicate if a label is off (P) or on (L). Refer to Section *RM1 Command Summary* for more details.

4.2 Updating With Functions

You can create function networks to update a CHARACTERS or LABELS node. Only four data types are accepted by the inputs to these nodes: an integer, a vector (2D or 3D), a character string, and a Boolean value. Any function which outputs one of these data types can be used to feed new values to a node containing text. In particular, the output of the string handling functions mentioned earlier can be used as input to a text node.

The function F:LABEL is designed specifically for updating a LABELS node. The data type output by this function is the only type accepted by input <append> of a LABELS node.

5. Creating and Using Different Character Fonts

A character font is a complete set of characters in the same size and type face. The PS 390 has a standard font consisting of the 128-character ASCII set. This is the default font for all textual displays. You can create and use alternate character fonts. The BEGIN_FONT_... END_FONT command lets

you create an alternate font and the CHARACTER FONT command lets you use that font.

5.1 Creating an Alternate Font

Alternate fonts are created as a sequence of itemized, two-dimensional vector lists defining each character in the font. Up to 128 ASCII character codes can be defined for each font.

Each character in the font is defined as follows.

```
C[i]: N=n vectors;
```

The parameters are:

[i] — The decimal ASCII code to be defined, i.e. a number from 0 to 128.

n — The number of vectors in the 2D vector list.

vectors — The vectors which make up the character.

The vectors which comprise a character must be itemized 2D vectors. Itemized vectors are each preceded by P or L identifiers to indicate whether a vector is a position or a line vector. The following is the definition of a capital **A** in a font called Simplex_Roman.

```
C[65]: N= 6
P  0.5455, 0.9545   L  0.1818, 0.0000
P  0.5455, 0.9545   L  0.9091, 0.0000
P  0.3182, 0.3182   L  0.7727, 0.3182;
```

The Simplex_Roman letter **A** is compared to an **A** in the standard font in Figure 10-9.

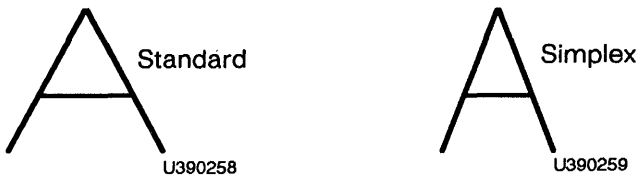


Figure 10-9. Standard A and Simplex Roman A

In an Old English font, the definition of the same letter is much more complex.

```
C[65]: N=49
P  0.2727, 0.8182   L  0.3636, 0.9091   L  0.4545, 0.9545
L  0.5455, 0.9545
L  0.5909, 0.9091   L  0.9091, 0.1818   L  0.9545, 0.1364
L  1.0455, 0.1364
P  0.5000, 0.9091   L  0.5455, 0.8636   L  0.8636, 0.1364
L  0.9091, 0.0455
L  0.9545, 0.0909   L  0.8636, 0.1364   P  0.3636, 0.9091
L  0.4545, 0.9091
L  0.5000, 0.8636   L  0.8182, 0.1364   L  0.8636, 0.0455
L  0.9091, 0.0000
L  0.9545, 0.0000   L  1.0455, 0.1364   P  0.2727, 0.6364
L  0.3182, 0.6818
L  0.4091, 0.7273   L  0.4545, 0.7273   L  0.5000, 0.6818
P  0.4545, 0.6818
L  0.4545, 0.6364   P  0.3182, 0.6818   L  0.4091, 0.6818
L  0.4545, 0.5909
P  0.0455, 0.0000   L  0.1364, 0.0909   L  0.2273, 0.1364
L  0.3636, 0.1364
L  0.4545, 0.0909   P  0.1818, 0.0909   L  0.3636, 0.0909
L  0.4091, 0.0455
P  0.0455, 0.0000   L  0.1818, 0.0455   L  0.3182, 0.0455
L  0.3636, 0.0000
L  0.4545, 0.0909   P  0.5455, 0.7727   L  0.2727, 0.1364
P  0.3636, 0.3636
L  0.7273, 0.3636;
```

This letter **A** is compared to the standard font **A** in Figure 10-10.

Figure 10-10. Standard A and Old English A

A complete set of character definitions is enclosed in a BEGIN_FONT ... END_FONT structure with the following format.

```
New_Font := BEGIN_FONT

            C[0]: N=n P, L, L, ... L;

            C[n]: N=n P, L, L, ... L;

            C[127]: N=n P, L, L, ... L;

            END FONT;
```

Notice that in the sample 2D vector lists given, the range of the vectors in X and Y is between 0 and 1. There is no limit on the range of the vectors you use, but you should keep within the range of 0 and 1 for the correct spacing and orientation of adjacent characters.

5.2 Using an Alternate Font

The BEGIN_FONT ... END_FONT command does not create a data node in a display structure but a look-up table of alternate character definitions. To switch to an alternate font in a structure, the CHARACTER FONT command is used to create an attribute node which indicates the font look-up table that must be read for the character definitions.

An alternate font called Old_English is included in the *PS390 Tutorial Demonstration Programs*. To use this font in a structure, you must create a node which points to the Old_English font and apply it to the text you want to display.

Create, scale, and display a character string.

```
Text := CHARACTERS -.5,0 'To be, or not to be';
Scale_Text := CHARACTER SCALE .05 APPLIED TO Text;
DISPLAY Scale_Text;
```

Now apply a CHARACTER FONT command to the scaled string to display it in the Old_English font.

```
New_Font := CHARACTER FONT Old_English APPLIED TO Scale_Text;
REMOVE Scale_Text;
DISPLAY New_Font;
```

Hamlet's question should now be displayed in the Old_English font. If it is displayed in the standard font instead, this means that the Old_English font was not available.

The display structure for New_Font is shown in Figure 10-11.

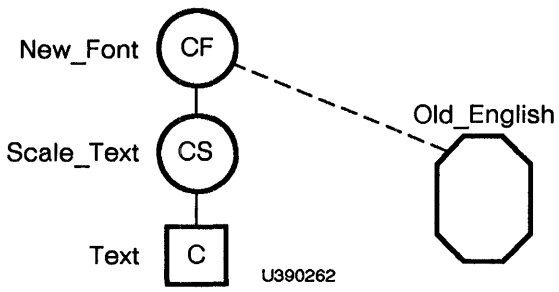


Figure 10-11. Display Structure with CHARACTER FONT Node

The Old_English font is shown as a look-up table which is not part of the actual structure. The CHARACTER FONT node New_Font points to this table as well as to the CHARACTER SCALE and CHARACTERS node.

5.3 The Character-Font Editor Program

Another way to create alternate character fonts is to use the program MAKEFONT which is distributed on the E&S magnetic tape and is documented in Section *TT7 Character Font Editor*. MAKEFONT is a menu-driven, graphical character font editing program which allows you to create a font from scratch by drawing each of the characters, or to make changes to existing alternate fonts.

6. Summary

Two commands create data nodes containing text: CHARACTERS and LABELS.

6.1 Creating Text Nodes

The CHARACTERS command creates a single text string of up to 240 characters. Optional parameters allow you to specify the starting location of the string and the horizontal and vertical spacing between characters. The syntax of the command is as follows.

```
name := CHARACTERS [x,y[,z]][STEP dx,dy] 'string';
```

The LABELS command creates a block of character strings or labels. Each label can be given its own starting location. The syntax of the command is as follows.

```
name := LABELS   x,y [,z] 'string'
                     .
                     .
                [xi,yi [,zi] 'string'];
```

6.2 Manipulating Text With Commands

Text nodes, just like any other data nodes, are affected by transformations. They can be rotated and scaled by 3x3 transformation matrices (created by the ROTATE and SCALE commands) or by exclusive 2x2 character transformation matrices.

6.2.1 Transforming Text

The commands which create these matrices are CHARACTER ROTATE, CHARACTER SCALE, and TEXT SIZE. The matrices which these commands create have no effect on three-dimensional data or nontextual two-dimensional data.

The CHARACTER ROTATE command creates a Z-rotation matrix from an angle of rotation which is entered as parameter. The syntax of the command is as follows.

```
name := CHARACTER ROTATE angle [APPLIED TO name1];
```

The CHARACTER SCALE command creates a uniform or nonuniform scaling matrix from the scale factor entered with the command. For nonuniform scaling an X and Y scale factor is given. The syntax of the command is as follows.

```
name := CHARACTER SCALE s [APPLIED TO name1];
name := CHARACTER SCALE sx,sy [APPLIED TO name1];
```

The TEXT SIZE command creates a 2x2 matrix node which overrides any 2x2 matrix settings above it in the display structure. Any character scales or

character rotations are superseded by this command. The command establishes a character size for text which is a multiple or fraction of the default character size of 1. The syntax of the command is as follows.

```
name := TEXT SIZE x [APPLied to name1];
```

6.2.2 Setting Character Orientation

When text forms part of an object that is being displayed and manipulated, the characters can be transformed with the object or they can remain unaffected by object transformations. The SET CHARACTERS command lets you determine the orientation of the text. The format of the command is as follows.

```
name := SET CHARACTERS orientation [APPLIED TO name1];
```

Three types of orientation may be set:

World_Oriented — Characters are transformed just like any part of the object containing them.

Screen Oriented — Characters are not affected by ROTATE or SCALE transformations. Intensity and size of characters still vary with depth (Z-position).

Screen_Oriented/Fixed — Characters are not affected by ROTATE or SCALE transformations. They are always displayed with full size and intensity.

6.3 Manipulating Text With Functions

Several functions are available for manipulating text and strings. These functions are listed below.

- Character- and String-Conversion Functions

  F:CHARCONVERT
  F:CHARMASK
  F:GATHER_STRING
  F:PRINT
  F:STRING_TO_NUM
  F:TRANS_STRING

- String-Formatting and Reformatting Functions

  F:CONCATENATE
  F:LABEL
  F:LINEEDITOR
  F:PUT_STRING
  F:SPLIT
  F:TAKE_STRING

- Miscellaneous String-Handling Functions

  F:COMP_STRING
  F:FIND_STRING
  F:LBL_EXTRACT
  F:LENGTH_STRING

- Character-Transformation Functions

  F:CROTATE
  F:CSCALE
  F:MATRIX2

6.4 Text Nodes

The CHARACTERS and LABELS commands create data nodes containing text. Both nodes have inputs which accept vectors, strings, integers, or Boolean values to update the contents of the node.

6.5 Updating Nodes

CHARACTERS and LABELS nodes can be updated using commands or the functions listed earlier. The following commands are most frequently used to update these nodes.

COPY

SEND

SEND VL

SEND number*mode

6.6 Alternate Character Fonts

Character fonts other than the standard font can be created using the BEGIN_FONT ... END_FONT command. The syntax for this command is as follows.

```
name := BEGIN_FONT
            [C[0]: N=n {itemized 2D vectors};]
               .
               .
               .
            [C[i]: N=n {itemized 2D vectors};]
               .
               .
               .
            [C[127]: N=n {itemized 2D vectors};]
        END_FONT;
```

Each character in the font is defined as a vector list consisting of itemized 2D vectors. The clause C[i]: identifies the ASCII character being defined; for example, C[65]: indicates that the character is a capital **A**. Up to 128 characters can be defined in an alternate font. Alternate fonts are used by including CHARACTER FONT nodes in a display structure. The syntax of the CHARACTER FONT command is as follows.

```
name := CHARACTER FONT font_name APPLIED TO name1;
```

The parameter **font_name** is the name of an alternate font defined with the BEGIN_FONT ... END_FONT command.

GT11. PICKING

GT11. PICKING

SELECTING DISPLAYED OBJECTS

CONTENTS

ILLUSTRATIONS

Section GT11
Picking
Selecting Displayed Objects

Introduction

Picking allows you to retrieve information about a selection or pick made on displayed data. This information contains details about the structure that makes up the displayed data. Details can include the name of the data node that the picked portion of the object is associated with; names of nodes along the branch of the display structure that was selected by a pick; an index into the vector list, character string or label that was picked; and the coordinate values of the location where the pick took place. The information is available in a special format called the pick list.

Normally, picking is done by using the data tablet and the stylus to select any part of a displayed object designed to allow for picking. The selection is made by moving the stylus across the surface of the data tablet; this positions the cursor on the screen. (The cursor is an X.) Picking is usually activated by pressing the tip of the stylus down when the cursor is positioned over the appropriate line, dot, or text character. The information that is returned when a pick takes place (the pick list) can be displayed, used to drive a function network, or sent to the host. The amount and kind of information received on the location of a pick is user-definable.

An obvious use of picking is to make selections from a menu, where the cursor is positioned over a line or the piece of text in the menu that is to be selected. By pressing the stylus down, that item on the menu is picked, and the appropriate function can be performed (i.e., move to another menu, exit from the menu, bring up a displayable structure, etc.)

Central to the picking process is the initial function instance PICK. PICK is enabled by sending any message to input <1> of PICK. (Normally this message is the X,Y location of the pick sent to PICK when the tipswitch of the stylus is pressed.) PICK feeds this trigger message to the display processor, asking for any pick information

within the data structure being traversed to be sent back to PICK. If this information is found (a pick occurs if there is data) the pick list is placed on the queue of output <1> of PICK. The main responsibility of PICK is to signal the display processor that picking has been enabled and to output the pick list that contains information about the location of the pick.

Before picking can take place, the data structure that you want to be able to pick from must contain certain nodes and pieces of information. Polygonal objects, because of their construction, cannot be picked.

This section defines the various elements involved in picking: picking-attribute nodes and the commands that create them, and the picking functions.

This section teaches how to place and set the appropriate attribute nodes used in picking and how to design a function network to use the information that is generated when a pick occurs.

Objectives

This section teaches you how to:

- Use picking-attribute nodes.
- Use initial picking functions instances for picking.
- Use the picking functions in a function network.

Prerequisites

You need to be familiar with the concepts presented in Sections *GT4 Modeling, GT5 Command Language, GT6 Function Networks I* and *GT7 Function Networks II.*

1. Using Picking-Attribute Nodes

Before an object can be picked, the display structure of the object must contain certain nodes, and the object must be displayed. These nodes provide for picking capabilities such as:

- Turning picking on and off.
- Determining the portions of the object (or branches of the display structure of the object) that can be picked.
- Selecting the name of the pick identifier that will be returned as part of the pick list.

1.1 Setting Picking ON and OFF

The first picking-attribute node that must appear in the display structure is the SET PICKING ON/OFF node. This node must be above the parts of the display structure where picking will take place. This node is turned on and off by Boolean values; a TRUE will enable picking in the data structure below the node, a FALSE will disable it.

The command that creates the SET PICKING ON/OFF node is:

```
name := SET PICKING OFF APPLIED TO name1;
```

The SET PICKING ON/OFF node is usually placed in the display structure in an OFF condition and activated when the Boolean value TRUE is sent to input <1> of the named node. As an example, the following two commands first create an instance of a SET PICKING ON/OFF node, and then activate that node:

```
Pick_Car := SET PICKING OFF APPLIED TO Car;
```

where Car is the name of the data structure, or the part of a data structure that you want to be able to pick from;

```
SEND TRUE TO <1>Pick_Car;
```

activates picking for Car. (The Boolean value is normally sent by a network connected to the node.)

In designing a pickable display structure, the placement of the SET PICKING ON/OFF nodes is very important. As with any other attribute node, this node controls only its descendants. In the structure in Figure 11-1, picking can be enabled and disabled for each branch individually because of the placement of the SET PICKING ON/OFF nodes. In Figure 11-2, picking is established for the whole structure, but not for the individual branches.

This placement can be important in complicated display structures, where there are close or overlapping data structures simultaneously displayed on the screen. In molecular modeling graphics applications, it can be useful to disable picking for specific parts of the molecule. This same principle holds for architectural or engineering applications, where only specific parts of the entire display are used as pickable structures.

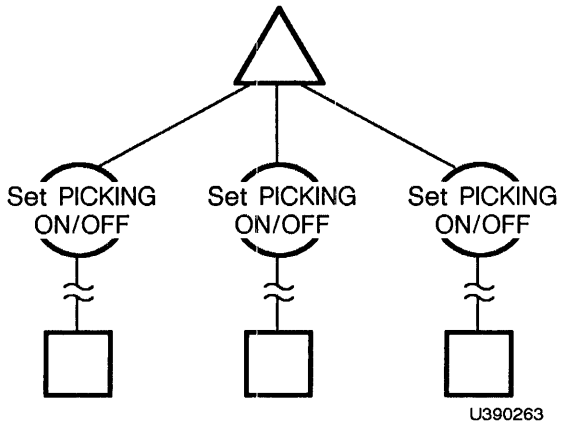


Figure 11-1. Picking Selectable by Branch

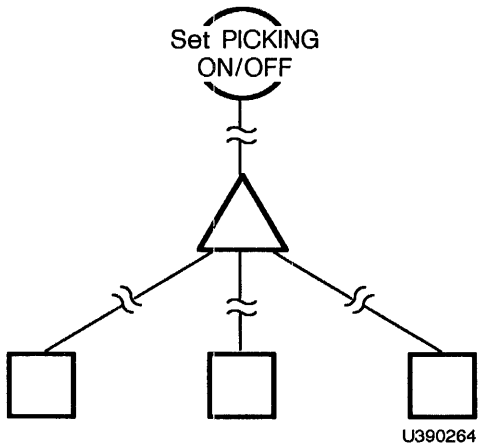


Figure 11-2. Picking an Entire Structure

1.2 Using Picking Identifiers

The other attribute node that must be placed in the display structure for picking is the SET PICKING IDENTIFIER node. This pick identifier node determines the detail of the information you get back in your pick list.

A picked object is identified by two types of names in the pick list. The first type of name is the picking identifier or the pick id. The second name is the name of the data node that contains the picked vector or character. In the command shown above, "Car" is the name of the node that contains the picked vector.

The command to create a SET PICKING IDENTIFIER node is:

```
name := SET PICKING IDENTIFIER = id_name APPLIED TO name1;
```

This command assigns **id_name** to be the picking identifier (the reported character string) to be output by PICK in the pick list if any part of name1 is picked. **id_name** can be the name of the data node, but in many cases several branches of a display structure terminate at the same data node. The name(s) of the pick identifiers in the pick list in such cases show which branch was traversed to get to the common data node.

1.2.1 Example

```
WheelPick1 := SET PICKING IDENTIFIER = Wheel1 APPLIED TO Wheel;
```

In this example, it is assumed that the display structure includes a car with four tires. There are five branches, four of which include an instance of the vector list for "Wheel." Each branch contains the appropriate translation and rotation operation nodes required to position the tires. To determine which instance of "Wheel" was picked, each branch must also contain a SET PICKING IDENTIFIER node with a unique name. This is illustrated in Figure 11-3.

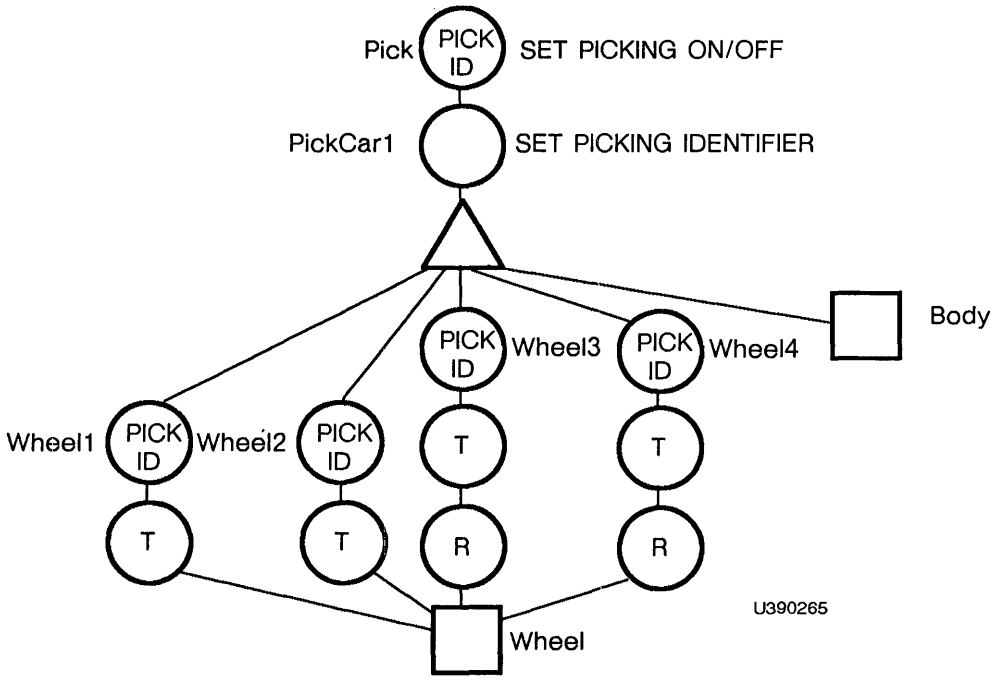


Figure 11-3. Display structure With Car and Four Tires

Assuming the right-front tire is Wheel1, then the pick list generated when a pick was made on the right-front tire would be:

```
<index> Wheel1,PickCar1 Wheel
```

If there were only one SET PICKING IDENTIFIER node directly below the SET PICKING ON/OFF node in Figure 11-3, when you picked from any part of the displayed object below the instance node, you would only get back the pick identifier for the whole data structure:

```
<index> PickCar1 Wheel (or Body)
```

The information in a pick list includes the names of all the SET PICKING IDENTIFIER nodes down the branch of the display structure enabled for picking. The pick list also includes the name of the picked data node. The pick list can be reported as a character string with pick IDs on that branch separated by commas. This list always starts with the name of the SET PICKING IDENTIFIER node closest to the picked vector or character.

The amount of detail about the display structure contained in information returned in the pick list is determined by the location and number of the SET PICKING IDENTIFIER nodes. In the code below, the pick list contains only one pick identifier (PickCar1).

```
DISPLAY Car;

 Car  := BEGIN_STRUCTURE
 Pick := SET PICKING OFF;
         SET PICKING IDENTIFIER = PickCar1;
         INSTANCE OF Body, Wheel1, Wheel2, Wheel3, Wheel4;
         END_STRUCTURE;
```

To set up the display structure to enable picking remember the following:

For picking to take place, there must be a SET PICKING ON/OFF node placed in the display structure, followed by at least one SET PICKING IDENTIFIER node down each pickable path. However, one structure can contain multiple SET PICKING ON/OFF nodes, and each SET PICKING ON/OFF node can be followed by multiple SET PICKING IDENTIFIER nodes.

2. Using Initial Picking Functions

The initial system function PICK was briefly described in the introduction to the section. The initial function network that should be built to make use of picking is shown in Figure 11-4.

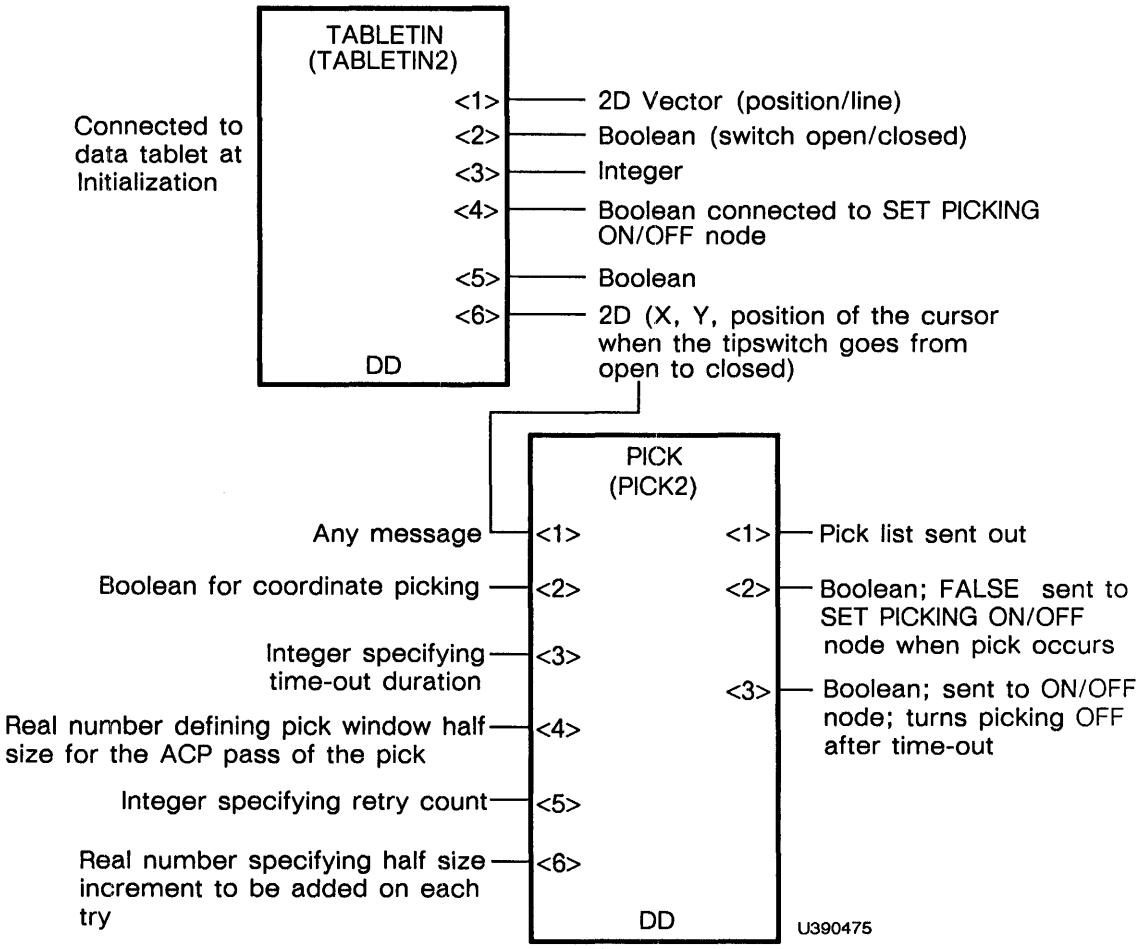


Figure 11-4. Diagram of TABLETIN and PICK

The system provides for picking with one other initial function, TABLETIN. TABLETIN accepts the X,Y vectors that identify the position of the picking location (the center of the cursor cross) as the stylus moves across the data tablet and uses these vectors to position the cursor on the screen. TABLETIN identifies the X,Y coordinates of the picking location that are

output when the tipswitch on the stylus is pressed. These coordinates are used to determine if a pick has occurred; and if it has, the location of the pick is made available.

Output <4> of TABLETIN is typically connected to the SET PICKING ON/OFF nodes in the display structure and is used to send Boolean values to the nodes. When the tipswitch on the stylus is pressed, a TRUE is sent to the node, enabling picking.

Input <1> of PICK accepts any message. Typically, this queue is connected to output <6> of TABLETIN which supplies the 2D coordinates of the pick location when the tipswitch is pressed. This arms the function, as the other two inputs to PICK are constants. Output <2> of PICK should be connected to the same SET PICKING ON/OFF nodes that are connected to output <4> of TABLETIN. This output sends a FALSE whenever a pick occurs which turns picking off until the tipswitch is again pressed and a TRUE is sent from TABLETIN to the ON/OFF node. (This FALSE is sent to disable picking so that the picking process ceases until a pick location is asked for.)

Input <2> of PICK accepts a Boolean value that allows you to select the kind of pick list that will be sent out of output <1>. A FALSE sent to input <2> of PICK indicates that the output pick list includes the pick ID names, the data node name, and an index into the vector list or character string (the data node). A TRUE sent to input <2> of PICK indicates that the pick list includes the pick ID names, the data node name, an index to the data node, and the picked coordinates and the dimension (2D or 3D) of the picked vector.

The format for the pick list then, with FALSE sent to input <2> PICK is:

```
<index> PickId1,PickId2,Name_of_Data_Node
```

where <index> is a pointer into the picked data node.

The following chart shows the data node types and the definition of the <index> that is returned when the value of the <index> is the integer 3.

Data Node Type	Definition for Index Value of Integer 3
Vector list	The third vector in the list was picked.
Character string	The third character in the string was picked.
Label	The third character string in the label was picked.
Polynomial or Rational polynomial curve	The value of the parameter (t) where the curve was picked

The format for the pick list with TRUE sent to input <2> of PICK (coordinate picking) is:

```
<index> [x,y,z] PickId1,PickId2,Name_of_Data_Node
```

where **X,Y,Z** are the coordinate points of the picked vector.

Performing coordinate picking on a character string returns an index into the string, not its picked coordinates.

Performing coordinate picking on a label block returns an index into the label, not its picked coordinates.

Coordinate picking cannot be performed on a vector over 500 units long.

The integer on input <3> of PICK is used to set a time-out interval for the PICK function in refresh frames. Timing starts when the PICK function receives any message on input <1>. This timing interval is used to determine if a pick occurs in the specified period of time. The allowable integers on input <3> are from 4 through 60. This is a safeguard feature: it deactivates PICK if no pick occurs within the time-out period.

Input <4> is a real number between 0 and 1 that defines the pick window half-size for the ACP pass of the pick. This is different from the size set by the SET_PICKing_LOCation operation node. The line generator or the frame buffer uses the operation node to determine if a pick has occurred, while the ACP uses the value placed on input <4> to do the actual pick pass on the data.

Input <5> is an integer specifying pick pass retries. Since it is possible that the ACP will not find the picked data during a pick pass, input <5> indicates the number of times to add the window half-size increment on input <6> and try another pick pass.

Input <6> is a real number between 0 and 1 which specifies the amount to increase the pick window half-size on each retry of the pick pass. The defaults for inputs <4>, <5> and <6>, are:

Input <4> 6.8359E-3
Input <5> 4
Input <6> 6.8359E-3

Once the PICK function is armed (by receiving input on input <1>), if no pick occurs within the specified time, PICK outputs a FALSE on output <3>. This output should be connected to the ON/OFF nodes to disable picking when a time-out occurs. Picking is enabled when the stylus is again pressed.

One other feature that is initialized by the system is the picking location. This is by default the center of the cursor. The picking location must be defined within the current viewport and can be modified with the following command:

```
name := SET PICKING LOCATION = x,y sizex,sizey APPLIED TO name1;
```

where:

the 2D vector **X,Y** specifies the center of the picking location and the 2D vector **sizex,sizey** specifies the size in X and Y from the center to the edge of the picking location. **name1** is the structure to which the pick location applies.

The pick location, then, specifies a region within a screen. If the pick-sensitive object (line, dot, or character) is within the pick location, it can be reported as having been picked.

The pick location can be moved within the viewport by sending the 2D vector that represents the coordinate location of the new set pick location to input <1> of the set picking location node. In effect, picking can take place by positioning the picking location over a displayed object (containing the appropriate picking attribute nodes) and sending a TRUE to input <1> of PICK.

Figure 11-5 shows a typical arrangement of the TABLETIN and PICK functions and their connections to the display structure.

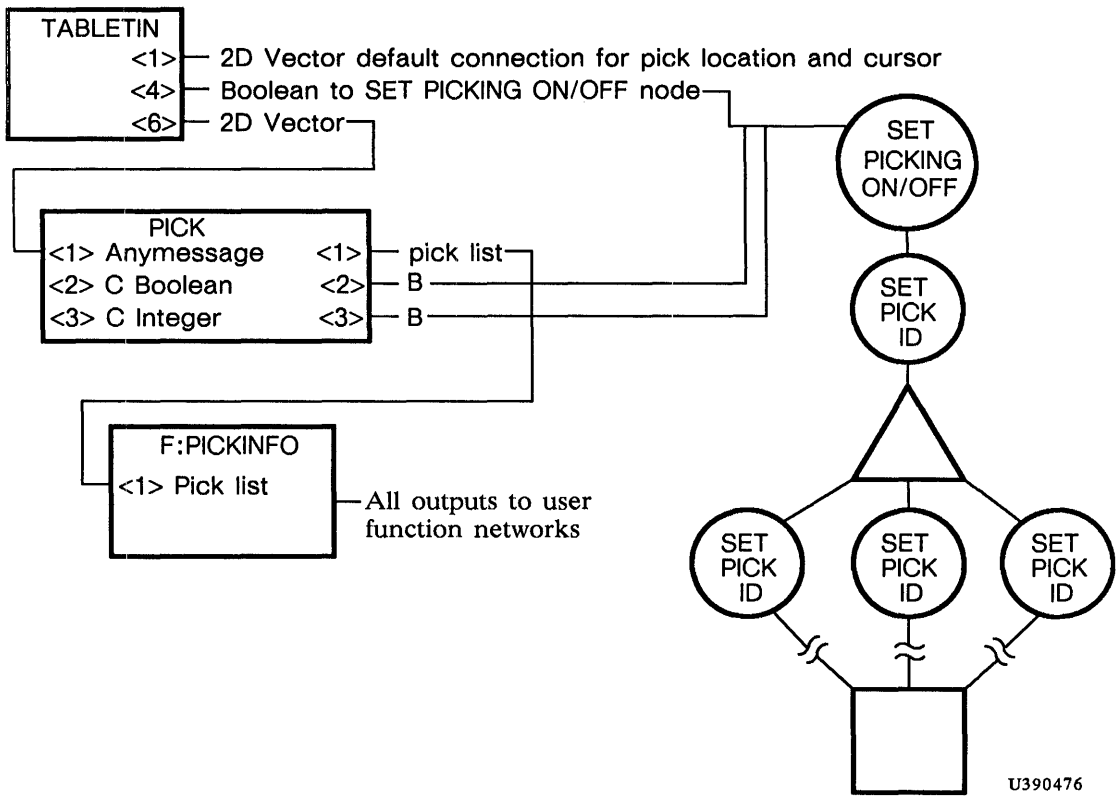


Figure 11-5. Typical TABLETIN and PICK Arrangement

3. Using the Picking Functions in a Function Network

A function associated with picking is F:PICKINFO. This function converts the pick list data type into character strings that are acceptable by other functions. There is only one active input to F:PICKINFO, <1>, and it should be connected to output <1> of PICK.

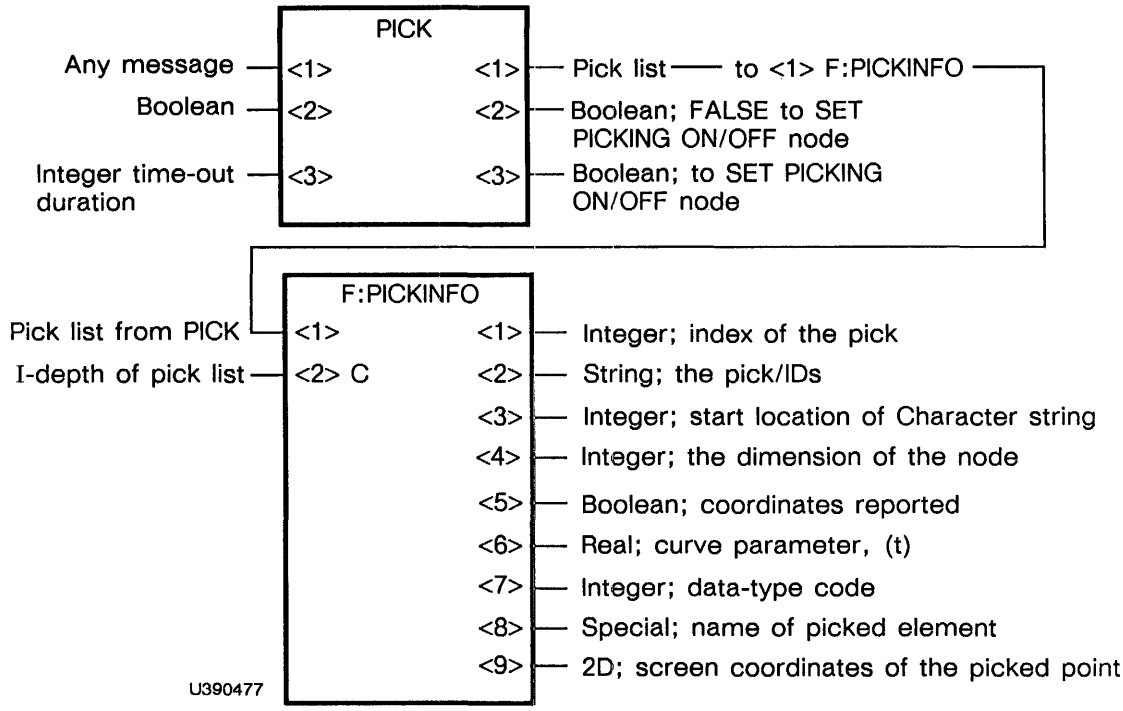


Figure 11-6. F:PICKINFO (Connected to PICK)

The pick list sent from output <1> of PICK can be connected to an instance of F:PICKINFO to convert the pick list into a logically useful format. The pick list can also be printed out or displayed by connecting output <1> of PICK to F:PRINT. F:PRINT converts the pick list code to printable characters.

The constant input <2> of F:PICKINFO accepts an integer that specifies the depth of the pick identifiers that will be output. Since the pick list contains all of the pick IDs in a picked branch of a display structure, this input allows you to select the depth. For example, if there were four pick IDs active when a pick occurred and the integer 2 was sent to input <2> of F:PICKINFO, then the two pick IDs closest to the data node and the name of the data node itself are output as the string on output <2> of F:PICK-INFO.

The output information from F:PICKINFO varies with the type of pick list supplied on input <1>. If the PICK function has a TRUE on input <2>, then it supplies a detailed coordinate pick list and most of F:PICKINFO outputs are activated. If the PICK function has a FALSE on input <2>, a less detailed pick list is supplied, and only outputs <1>, <2>, and <5> are active.

Refer to Section *RM2 Intrinsic Functions* for a complete description of the outputs of F:PICKINFO.

The best use of picking is when the pick list is sent to an instance of F:PICKINFO. Then information generated by the function can be used to drive function networks that can be triggered by typical data types. Examples of what this data can be used for are described in the next section.

3.1 Examples of Picking

The following example demonstrates how picking can be used to trigger a switching network for an object designed to have parts with independent motion. The control dials are normally used to rotate, translate, and scale objects in three dimensions. If the designed object requires more than eight elements of freedom (the maximum number that can be provided by one set of control dials), a picking network can be set up to access a bank of switching functions that control the output of the dials. This network will allow you to point at the part that you want to manipulate and the picking information will drive the function network that routes the dial outputs to various networks.

In this example, the display structure that defines a robot figure includes SET PICKING IDENTIFIER nodes in each branch of the figure networked for motion through a switch function to DIALS. This is the same robot that was built in Section *GT5 Command Language*, and it is connected to the function networks that were designed in Sections *GT6 Function Networks I* and *GT7 Function Networks II*. The function network provides for several modes for the control dials. These modes provide the triggers to animate each part of the robot that requires independent movement, i.e., rotation of each shoulder joint, knee joints, torso, head, etc.

The picking network will use the data tablet to trigger the mode of the dials. In Section *GT6 Function Networks I*, the function keys were used for dial-mode switching. If you examine the design of the robot, you will notice that there are 41 elements of freedom designed into the structure. This will require 41 modes of the dials. As the picking network will be used to trigger the dials mode, 41 SET PICKING IDENTIFIER nodes must be coded into the structure.

The picking network to switch the modes for dials that are connected to the robot display structure works in the following manner. When the cursor is positioned over a part of the robot with independent motion controlled by a dial (like the shoulder) and the tipswitch of the stylus is pressed, the name of the pickID in the shoulder branch of the display structure is sent from PICK to an instance of F:PICKINFO.

Output <2> of this instance of of F:PICKINFO is connected to an instance of F:CHARCONVERT. F:CHARCONVERT converts the bytes of the string it receives on input <1> into a stream of integers. If the pick ID sent to F:PICKINFO is A, F:CHARCONVERT will translate A to the ASCII 65. If this is then sent to an instance of F:SUBC, it can subtract 64 and output the integer 1 that can be used to trigger the appropriate bank of switches for the dials.

Figure 11-7 illustrates the function network described above.

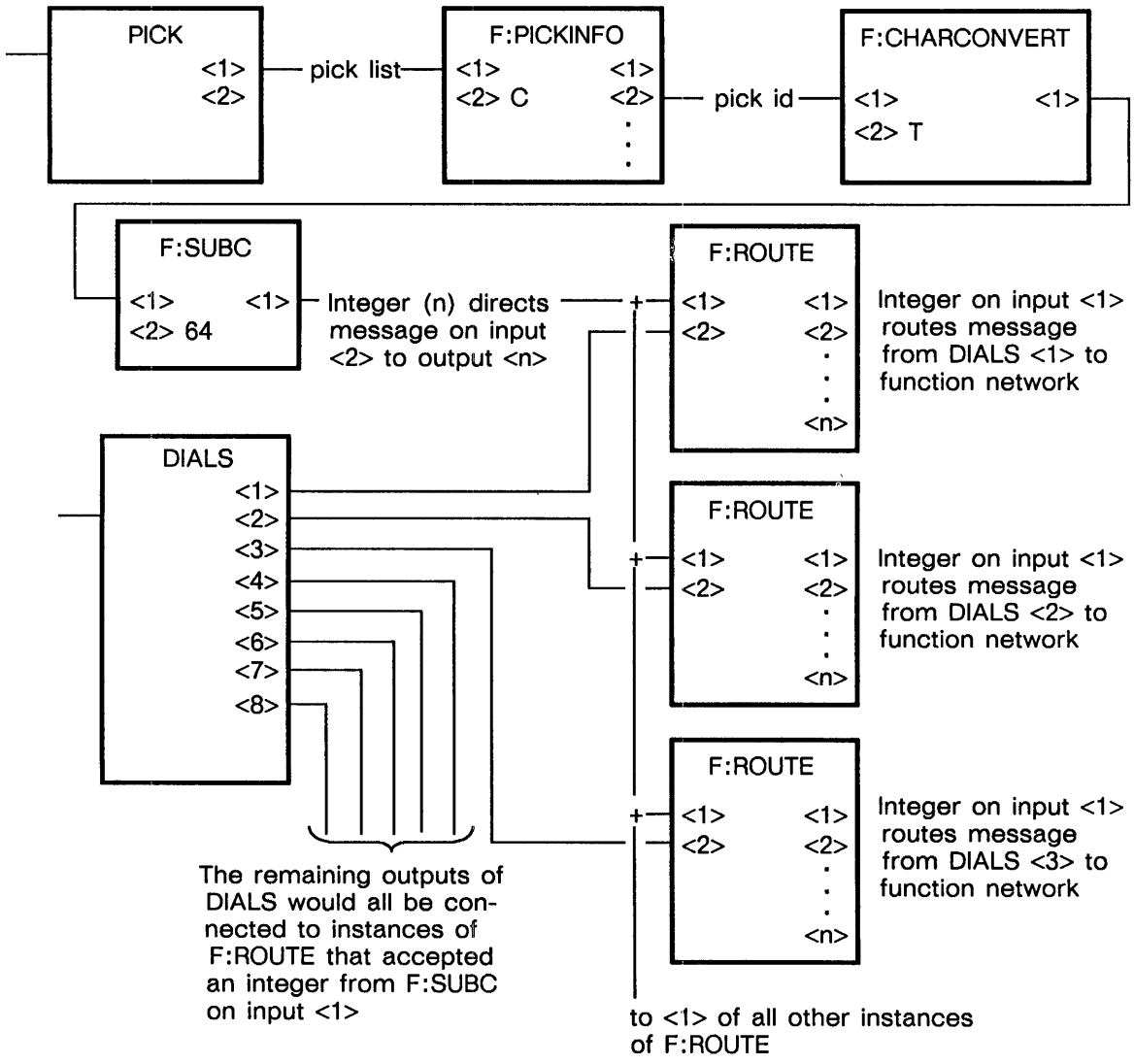


Figure 11-7. Diagram of PICK Through F:SUBC Feeding a Bank of F:ROUTE(n) Instances

To implement the previous example of picking as an exercise demonstrating the placement of the picking-attribute nodes and the connections that should be made for the picking network, use the source code supplied for the robot in Section *GT5 Command Language*. Picking attribute nodes can be set into the display structure and then connected to the picking function network that is used in the picking demonstration available in the PS 390 Tutorial Demonstration Programs.

3.2 Exercise

Design a pickable display structure with several instances of a primitive.

Design a function network that outputs the pick list to the screen. Use F:PRINT and a character data node. Code your display structure and function network. Display and pick each primitive.

4. Summary

Picking allows you to retrieve information about a selection made on displayed data. The information is available in a special format called the pick list. Before picking can take place, the data structure that you want to be able to pick from must contain certain nodes and pieces of information.

- Picking-Attribute Nodes

  The first picking-attribute node that must appear in the display structure is the SET PICKING ON/OFF node. This node must be above the parts of the display structure where picking will take place. This node is turned on and off by Boolean values; a TRUE will enable picking in the data structure below the node, a FALSE will disable it.

  The command that creates the SET PICKING ON/OFF node is:

```
name := SET PICKING OFF APPLIED TO name1;
```

  The other attribute node that must be placed in the display structure for picking is the SET PICKING IDENTIFIER node. This pick identifier node determines how detailed the information you get back in your pick list will be.

  A picked object is identified by two types of names in the pick list. The first type of name is the picking identifier or the pick ID. The second name is the name of the data node that contains the picked vector or character.

  The command to create a SET PICKING IDENTIFIER node is:

```
name := SET PICKING IDENTIFIER = id_name APPLIED TO name1;
```

For picking to take place, there must be a SET PICKING ON/OFF node placed in the display structure, followed by at least one SET PICK IDENTIFIER node down each pickable path. However, one structure can contain multiple SET PICKING ON/OFF nodes, and each SET PICKING ON/OFF node can be followed by multiple SET PICKING IDENTIFIER nodes.

- Picking Functions

  The initial function instance used for picking is PICK. Input <1> of PICK (usually connected to output <6> of TABLETIN) accepts any message type as a trigger message to activate picking. The pick list is placed on the queue of output <1> of PICK. The main responsibility of PICK is to signal the display processor that picking has been enabled and to output the pick list.

  An intrinsic user function associated with picking is F:PICKINFO. This function converts the pick list data type into character strings that are acceptable by other functions. There is only one active input to F:PICKINFO, <1>, and it should be connected to output <1> of PICK.

GT12. VIDEO OUTPUT CONTROL

GT12. VIDEO OUTPUT CONTROL

CONTENTS

TABLES

Section GT12
Video Output Control

This section describes how to control the video output of the PS 390 graphics system. It describes how to select a background color, how to select the configuration and color of the screen cursor, how to select a video timing format, and how to select filters to implement antialiasing.

Objectives

This section provides the following information:

- Basic raster display concepts
- How to select background color
- How to select a cursor color
- How to define the shape of a cursor
- How to select a video timing format
- How to select a line filter

1. Video Theory and the PS 390

This section explains general raster video output concepts and how the PS 390 handles video output.

1.1 Calligraphic and Raster Displays

There are two types of displays used in computer graphics: calligraphic displays and raster displays. Calligraphic displays draw lines on the screen by moving the electron beam of the display along the line. The calligraphic display produces superior quality lines, but has problems matching the end points of lines, uses much more power than a raster display, has limited polygon fill and no polygon rendering capabilities, and can draw only at limited speeds, which causes the display to flicker when complex objects are displayed.

Raster displays, such as the displays used with the PS 390, always have their electron beam draw the same grid pattern, but adjust the brightness of the beam to display a picture. The raster pattern is a grid composed of picture elements known as pixels. Resolution is defined in terms of pixels; the default PS 390 video timing format has a resolution of 1024 pixels (in the horizontal direction) by 864 pixels (in the vertical direction).

A raster display draws horizontal lines of pixels known as "scan lines," starting at the upper left-hand corner of the screen. The scan lines are drawn from left to right in succession from the top to the bottom of the screen. Since the lines are always being drawn, a resonant circuit is used to move the electron beam, which means that a raster display uses only about one tenth the power of a calligraphic display.

Raster displays can produce solid renderings, but in the past have been unable to match the line drawing quality of calligraphic displays due to a phenomenon known as "aliasing." Angled lines (such as a 45 degree line) take on a "stairstepped" effect due to the approximately square shape of the pixels. The PS 390 solves this problem by using a set of programmable line filters to smooth out the jagged angled lines, producing lines that approach calligraphic-display line quality. This revolutionary antialiasing capability of the PS 390 is achieved through proprietary, high-speed custom VLSI circuitry, known as Shadowfax™ technology.

1.2 Raster Display Characteristics

Raster displays are either interlaced or non-interlaced. Non-interlaced displays draw all of the horizontal lines each time they draw from the top to the bottom of the screen. All the lines drawn on the screen make up a "frame." The default PS 390 video timing format is non-interlaced.

Commercial television is an example of an interlaced display. Interlaced displays draw the first scan line on the top of the screen, then draw the third scan line, then the fifth, and so on to the bottom of the screen, drawing all the odd numbered scan lines. The beam then goes back to the top of the screen and draws all of the even numbered scan lines. The odd numbered scan lines are called the odd field and the even numbered scan lines are called the even field. The two fields make up one frame.

Interlaced displays can have the same resolution as a non-interlaced display with only about 60% of the performance of the non-interlaced display because the interlaced display draws a given pixel only one half as often as the non-interlaced display. The field rate is still 60 Hz, so there is not a problem with flicker. Interlaced displays produce some peculiarities (artifacts) when there is motion in the picture. If the eye follows an object that is moving vertically, the object can move one scan line per field, and the eye will see the two fields superimposed on each other. The object appears to have horizontal black lines drawn through it. This is one of the class of artifacts known as "temporal aliasing." One familiar example of temporal aliasing is wagon wheels appearing to turn backwards in western movies.

1.3 Video Timing Formats

The PS 390 software supports three video timing formats:

- 1024 by 864 non-interlaced
- 1024 by 864 interlaced (RS-343-A)
- 640 by 484 interlaced (RS-170-A)

The 1024 by 864 non-interlaced format is the default PS 390 video timing format. This format does not conform to any established standard video timing format.

The RS-343-A 1024 by 864 interlaced PS 390 video timing format conforms to the Electronics Industry Association (EIA) RS-343-A standard for "Electrical Performance Standards for High Resolution Monochrome Closed Circuit Television Camera." This standard defines a generic high-resolution interlaced timing format. This format is used for color cameras, projection systems, and other medium performance, high-resolution devices.

Black and white television in the U.S. conforms to Electronics Industry Association (EIA) standard RS-170. Color television in the U.S. conforms to EIA standard RS-170-A. The color standard defines an encoding of the red, green, and blue video signals into one signal. This encoding is also known as National Television Standards Committee (NTSC) encoding. The standard PS 390 can generate a video timing format compatible with RS-170-A. Before video signals from the PS 390 can be recorded on a video recorder, they must be encoded using a device called an NTSC Encoder. The encoded

signal cannot be hooked up to the antenna leads of a TV, but must be modulated over a standard TV broadcast frequency. Because the PS 390 does not generate an encoded signal in the RS-170-A or any other video timing format, it conforms to the timing standards of RS-170-A standard, but not the encoding standard.

The PS 390 optionally can support two alternate video timing formats. For a description of these alternate formats, multiple PS 390 video hookups, and PS 390 video specifications, refer to *Appendix A*.

Different video timing formats are used to display pictures. Every display is configured to work with a certain video timing format. The important features of a video timing format are:

- Pixel Rate
- Horizontal Frequency
- Field Rate
- Frame Rate

Pixel rate is the rate at which video information can change. The default PS 390 video timing format has a pixel rate of approximately 70 MHz.

Horizontal frequency is the number of times a horizontal line is drawn across the screen each second. The default PS 390 video timing format has a horizontal frequency of 54 KHz.

Field rate is the rate that the electron beam goes from the top to the bottom of the screen. The default PS 390 video timing format has a field rate of 60 Hz.

Frame rate is the rate at which the entire screen is redrawn. In non-interlaced displays, the frame rate is the same as the field rate.

All timing information is conveyed on the composite sync signal. The PS 390 generates only one composite sync signal, so only one video timing format can be generated at a time. A different video timing format requires a different composite sync signal.

Displays with different video timing formats can be hooked up at the same time, but only displays that are configured for the video timing format which is generated by the PS 390 at that time will produce a good picture.

2. The PS390ENV Function

The PS390ENV is an initial function instance that sets up the background color, selects a cursor and a cursor color, and selects a video timing format.

Input <1> is a trigger which accepts any data type to make the function run.

2.1 Selecting a Background Color

Input <2> of PS390ENV is a constant which accepts a 3D vector with hue, saturation, and intensity values to specify a background color used in depth cueing. The default background color is 0,0,0, which specifies black as the background color. Saturation and intensity must be in the range of [0,1] or an error message is generated. Hue is in the range of [0,360]. For any value specified outside this range, multiples of 360 are added or subtracted to bring it into the [0,360] range.

2.2 Selecting a Cursor Color

Input <3> of PS390ENV is a constant which accepts an integer in the range [0,7] to specify the color of the screen cursor. The following values select the following cursor colors:

```
0. Black

1. Blue

2. Green

3. Cyan

4. Red

5. Magenta

6. Yellow

7. White
```

White is the default cursor color. Any color outside the [0,7] range generates an error.

2.3 Selecting a Cursor

Input <4> of PS390ENV is a constant which accepts an integer to select the type of cursor. There are three types of cursors that can be selected:

```
    0. Update Rate Cursor (default)

    1. Refresh Rate Cursor

 2-32. Programmable Refresh Rate Cursors
```

The update rate cursor is the default system cursor that appears when the PS 390 is booted up. The update rate cursor is part of the vector list in the graphics data structure and the shape of the update rate cursor can be reprogrammed by changing the vector list. The update rate cursor is synchronized with the system's update rate. Sending a value of 0 to Input <4> of PS390ENV selects the system-defined update rate cursor, which is an x-shaped cursor.

When used in a static viewport, the update rate cursor is a "destructive" cursor; that is, when the update rate cursor is dragged across an object, it "destroys" the foreground pixels and leaves a path of the background color along the pixels touched by the path of the cursor. This problem does not occur in a dynamic viewport.

The system-defined refresh rate cursor is a 31 pixel by 31 pixel cross-shaped cursor that is a single pixel wide. The refresh rate cursor matches the system's refresh rate and is only supported by the default 1024 by 864 non-interlaced video timing format (video timing format 0). Sending a 1 to Input <4> of PS390ENV selects the system-defined refresh rate cursor.

The programmable refresh rate cursors are not yet available.

The refresh rate cursor is a "nondestructive" cursor that can be dragged across an object without altering pixel values in either dynamic or static viewports.

When the refresh rate cursor is selected, the initial viewports HVP1$ and GVP0$ must NOT be changed for the refresh rate cursor to work properly with picking.

2.4 Selecting a Video Timing Format

Input <5> of PS390ENV is a constant which accepts an integer to specify the video timing format for the display. The 1024 by 864 non-interlaced format is the default video timing format. The standard PS 390 monitor only supports this default video timing format. To use the alternate video timing formats, you must have an interlaced monitor physically connected to the PS 390. When sent to Input <5> of PS390ENV, the following values select the following video timing formats:

```
0. 1024 by 864 non-interlaced (default)

1. Reserved for Diagnostic Use

2. 1024 by 864 interlaced (RS-343-A)

3. 640 by 484 interlaced
```

Option 1 is reserved for diagnostic use and cannot be selected as a video timing format. Selecting option 1 generates an error message.

2.4.1 Reconfiguring Viewports for Alternate Video Timing

You MUST send commands to reconfigure your base viewport when you select the alternate 640 by 484 interlaced video timing format or switch from 640 by 484 interlaced back to one of the 1024 by 864 formats.

To reconfigure the base viewport when you go from the 640 by 484 interlaced format to the 1024 by 864 non-interlaced format, send the following command:

```
configure a;
vpf1$ := view horiz = -.84179688:.84179688 vert = -.68359375:1
inten=0:1;then HVP1$;
finish configuration;
send fix(0) to <5>ps390env;
send true to <1>ps390env;
```

To reconfigure the base viewport when you go from the 640 by 484 interlaced format to the 1024 by 864 interlaced format, send the following command:

```
configure a;
vpf1$ := view horiz = -.84179688:.84179688 vert = -.68359375:1
inten=0:1;then HVP1$;
finish configuration;
send fix(2) to <5>ps390env;
send true to <1>ps390env;
```

To configure the correct viewport for the 640 by 484 interlaced video timing format, send the following commands:

```
configure a;
vpf1$ := view horiz = -.84570313:.09570313 vert = .05859375:1
inten=0:1;then HVP1$;
finish configuration;
send fix(3) to <5>ps390env;
send true to <1>ps390env;
```

2.5 Selecting a Line Filter

The PS 390 supports four selectable line filters to determine the type of aliased or antialiased line the system will draw. The following command selects a line filter:

```
Name1 := Select Filter n THEN Name2;
```

where n = 0, 1, 2, or 3. This selects the filter applied to Name2 and subsequent structures. This command creates an operation node in the data structure.

The type of filter determines the quality of the lines displayed. Four line filters are provided:

```
0. SIN (X)/X Filter

1. Narrow Gaussian (Default)

2. Wide Gaussian

3. Jagged (No filter)
```

The SIN (X)/X filter (filter 0) produces the sharpest, best quality lines and works well with images such as text characters that require fine detail. However, the SIN (X)/X filter only works with limited background colors; it works best with light background colors, such as gray. The SIN (X)/X filter produces more artifacts than the Gaussian filters when multiple lines overlap.

The default line filter is the narrow Gaussian filter (filter 1). The narrow Gaussian filter is the best general-purpose filter and produces good quality, sharp lines. It works with any background color and works well with detailed images such as those that contain radial lines.

The wide Gaussian filter (filter 2) creates wider lines with less definition. The wide Gaussian filter produces no artifacts and works well with primitives such as dots.

The jaggy filter (filter 3) produces unfiltered, aliased lines.

Values outside the 0–3 range default to the narrow Gaussian filter (filter 1), with the following warning message:

```
W2045 ** Illegal filter selection, default filter 1 used

Inputs allowed:
Qinteger <1> selects filter. Anything other than 0,1,2,3 will
default to 1 and generate the above warning message.
```

Appendix A
Guide To PS 390 Video

This appendix explains the raster video concepts and the raster video output of the PS 390. The second section contains video specifications for the PS 390.

Video Signals

The terminal controls the display. The terminal controls how bright the electron beam is for each point of the picture and when the beam draws each scan line. The information that controls the brightness of the beam is called "active video." The information that tells the beam when to go back and forth in the horizontal direction is called "horizontal sync." The information that tells the beam to go back and forth in the vertical direction is called "vertical sync." Vertical sync and horizontal sync are often combined to form "composite sync." The information is separated in the display by special filters.

The composite sync signal does not need to convey any information while active video information is being sent. This allows the composite sync signal to be included with one of the video signals. The resulting signal is called "composite video." Most color video terminals have one video signal for each of the primary colors of light: red, green, and blue. The composite sync signal is usually included with the green video signal. The PS 390 uses this red, green, blue (RGB) video system with composite sync carried on the green video signal.

The video signal has defined voltage levels which convey information about brightness, composite sync, and more to the display. The Electronic Industries Association's RS-343-A standard is one common standard. The signal voltage levels from the PS 390 conform to this standard.

Video Timing Formats

Different video timing formats are used to display pictures. Every display is configured to work with a specific video timing format. Remember that the important quantities of a video timing format are:

- Pixel Rate
- Horizontal Frequency
- Field Rate
- Frame Rate

All timing information is conveyed on the composite sync signal. The PS 390 generates only one composite sync signal, so only one video timing format can be generated at a time. A different video timing format requires a different composite sync signal. The PS 390 supports three video timing formats that can be selected from runtime. Displays with different video timing formats can be hooked up at the same time, but only displays that are configured for the video timing format which is generated by the PS 390 at that time will have a good picture.

Multiple PS 390 Video Hookups

The red, green, and blue video signals are carried on three coaxial cables bundled together in a large shielded cable. The coax has a characteristic impedance of 75 ohms. The connectors are standard BNC. The PS 390 generates only one set of video signals, so multiple output devices such as displays and cameras must be connected in a "daisy chain." This means that the first device hooked up to the PS 390 must have both inputs and "loop-thru" outputs. The next device connects the loop-thru outputs to its inputs. Each additional device along the chain works the same way. All devices except the device at the end of the chain must have its termination removed or turned off. The last device in the chain must terminate the video signals. This is usually done with BNC caps with a built-in 75 ohm resistor. Picture quality may degrade if more than one device is connected to the PS 390. In such cases, E&S recommends the use of wide-band video distribution amplifiers with a minimum bandwidth of 70 MHz to implement active "loop-thru". When using these active "loop-thru" devices, all devices should be terminated at 75 ohms.

Video Cables

PS 390 video cables are available in lengths of 15, 25, 50, and 100 feet. Composite sync at TTL levels is also available (204584-xxx). Evans & Sutherland does not provide cables for connecting multiple video devices.

Video Options

The PS 390 supports a total of five video timing formats. The PAL/SECAM and RS-343-A 1024 by 1024 formats are not standard and require installation of a hardware option. Supported video timing formats are:

- 1024 by 864 non-interlaced (PS 390 default)
- RS-343-A 1024 by 864
- RS-343-A 1024 by 1024
- 640 by 484 interlaced (RS-170-A/NTSC timing)
- 768 by 574 interlaced (PAL/SECAM timing)

The three video timing formats that are supported by PS 390 system software are:

- 1024 by 864 non-interlaced
- 1024 by 864 (RS-343-A timing)
- 640 by 484 interlaced (RS-170-A/NTSC timing)

The combinations available as options are shown in the following table:

Format 0 (Hobson's Choice)	**Format 2** (Pick One)	**Format 3** (Pick One)
1024 by 864 (non-interlaced)	RS-343-A 1024 by 864 (interlaced)	RS-170-A (NTSC) 640 by 484 (interlaced)
	RS-343-A 1024 by 1024 (interlaced)	PAL/SECAM 768 by 574 (interlaced)

Custom Video Timing Formats

Other video timing formats may be available upon request. The PS 390 supports gen lock capability as an option. The PS 390 does not support video mixing.

References:

1. EIA RS-170 Standard
2. EIA RS-170-A Standard
3. EIA RS-343-A Standard
4. *Raster Graphics Handbook*, Conrac Division, Conrac Corporation, (New York: Van Nostrand Reinhold Company, 1985), Second Edition. Chapter 8 is especially informative.

Video Output Specifications for the PS 390

Connections

There are three BNC connectors, Red, Green, and Blue. Composite sync is on Green. Composite sync is also available at TTL levels. There must be provisions on the monitor for grounding the shield of the display cable.

Voltage Levels

Level	Value
0.000 V	Peak White
-0.071 V	Reference White
-0.714 V	Reference Black
-0.785 V	Blanking
-1.071 V	Composite Sync

Voltage levels are given referenced to "earth" ground. All video signals must be terminated in 75 ohms to ground.

Nomenclature

Front Porch refers to the time interval between the end of active video and the beginning of sync, during which the video is at Blank level. Back porch refers to the time interval between the end of sync and the beginning of active video, during which the video is at Blank level. The symbol "H," when used in the vertical timing specification, means one horizontal period.

Tables 12-1 to 12-5 define the video characteristics of the PS 390 video formats.

Table 12-1. 1024 by 864, 60 Hz, non-interlaced Video Format

Effective resolution 8192 by 6912

Aspect Ratio, H:V: 4:3.38

Horizontal timing:

Frequency:	54.06 KHz
Front Porch:	173 nsec
Sync Pulse:	1850 nsec
Back Porch:	1676 nsec
Total Blanking:	3700 nsec
Active Video:	14798 nsec
Horz Period:	18498 nsec
Pixels Displayed:	1024

Vertical timing:

Frequency:	60 Hz
Front Porch:	none
Sync Pulse:	3H
Back Porch:	34H
Total Blanking:	37H (684 usec)
Active Video:	864H (16.0 usec)
Total Vertical Time:	901H (16.67 msec)

Pixel Frequency:	69.1968 MHz
Pixel Period:	14.452 nsec

Table 12-2. RS-343 1024 by 864, 30 Hz, Interlaced Video Format

Effective resolution 8192 by 6912

Aspect Ratio, H:V:	4:3.38

Horizontal timing:

Frequency:	28.197 KHz
Front Porch:	1001 nsec
Sync Pulse:	2779 nsec
Back Porch:	3224 nsec
Total Blanking:	7004 nsec
Active Video:	28460 nsec
Horz Period:	35464 nsec
Pixels Displayed:	1024

Vertical Timing:

Frequency:	60 Hz per field
Front Porch:	3H
Sync Pulse:	3H
Back Porch:	31H
Total Blanking:	37H (1312 usec) per field
Active Video:	432H (15.32 msec) per field
Total Vertical Time:	939H (33.3 msec)

Pixel Frequency:	35.98 MHz
Pixel Period:	27.793 nsec

Table 12-3. 640 by 484, 30 Hz, Interlaced (RS-170-A (NTSC)) Video Format

Effective resolution 5120 by 3872

Aspect Ratio, H:V: 4:3

Horizontal timing:

Frequency:	15.734 KHz
Front Porch:	1638 nsec
Sync Pulse:	4914 nsec
Back Porch:	4586 nsec
Total Blanking:	11139 nsec
Active Video:	52417 nsec
Horz Period:	63556 nsec
Pixels Displayed:	640

Vertical Timing:

Frequency:	59.94 Hz per field
Front Porch:	3H
Sync Pulse:	3H
Back Porch:	14H
Total Blanking:	20H (1271 usec) per field
Active Video:	242.5H (15.41 msec) per field
Total Vertical Time:	525H (33.36 msec)

Pixel Frequency:	12.2098 MHz
Pixel Period:	81.9014 nsec

Table 12-4. 768 by 574, 25 Hz, Interlaced (PAL, SECAM) Video Format

Effective resolution 6144 by 4592	
Aspect Ratio, H:V:	4:3
Horizontal timing:	
Frequency:	15.625 KHz
Front Porch:	1627 nsec
Sync Pulse:	4610 nsec
Back Porch:	5695 nsec
Total Blanking:	11932 nsec
Active Video:	52068 nsec
Horz Period:	64000 nsec
Pixels Displayed:	768
Vertical Timing:	
Frequency:	50 Hz per field
Front Porch:	2.5H
Sync Pulse:	2.5H
Back Porch:	20H
Total Blanking:	25H (1600 usec) per field
Active Video:	287.5H (18.40 msec) per field
Total Vertical Time:	625H (40.0 msec)
Pixel Frequency:	14.7500 MHz
Pixel Period:	67.7966 nsec

Table 12-5. RS-343 1024 by 1024, 30 Hz, Interlaced Video Format

Effective resolution 8192 by 8192	
Aspect Ratio, H:V:	1:1
Horizontal timing:	
Frequency:	33.1 KHz
Front Porch:	1001 nsec
Sync Pulse:	2730 nsec
Back Porch:	3185 nsec
Total Blanking:	6916 nsec
Active Video:	23296 nsec
Horz Period:	30212 nsec
Pixels Displayed:	1024
Vertical Timing:	
Frequency:	60 Hz per field
Front Porch:	3H
Sync Pulse:	3H
Back Porch:	33H
Total Blanking:	39H (1178 usec) per field
Active Video:	512.5H (15.48 msec) per field
Total Vertical Time:	1103H (33.3 msec)
Pixel Frequency:	43.956 MHz
Pixel Period:	22.75 nsec

GT13. POLYGONAL RENDERING

GT13. POLYGONAL RENDERING

CONTENTS

ILLUSTRATIONS

Section GT13
Polygonal Rendering

Introduction

The commands and the function covered in this section allow the user to define objects eligible for rendering and to perform rendering operations on these objects. It is intended both as an introduction to rendering concepts and as a detailed statement of the rules for using a PS 390 configured with the rendering option.

Objects composed of polygons defined by the POLYGON command are the only objects that are eligible for rendering operations. Objects created by other data definition commands, such as VECTOR_LIST, CHARACTERS, LABELS, POLYNOMIAL, RATIONAL POLYNOMIAL, BSPLINE, and RATIONAL BSPLINE, are displayed along with polygonal objects prior to rendering, but are omitted from renderings. (However, special vector lists output from F:XFORMDATA can be used to render spheres and lines in a static viewport. This type of rendering operation is mainly used by molecular modelers and is described in Section *TT2 Helpful Hints.*)

Specifically, this section explains how to use the following PS 390 commands:

- POLYGON
- SOLID_RENDERING
- SURFACE_RENDERING
- SECTIONING_PLANE
- ATTRIBUTES
- ILLUMINATION

This section also discusses how to use the following PS 390 function:

- SHADINGENVIRONMENT

Objectives

This section presents the following topics and operations in the order listed below. This order is not necessarily the order in which they should be performed. After reading this section you should be able to:

- Identify the different rendering operations that can be performed in dynamic or static viewports.
- Define a polygonal object with the POLYGON command using all the command options (WITH ATTRIBUTES, WITH OUTLINE, COPLANAR, Normals, Soft Edges, Vertex Colors).
- Establish a workspace in memory.
- Mark an object as a solid or a surface for rendering.
- Render the object.
- Save and compound a rendering.
- Display a shaded object in a static viewport and change the shading environment in which the object is displayed.

For those already familiar with the PS 390, a reference summary at the end of this section lists important rules and guidelines. Also, Section *GT15 Sample Programs* contains a polygonal-rendering example that illustrates many of the rules and concepts discussed in this section.

Prerequisites

Before reading this section, you should be familiar with programming the PS 390. It is helpful to have an understanding of the representation of polygonal objects in graphics applications. It is assumed that you have some method, such as an application program, to automatically generate polygonal data structures. It is also assumed that you have some knowledge of the parameters used in rendering and shading objects for display on a raster screen.

Boot the PS 390 with the rendering firmware and run the Rendering Option Performance Verification Test. The test graphically illustrates many of the concepts discussed in this section.

1. Dynamic and Static Viewport Rendering

There are two types of rendering operations: those applied to objects displayed in a dynamic viewport and those applied to objects displayed in a static viewport. Once an object has been correctly defined with the POLYGON command, it can be displayed in either a dynamic or a static viewport without any modification to the data definition.

1.1 Dynamic Viewport Renderings

Rendering operations performed in a dynamic viewport include the following:

- Backface removal (for solid wireframe polygonal models)
- Sectioning (for both surface and solid wireframe polygonal models)
- Cross-sectioning (for solid wireframe polygonal models)

1.1.1 Backface Removal

Backface removal is an intermediate step in hidden-line removal, during which all polygons facing away from the viewer are removed. Because backface removal takes considerably less time than hidden-line removal, this operation is provided separately to allow you to see an approximation of a hidden-line rendering's appearance.

This operation is especially useful in obtaining quick previews of hidden-line renderings of complex solids when an appropriate viewing angle is being decided upon by trial and error. Because the backface removed rendering is an unfinished hidden-line rendering, it is not identical to the hidden-line rendering in every line segment, but it is close enough to give a rough idea of the hidden-line rendering.

Only solids can be subjected to backface removal; the operation has no visual effect on surfaces.

Figure 13-1 is an example of a solid before and after backface removal.

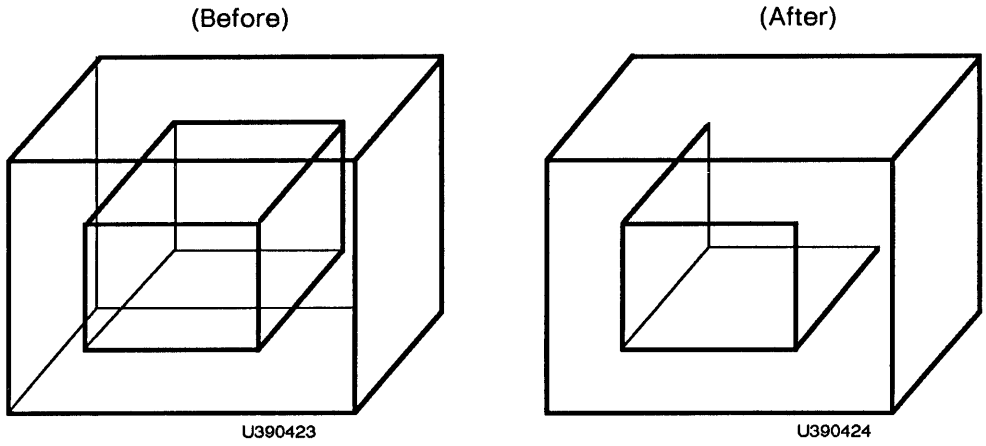


Figure 13-1. Object Before and After Backface Removal

1.1.2 Sectioning

Sectioning makes use of a sectioning plane that passes through an object and divides the object into two pieces. This operation yields a "cutaway view" of the object. The part of the object that is behind the plane is discarded and only the section in front of the plane is displayed. For solids, capping polygons are generated to maintain the integrity of the solid.

A sectioned object may be saved and then subjected to further surface rendering operations such as resectioning, hidden-line removal, or backface removal.

Although there is generally no immediate visual evidence that a capping polygon has been produced, capping polygons become a part of the definition of a sectioned solid, and further rendering can disclose their existence. For example, suppose that a solid and a surface are each sectioned vertically, yielding the two sectioned objects shown in Figure 13-2. Assume that each object intersects with its sectioning plane at its two right-most faces. It is impossible to tell which object is capped.

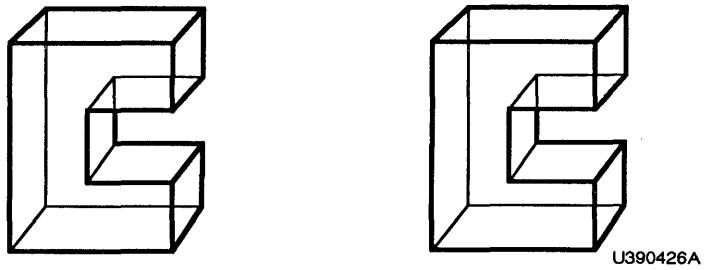


Figure 13-2. Sectioned Object With Capping Polygons

Hidden-line removal shows that the object on the left is a solid, while the object on the right is open at its right-most faces (Figure 13-3).

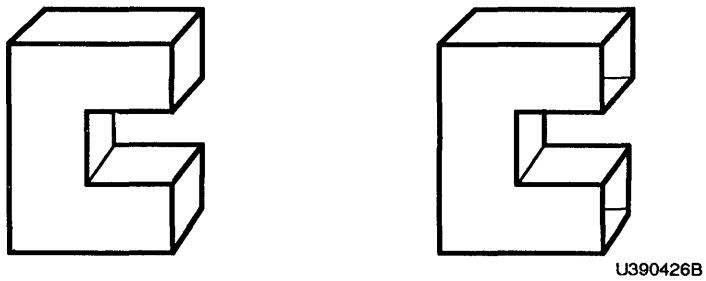


Figure 13-3. Sectioned Object With Hidden-lines Removed

Sectioning occurs within 1–3 seconds; the display may blink briefly while sectioning is applied.

1.1.3 Cross-sectioning

The cross-sectioning operation makes use of a defined sectioning plane to create a cross section of an object. When this operation is used, both sides of the object are discarded and only the slice defined by the sectioning plane remains.

1.2 Static Viewport Renderings

Rendering operations that apply to objects in a static viewport include:

- raster hidden-line removal
- wash shading
- flat shading
- Gouraud shading
- Phong shading

1.2.1 Hidden-Line Removal

Hidden-line removal generates a view in which only the unobstructed portions of an object are displayed. All polygon edges or parts of edges that would be obscured by other polygons are removed (Figure 13-4).

Three steps are involved in hidden-line removal.

1. Backfacing polygons are discarded or made front facing. This happens within 1–3 seconds. During this time the screen is blank.
2. The remaining polygons are sorted by their Z coordinates. This step takes approximately 30 seconds for 3,000 polygons, during which time the intermediate backface-rernoved picture is created. The time required for sorting depends on the number of polygons and the order in which they are defined.
3. A static raster rendering is produced with the polygon outlines turned on. The polygon interiors are colored black, but they still have Z values which render into the scanline z-buffer thus resolving obscurities. This algorithm handles convex polygons and interpenetrating polygons.

The last step may take one minute or more, depending on the number of polygons and the view. In general, it takes more time to process polygons that cover a large area of the screen than it does to process those that cover a small area.

Hidden-line removal may be performed on both solids and surfaces. Hidden-line views cannot be subjected to further rendering operations.

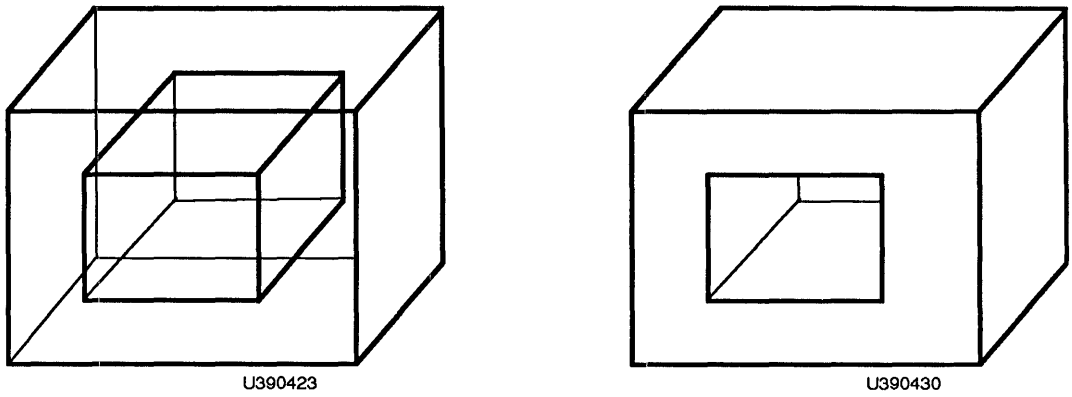


Figure 13-4. Solid Before and After Hidden-line Removal

1.2.2 Wash Shading

Wash shading produces an object with area-filled colored polygons ignoring normals, light sources, all lighting parameters, and all depth cueing parameters. This operation does not produce objects that simulate a curved surface.

1.2.3 Flat Shading

Flat shading considers color, multiple light sources and depth cueing to shade the polygons in the object accordingly. Flat shading produces objects that simulate a faceted surface.

1.2.4 Gouraud Shading

Gouraud shading is a smooth shading style. This shading process eliminates much of the faceted appearance of flat shading. The color of a polygon is varied across its surface, considering the normals at the vertices of the polygon, the direction and color of various active light sources, the attributes of the polygon (both color and highlights), and depth cueing.

In Gouraud shading the degree of light which is transmitted is derived by first calculating the degree of light transmitted at the vertices using the normal you supply and then interpolating between the vertices.

1.2.5 Phong Shading

Phong shading is also a smooth shading style. Phong shading processes are the most complex of all the shading styles. The color of a polygon is varied

across its surface, using the surface normal derived by interpolating the normals supplied at each vertex of the polygon. The direction and color of various active light sources, the attributes of the polygon (both color and highlights), and depth cueing are also incorporated to achieve the final result. Phong shading is the slowest shading style to apply, but it results in a smooth appearance of higher quality than Gouraud shading.

2. Defining Polygonal Objects

The first step in defining a polygonal object is to determine the correct geometry to define that object in the world coordinate space. This is done typically by an application program because determining the vertices of all the polygons of an object is too complex to do manually. The polygons that make up an object must be defined in the POLYGON command according to certain rules. If these rules are not followed, the results of a rendering operation applied to that object are unpredictable and usually incorrect even though the object may appear correct when displayed in the dynamic viewport.

After an object is correctly defined with the POLYGON command, it can be displayed in either a dynamic or a static viewport. The operations that can be applied in each type of viewport were discussed in Section 1.1 and Section 1.2.

2.1 Using the POLYGON Command

A POLYGON clause, part of the POLYGON command, defines an individual polygon or face of an object by specifying the coordinates of its vertices. Since an object has many faces, several POLYGON clauses are used to define the entire object. The number of POLYGON clauses in the POLYGON command is equal to the number of polygons in the object.

The edges of the polygon are defined by lines that connect the polygon's vertices. It is important that the vertices be connected in a clockwise order when being viewed from the outside of the model. If they are not connected in this manner, the polygon appears in a surface rendering but not in a solid rendering. Also, backface removal is impossible, and there are problems defining holes, concavities, and/or capping polygons in sectioned renderings.

In the PS 390, a polygon must have at least three vertices and no more than 250, all of which should lie in the same plane.

Concave polygons are acceptable. Degenerate polygons (less than three vertices) are not acceptable. Polygons are not pickable and polygon data nodes have no inputs to allow them to be modified by function networks.

The syntax for the POLYGON clause is the word POLYGON and a set of X,Y,Z coordinates. Normal and vertex color specifications may or may not be present. A named group of one or more polygon clauses, with a semicolon at the end, constitutes a POLYGON data definition command (or POLYGON command for short). This command defines the data node in the data structure of that object. There is no syntactical limit on the number of polygon clauses in the group.

An option of the POLYGON command declares polygons to be coplanar, indicating that the polygons have the same plane equation. This provides the capability to create objects with holes by defining a coplanar polygon (with vertices in a counter-clockwise orientation) inside the clockwise ordered vertices of the outer polygon. Another option allows you to define the color of the edges of polygons in static raster renderings.

There are additional POLYGON command options that associate characteristics or attributes with polygons for use in creating shaded images in a static viewport. These options include color and the concentration of specular highlights. Normals can be specified for the vertices of an object to create a smoothly shaded image that simulates a curved surface. Vertex colors can be defined for each vertex and are rendered by linearly interpolating their red-green-blue (RGB) color values. These options are shown below and explained briefly; complete details are discussed throughout this section.

The POLYGON command is:

```
Name := <polygon> <polygon> . . . <polygon> ;
```

where each polygon has the definition:

```
<polygon> [WITH ATTRIBUTES name1] [WITH OUTLINE h] [COPLANAR]
POLYGON <vertex> ... <vertex>
```

and each vertex definition has the form:

```
[S] x,y,z [N x,y,z][C h[,s[i]]]
```

The following list is a brief explanation of each parameter in the command and in the vertex definition that a command contains:

- **WITH ATTRIBUTES** is an option that assigns the attributes defined by **name1** for all polygons until superseded by another WITH ATTRIBUTES clause. This option is fully discussed in Section 6, Displaying Shaded Images.
- **WITH OUTLINE** is an option that specifies as a real number (h) the color of the outline to be drawn around polygon borders in enhanced-edge shaded images, or the color of polygon edges in hidden-line renderings. It has no effect on the color of polygon edges in the dynamic viewport.
- **COPLANAR** declares that the specified polygon and the one immediately preceding it have the same plane equation.
- **S** indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon. If the **S** specifier is used for the first vertex in a polygon definition, the edge connecting the last vertex with the first is soft.
- **N** indicates a normal to the surface with each vertex of the polygon. Normals are used only in smooth shaded renderings. Normals must be specified for all vertices of a polygon or for none of them. If normals are not specified for a polygon, their values default to the values for the normal to the plane in which the polygon lies.
- **x**, **y**, and **z** are coordinates in a left-handed Cartesian system.
- **C** indicates a color that is assigned to the vertex. During shading operations, this color is interpolated across the polygon to the other vertices.
- **h,s,i** are values in the hue-saturation-intensity color system. Together, these values create a color.

2.2 Constructing Surfaces and Solids

The PS 390 command language allows you to define two classes of polygons: surfaces and solids. Solids enclose a volume of space, while surfaces do not.

Surfaces can have edges that belong to just one polygon. For example, in Figure 13-5, edge CD is a part of polygon 3 but not of any other polygon.

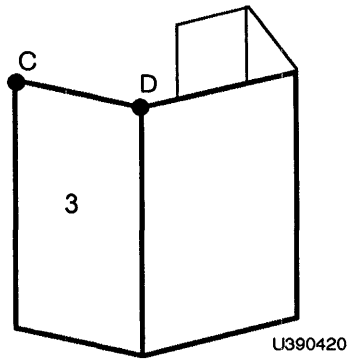


Figure 13-5. Surface Object

In a solid, each edge of each polygon must coincide with the edge of an adjacent polygon. For example, edge AB in Figure 13-6 is defined as part of polygon 1 and as part of polygon 2, and each edge of each polygon is similarly repeated in different polygons.

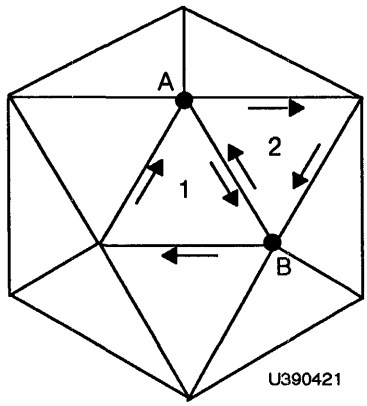


Figure 13-6. Solid Object

A solid cannot contain three or more polygons which have a single edge in common, although surfaces like the one in Figure 13-7 can.

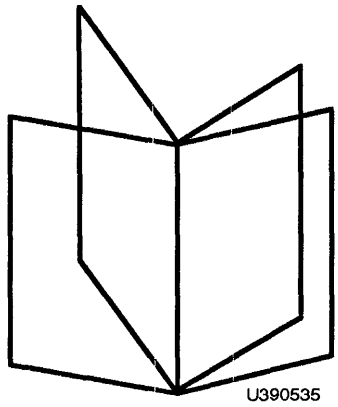


Figure 13-7. Surface With Three Common Edges

The nature of a polygonal object, representing a surface or a solid, is determined not only by the construction but by placing it beneath a rendering node in the PS 390 data structure created by the SOLID_RENDERING and SURFACE_RENDERING commands. These commands are discussed in detail in Section 4, Marking an Object for Rendering.

2.3 Specifying Vertices for Surfaces or Solids

Polygons are closed implicitly, so the first vertex is not repeated when defining a polygon. The system connects the last vertex given to the first vertex.

In solids, the direction in which the vertices are ordered within each polygon clause has important consequences for rendering operations. The vertices should be listed so that if you start at any vertex and move to the next vertex (as indicated by the order in the polygon clause), you are traveling around the edges of the polygon in a clockwise direction, as seen from outside the subject.

There are no similar restrictions for surfaces. The vertices of a surface can be listed in either a clockwise or a counterclockwise direction.

For example, let A (0,0,0), B (.5,.87,0) and C (1,0,0) be the vertices of one triangular face of a solid icosahedron as shown in Figure 13-8.

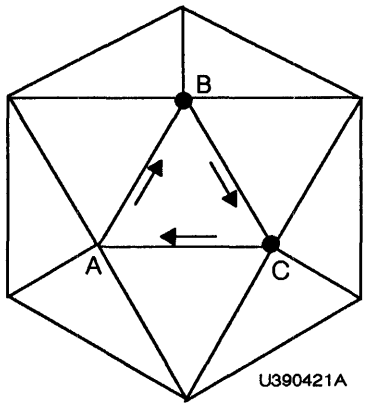


Figure 13-8. Icosahedron With Correct Vertex Ordering

Because the points A, B, and C have the arrangement indicated by the arrows when the triangular face is viewed from the outside of the icosahedron, that triangle could be defined correctly by any one of the following clauses, all of which specify the vertices in clockwise order:

```
... POLYGON  0,0,0     .5,.87,0   1,0,0 ...
... POLYGON  .5,.87,0  1,0,0      0,0,0 ...
... POLYGON  1,0,0     0,0,0      .5,.87,0 ...
```

However, the following definition is incorrect for this polygonal face because it specifies the vertices in counterclockwise order:

```
... POLYGON 0,0,0   1,0,0   .5,.87,0 ...
```

Another method to determine the order of vertices is to use the right-hand rule. The right-hand rule states that if you point the thumb of your right hand towards the center of the object and rotate your fingers towards your wrist, the direction that your fingers move indicate the order in which the vertices of that polygon should be listed.

In all correctly defined solids, each edge is repeated in two different polygons. For each pair of adjacent polygons, the common edges run in opposite directions. This is true for any edge of a correctly defined solid, even if the solid contains inner contours. For solids, all vertices must run clockwise and all common edges of adjacent polygons must run in opposite directions.

Although it is not required that surface vertices run clockwise, it is a good idea to follow the clockwise rule because it allows surfaces to be easily upgraded to solids. Assuming that polygon data are equally available in either form, it is better to have the vertices of a surface in a clockwise order.

Given the following object in Figure 13-9:

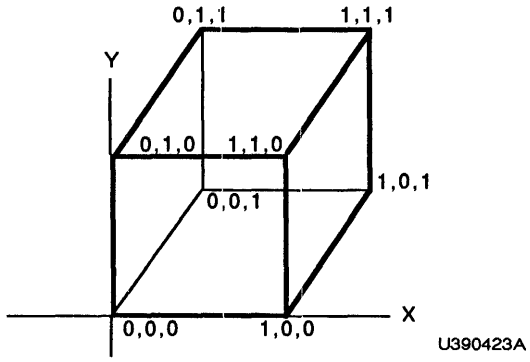


Figure 13-9. Cube

A correct syntax to define this object is as follows:

```
Cube :=   POLYGON  0,0,0   0,1,0   1,1,0   1,0,0
          POLYGON  1,0,0   1,1,0   1,1,1   1,0,1
          POLYGON  1,1,1   0,1,1   0,0,1   1,0,1
          POLYGON  0,1,1   0,1,0   0,0,0   0,0,1
          POLYGON  0,1,1   1,1,1   1,1,0   0,1,0
          POLYGON  1,0,0   1,0,1   0,0,1   0,0,0;
```

2.4 Using the COPLANAR Option

A polygon that represents a face of an object is called an outer contour. Other polygons, known as inner contours, represent cavities or holes in an object.

For the PS 390 to interpret inner contours properly, two things must be done. One is to observe the vertex ordering convention for inner and outer contours. The other is to use the COPLANAR option in the POLYGON clause to associate inner and outer contours.

The vertex ordering rule for inner and outer contours is as follows: vertices of inner contours must run in the opposite sense to the corresponding outer contour. The vertices of the following triangular polygon face (outer contour) with a hole in it (inner contour) are ordered as follows in Figure 13-10.

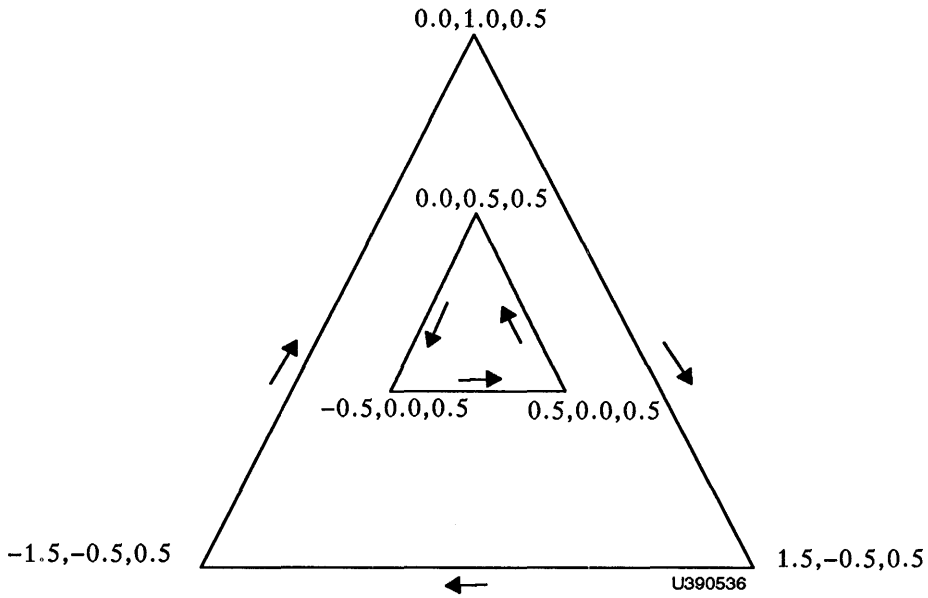


Figure 13-10. Surface With Inner/Outer Contours

A POLYGON command syntax for this object is:

```
OBJECT := INST Triangle_with_hole;
Triangle_with_hole :=
    POLYGON
      0.0,  1.0, 0.5
      1.5, -0.5, 0.5
     -1.5, -0.5, 0.5
    POLYGON COPLANAR
      0.0,  0.5, 0.5
     -0.5,  0.0, 0.5
      0.5,  0.0, 0.5
```

Note that the vertices for the inner contour in the above example are in the opposite order of those of the outer contour.

An inner contour is always coplanar with some surrounding outer contour. To define the vertices of a polygon as an inner contour, you must associate it with the appropriate outer contour by declaring an inner contour to be coplanar with the outer contour. The COPLANAR specifier makes this declaration. COPLANAR is an option of the polygon clause which declares that the specified polygon and the one immediately preceding it have the same plane equation (are in the same plane).

A polygon declared to be coplanar must lie in the same plane as the previous polygon if correct renderings are to be obtained. The system does not check for this condition. A polygon without a COPLANAR specifier immediately preceding the consecutive coplanar polygons is also taken to be in the set.

Polygons that are coplanar can be included in the polygon list without the COPLANAR specifier, as long as the polygons are not to be associated as an outer/inner pair.

If COPLANAR is specified for the first polygon in a polygon list, it has no effect.

In Figure 13-11 the second polygon is coplanar with the first polygon. The third polygon is not coplanar with either of the two preceding polygons.

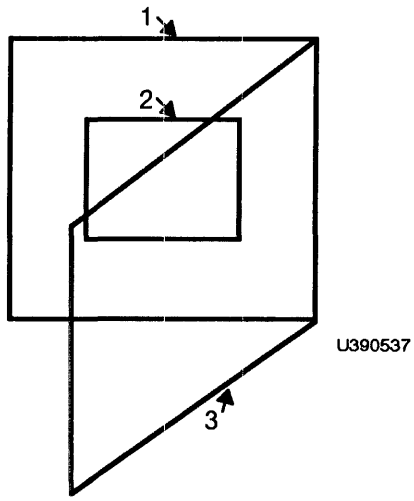


Figure 13-11. Object With Coplanar Polygon

A polygon should not be defined as an inner contour unless it is coplanar with a surrounding contour. Tunnels, protrusions, and holes do not need inner contours unless this coplanar arrangement is present. For example, in Figure 13-12 neither of the objects contains inner contours.

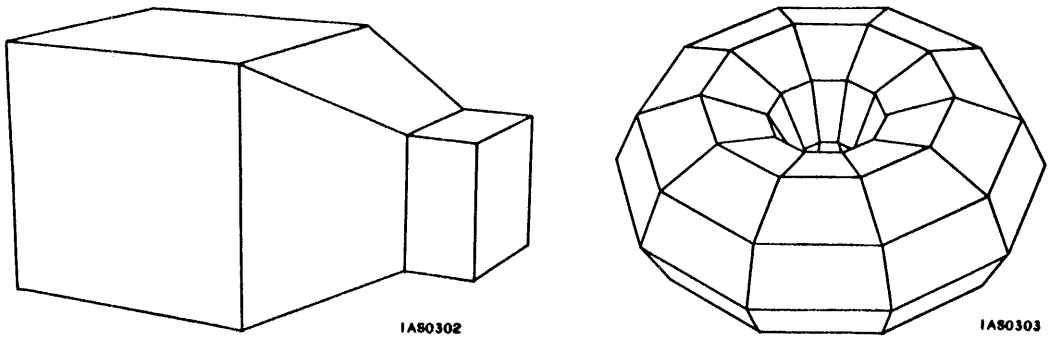


Figure 13-12. Solid Without Inner Contours

The cube with a tunnel running through it (in Figure 13-13) has two inner contours in its polygon definition, one for each opening of the tunnel.

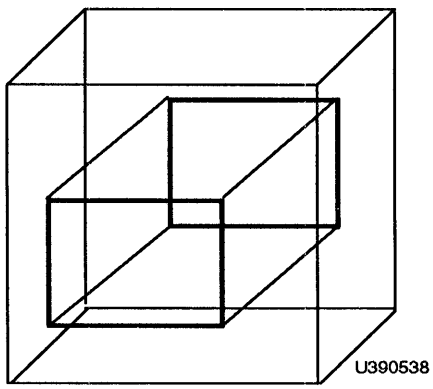


Figure 13-13. Cube With a Tunnel

A POLYGON command syntax for this object is:

```
Object :=
POLYGON -.6,-.6,.6  .6,-.6,.6  .6,.6,.6  -.6,.6,.6
POLYGON COPLANAR -.3,-.3,.6  -.3,.3,.6  .3,.3,.6  .3,-.3,.6
POLYGON -.6,-.6,-.6  .6,-.6,-.6  .6,-.6,.6  -.6,-.6,.6
POLYGON .6,-.6,-.6  .6,.6,-.6  .6,.6,.6  .6,-.6,.6
POLYGON .6,.6,-.6  -.6,.6,-.6  -.6,.6,.6  .6,.6,.6
POLYGON -.6,.6,-.6  -.6,-.6,-.6  -.6,-.6,.6  -.6,.6,.6
POLYGON .6,.6,-.6 .6,-.6,-.6 -.6,-.6,-.6 -.6,.6,-.6
POLYGON COPLANAR -.3,.3,-.6  -.3,-.3,-.6  .3,-.3,-.6  .3,.3,-.6
POLYGON -.3,-.3,-.6  -.3,-.3,.6  .3,-.3,.6  .3,-.3,-.6
POLYGON .3,.3,-.6  .3,-.3,-.6  .3,-.3,.6  .3,.3,.6
POLYGON .3,.3,-.6  .3,.3,.6  -.3,.3,.6  -.3,.3,-.6
POLYGON -.3,.3,-.6  -.3,.3,.6  -.3,-.3,.6  -.3,-.3,-.6;
```

An object with holes can often be defined in a way that does not require the inner contour. For example, the polygon with a hole (outer/inner contour pair) in this object could be replaced by four individual polygons without holes (coplanar outer contours).

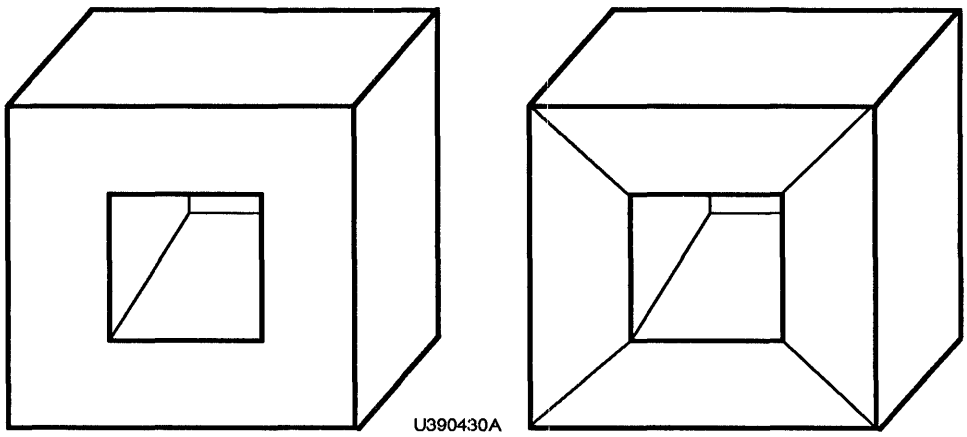


Figure 13-14. Objects With Coplanar Outer Contours

Both objects are admissible and can be rendered correctly.

In solids, misplaced capping polygons and extra missing lines are often traceable to an outer contour defined with the wrong vertex order as shown in Figure 13-15.

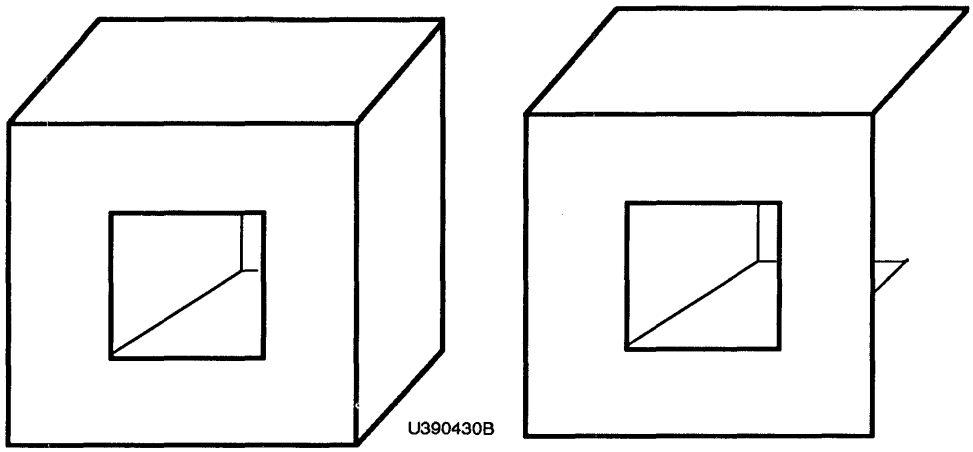


Figure 13-15. Objects With Incorrect Vertex Ordering

2.5 Using the Soft Edge Option

The S option before a set of X,Y,Z coordinates indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon. If S is placed before the first set of X,Y,Z coordinates in a polygon clause, the edge connecting the last vertex with the first is soft.

When using the S specifier in the POLYGON command to define an object, there are some rules to remember about the way the system treats edges that are declared to be soft.

An **S** specifier causes the system to apply a move operation rather than a draw operation to the associated polygon vertex. Therefore, if a single polygon containing a soft edge is displayed, the soft edge is invisible on the display.

Each polygon edge in a solid coincides with an edge of a neighboring polygon so that the solid is made up of common edge pairs. Common edge pairs can be defined as two hard edges, one hard and one soft edge, or two soft edges.

In drawing a hard common-edge pair, the same vector is drawn twice in opposite directions. If one member of the pair is soft, the vector is only drawn once. There is no variation in the intensity between a line drawn once and one drawn twice, however, high quality anti-aliasing is more precise when edges are drawn just once. When both edges of a common-edge pair are defined as soft, the common-edge pair is invisible in both the rendered view and the original object.

In the dynamic viewport, the soft edge is always displayed, with the exception of a soft common-edge pair. The same holds true of dynamic renderings, and of hidden-line renderings in the static viewport. In shaded static raster renderings, however, the default is to display neither hard nor soft edges. Input <15> of the SHADINGENVIRONMENT function controls the display of edges. As mentioned, the default (FALSE or Fix(0)) displays neither hard nor soft edges. Sending a Fix(1) to the input causes all edges to be displayed (enhanced). Sending a Fix(2) to this input allows you to enhance edges that are declared as hard, and leave edges declared as soft as undisplayed.

2.6 Defining Color For Dynamic Wireframe Polygons

The color of the edges of a polygon displayed as a dynamic wireframe model is set by using the SET COLOR command:

```
name := SET COLOR hue,sat [APPLIED TO name1];
```

where hue is a value from 0 to 360, and saturation is a value between 0 and 1. Refer to the SET COLOR command for a full description of the parameters.

To obtain pure green edges for a dynamic wireframe polygon list, the command sequence would be as follows:

```
A := SET COLOR 240,1 APPLIED TO B;
B := POLYGON...;
```

Color is specified for complete polygons, not individual edges. The default color is white. The default intensity is 1.

2.7 Using the WITH OUTLINE option to Define Color

To define the color of enhanced edges of polygon borders in static raster renderings, or the color of lines in a hidden-line rendering, the optional WITH OUTLINE clause is used. For example, the following commands can be used to define a polygon which will have green edges in the dynamic viewport, but red outlines when rendered in the static viewport.

```
A := SET COLOR 240,1 APPLIED TO B;
B := WITH OUTLINE 120
     POLYGON ...;
```

The specifier (**h**) in the WITH OUTLINE clause is an index into the Spheres and Lines Attribute Table loaded at boot time. The overlay of outlines on shaded images (polygon edge enhancement) can be turned on or off by use of the SHADINGENVIRONMENT function. Refer to the explanation for Input <14> of SHADINGENVIRONMENT for more information on the attribute table.

2.8 Defining Color and Highlights for Static Raster Renderings

Specifying the color, diffuse reflection, specular highlights, and transparency (called attributes) of surfaces and solids is done using the WITH ATTRIBUTES clause of the POLYGON command.

Given the ATTRIBUTES command syntax:

```
Name := ATTRIBUTES attributes [AND attributes];
```

where the attributes of a polygon are defined as follows:

```
[Color h[s[i]]] [Diffuse d] [Specular s] [Opaque t]
```

The ATTRIBUTES command creates a named attribute node in mass memory that defines specific qualities to be applied to polygons when referenced by the polygon data structure. The attributes specified in a WITH ATTRIBUTES clause of a polygon command apply to all subsequent polygons until superseded by another WITH ATTRIBUTES clause. If no WITH ATTRIBUTES option is given for a polygon node, default attributes are assumed.

Polygons may be solidly colored by specifying a color through the ATTRIBUTES command, or the colors may be assigned to the vertices by giving a color with each vertex specified. The color is specified by giving, first, the vertex and then, the color **(h,s,i)**. If just the hue and saturation are given, the intensity defaults to 1. If no vertex colors are given, the vertex colors default to those specified in the attribute clause.

Vertex colors must be specified for all vertices of a polygon or for none of them. However, as with normals, some polygons may have color at their vertices while other polygons may not have color at their vertices. This means that it is possible to have some objects in the picture color interpolated, while others are not.

Although color of polygon vertices is specified with a hue, saturation, and intensity component, the colors are linearly interpolated across the vertices in the red-green-blue color system. If colors are not interpolating the way you would like them to, add more vertices to the polygon, or break up large solid volumes into smaller sub-volumes and assign the desired colors to the new vertices in the object.

You can specify color for a polygon with both the ATTRIBUTES command and the color by vertex specification. An input to SHADINGENVIRONMENT allows you to switch between attribute-defined color and vertex-defined color. Input <10> of SHADINGENVIRONMENT accepts a Boolean to determine how color is specified.

The WITH ATTRIBUTES clause and the ATTRIBUTES command are explained in more detail in Section 6, Displaying Shaded Images.

2.9 Specifying Normals

When a polygon is used to approximate a curved surface, the smooth appearance of the surface can be restored in a smooth shaded rendering by approximating a surface using normals. Normals only apply to shaded renderings. Normals to a surface are specified with one normal per vertex. Like vertices, normals are defined by coordinate values of X,Y,Z. The shaded rendering process interpolates between vertex normals when rendering the polygon.

Normals must be specified for all vertices of a polygon or for none of them. If normals are not explicitly defined for a polygon, their values default to

the value of the normal to the plane in which the polygon lies. Normals are needed only in smooth-shaded renderings. If you do not define vertex normals, but you request a smooth-shaded rendering, the result is a flat-shaded rendering (except that specular and diffuse attributes will apply).

In Phong shading, the surface normal used is the one derived by interpolating the normals you supply at each vertex. In flat shading, the normal used is the vector perpendicular to the polygon. In Gouraud shading the degree of light which is transmitted is derived by first calculating the degree of light is transmitted at the vertices using the normal you supply and then interpolating between the vertices. In wash shading, no surface normal is used and no lights are used.

The following is the vector list for an octahedron with normals and attributes specified.

```
object  := INSTANCE octahedron;

blue    := ATTRIBUTES COLOR 0;
magenta := ATTRIBUTES COLOR 60;
purple  := ATTRIBUTES COLOR 90;
red     := ATTRIBUTES COLOR 120;
orange  := ATTRIBUTES COLOR 150;
yellow  := ATTRIBUTES COLOR 180;
green   := ATTRIBUTES COLOR 240;
cyan    := ATTRIBUTES COLOR 300;
white   := ATTRIBUTES COLOR 0,0,1;
black   := ATTRIBUTES COLOR 0,0,0;
grey    := ATTRIBUTES COLOR 0,0,0.5;

octahedron:=
 WITH ATTR cyan
  POLYGON
    0.000, 1.000, 0.000 N 1.000, 1.000, 1.000 C 0, 0, 1   {White}
    0.000, 0.000, 1.000 N 1.000, 1.000, 1.000 C 270, 1, 1 {Turquoise}
    1.000, 0.000, 0.000 N 1.000, 1.000, 1.000 C 0, 1, 1   {Blue}
 WITH ATTR magenta
  POLYGON
    0.000, 1.000, 0.000 N 1.000, 1.000, -1.000 C 0, 0, 1   { White}
    1.000, 0.000, 0.000 N 1.000, 1.000, -1.000 C 0, 1, 1   {Blue}
    0.000, 0.000, -1.000 N 1.000, 1.000, -1.000 C 90, 1, 1 {Purple}
```

```
WITH ATTR yellow
  POLYGON
    0.000, 1.000, 0.000 N -1.000, 1.000, -1.000 C 0, 0, 1 {White}
    0.000, 0.000, -1.000 N -1.000, 1.000, -1.000 C 90, 1, 1 {Purple}
   -1.000, 0.000, 0.000 N -1.000, 1.000, -1.000 C 180, 1, 1 {Yellow}
 WITH ATTR blue
  POLYGON
    0.000, 1.000, 0.000 N -1.000, 1.000, 1.000 C 0, 0, 1 {White}
   -1.000, 0.000, 0.000 N -1.000, 1.000, 1.000 C 180, 1, 1 {Yellow}
    0.000, 0.000, 1.000 N -1.000, 1.000, 1.000 C 270, 1, 1 {Turquoise}
 WITH ATTR red
  POLYGON
    0.000, -1.000, 0.000 N 1.000, -1.000, 1.000 C 0, 0, 0   {Black}
    1.000, 0.000, 0.000 N 1.000, -1.000, 1.000 C 0, 1, 1   {Blue}
    0.000, 0.000, 1.000 N 1.000, -1.000, 1.000 C 270, 1, 1 {Turquoise}
 WITH ATTR green
  POLYGON
    0.000, -1.000, 0.000 N 1.000, -1.000, -1.000 C 0, 0, 0  {Black}
    0.000, 0.000, -1.000 N 1.000, -1.000, -1.000 C 90, 1, 1 {Purple}
    1.000, 0.000, 0.000 N 1.000, -1.000, -1.000 C 0, 1, 1  {Blue}
 WITH ATTR purple
  POLYGON
    0.000, -1.000, 0.000 N -1.000, -1.000, -1.000 C 0, 0, 0  {Black}
   -1.000, 0.000,  0.000 N -1.000, -1.000, -1.000 C 180, 1, 1 {Yellow}
    0.000, 0.000, -1.000 N -1.000, -1.000, -1.000 C 90, 1, 1  {Purple}
 WITH ATTR orange
  POLYGON
    0.000, -1.000, 0.000 N -1.000, -1.000, 1.000 C 0, 0, 0   {Black}
    0.000, 0.000,1.000 N -1.000, -1.000, 1.000 C 270, 1, 1 {Turquoise}
   -1.000, 0.000, 0.000 N -1.000, -1.000, 1.000 C 180, 1, 1 {Yellow}
;
```

3. Establishing a Workspace in Memory

The rendering process requires that a large block of mass memory be available. This workspace is known as working storage, and once reserved, is not available for other uses until it is unreserved. The working storage requirement can be calculated automatically by the PS 390, or may be explicitly reserved.

3.1 Automatic Reservation of Working Storage

To have the system automatically reserve working storage for the rendering process, the command

```
RESERVE_WORKING_STORAGE 0;
```

should be entered. After the rendering operation is complete, the PS 390 will display the amount of memory required. This number can be written down and used to explicitly reserve memory for the next rendering operation.

The automatic calculation of working storage is more efficient in memory usage, but requires extra time during the rendering process. To avoid this, you may explicitly reserve working storage, either using the number reported by an automatic calculation, or by using the guidelines that follow.

3.2 Explicit Reservation of Working Storage

To explicitly reserve memory for a rendering, the syntax of the command is as follows:

```
RESERVE_WORKING_STORAGE n;
```

where:

the current working storage block is replaced with another containing at least **n** bytes. If n is less than or equal to 0, the system will calculate the exact memory requirement for the rendering.

The best time to reserve working storage is immediately after booting, when large requests can be filled easily.

Each polygon of a solid object with four vertices requires approximately 150 bytes of reserve working storage. Memory needs will vary from figure to figure depending on the complexity of the object, the operations to be performed on the data structure, and the view. Typically, 200,000 to 400,000 bytes of working storage should be reserved when you begin a session.

After one working storage request is made, subsequent requests do not add to the original working storage; they replace the original working storage.

Working storage is not freed by the INITIALIZE command. However, if a working storage request is followed by another, smaller request, an amount of memory equal to the difference between the two requests is freed.

If a contiguous block of memory cannot be allocated, no working storage is allocated and any previous storage is deallocated. If the system is unable to reserve enough working storage to complete a rendering, the rendering terminates prematurely and an error message is issued.

3.3 Additional Memory Requirements

In addition to the working storage space, extra mass memory is needed to create static renderings. This memory is referred to as transient memory and is automatically allocated and deallocated by the system. If adequate mass memory is not available for transient storage, the static process terminates prematurely, and an error message is generated. For this reason E&S recommends 4Mb or more of memory for renderings of objects with numerous polygons.

4. Marking an Object for Rendering

A polygonal object must be defined by a PS 390 command to be either a surface or a solid before rendering operations can be applied to it.

The commands to mark a polygonal object as a surface or solid are:

- SOLID_RENDERING
- SURFACE_RENDERING

The SOLID_RENDERING command creates an operation node in the data structure. The default value of this command declares that all of its descendant polygon data nodes define solids.

The SURFACE_RENDERING command also creates an operation node in the data structure. The default value of this command declares that all of its descendant polygon data nodes define surfaces.

The nodes established by the commands are called rendering operation nodes. Rendering nodes should not be instanced more than once either directly or indirectly, as only one node at a time may be triggered.

Before you can render an object, its rendering node must be part of a structure which is displayed (using the DISPLAY command). If the object itself is displayed but its rendering node is not, no renderings can be created.

For example, if the command sequence

```
A := SOLID_RENDERING APPLIED TO B;
B := POLYGON ... ... ... ... ;
```

has been entered, the DISPLAY command should be DISPLAY A; and not DISPLAY B;.

Syntax for the rendering commands is:

```
Name := SOLID_RENDERING APPLIED TO Name1;
Name := SURFACE_RENDERING APPLIED TO Name1;
```

where Name1 names either (a) a POLYGON node, or (b) an ancestor of one or more POLYGON nodes. If (b) is the case, any rendering referring to Name1 is performed immediately on all of the POLYGON objects descended from Name1.

A descendent polygonal object originally declared as a surface with the SURFACE_RENDERING command can be changed to a solid by sending to input <2> of the SURFACE or SOLID_RENDERING node. A TRUE sent to input <2>declares the descendent object as a solid; a FALSE sent to this input declares the object as a surface. This input is useful for updating objects originally defined as surfaces to solids, making the full range of rendering operations possible.

4.1 Admissible Descendants for Rendering Operation Nodes

The following commands may be placed between a rendering node and its data:

- IF and SET CONDITIONAL BIT
- IF and SET LEVEL_OF_DETAIL
- INCREMENT LEVEL_OF_DETAIL
- DECREMENT LEVEL_OF_DETAIL
- IF PHASE
- SET COLOR
- SET RATE
- SET RATE EXTERNAL
- SET DEPTH_CLIPPING
- BEGIN_STRUCTURE...
- END_STRUCTURE

A rendering takes into account any effects of these nodes at the time the request is made. For example, if IF PHASE and SET RATE are being used to blink an object and that object is "off" at the moment the request is made, the object is excluded from the rendering.

The nodes mentioned above can also be placed above the rendering node with the same result. Placement of these nodes above the rendering nodes is generally regarded as good programming practice, although it is admissible to put them between the rendering node and its data.

Transformations created with the following commands may be placed between a rendering node and its data node(s):

- ROTATE
- TRANSLATE
- SCALE
- MATRIX_2X2
- MATRIX_3X3
- MATRIX4X3
- LOOK

The transformation nodes should be used with caution: like the operation nodes mentioned above, their effects are incorporated into renderings, and which may result in imprecision. Another potential problem with interposing these transformations between a rendering node and the data arises when renderings are being saved.

Since most vertices in an object usually belong to more than one polygon, each vertex should be defined with the same numerical value in each of its polygons; otherwise, precision discrepancies may cause inaccurate renderings.

In general, nodes created with the following five commands should not be made descendants of a rendering node:

- WINDOW
- VIEWPORT
- EYE BACK
- FIELD_OF_VIEW
- MATRIX_4X4

Like other transformations, the five node types listed above are incorporated into the output data from a rendering operation. However, rendered data is generally displayed within a framework that already includes global 4x4 matrix transformations. Including the transformations listed above as part of the rendering usually has the effect of applying an unwanted double (double VIEWPORT, double WINDOW, etc.) to the rendered object.

SOLID_RENDERING and SURFACE_RENDERING may not be descendants of a SURFACE or SOLID_RENDERING node. If this rule is not observed, bad renderings or a system crash may result. The system does not check for this condition.

4.2 Creating Renderings

An appropriate integer sent to a SOLID_RENDERING or SURFACE_RENDERING node produces a rendering of that node's descendant polygonal object. When a rendering is first created for an object, a second set of data is created and "grafted" just below the rendering node for the original object. To display the rendering, the Joint Control Processor traverses the path to this new data. The original data remain intact and are accessible through input to the rendering node (Figure 13-16).

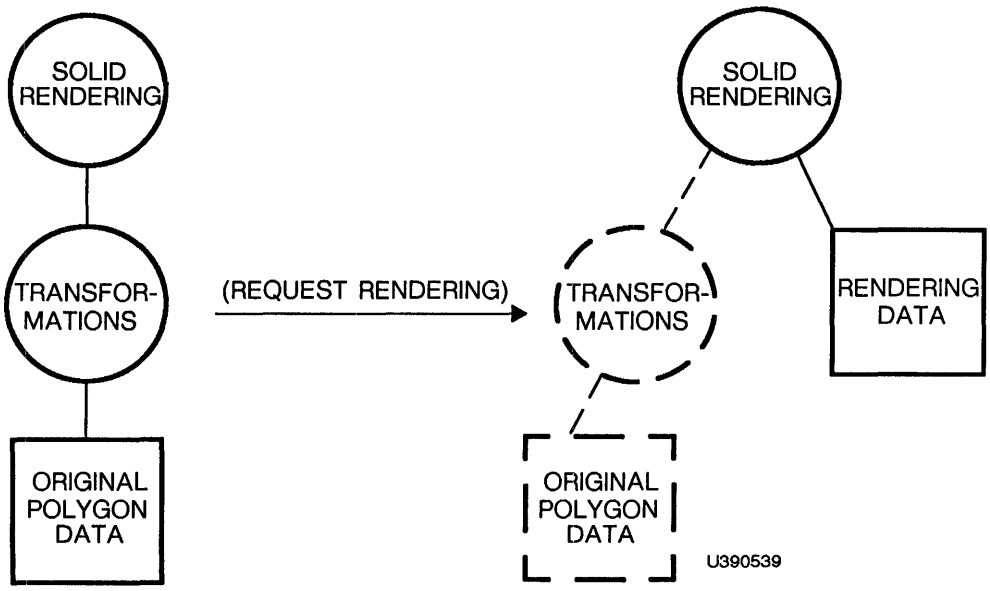


Figure 13-16. Path to Rendering Data

When the original object is redisplayed, the path to the original object is traversed and the rendering data remains intact (Figure 13-17).

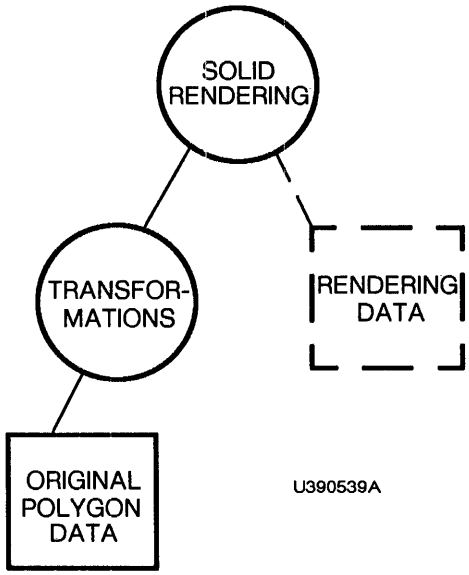


Figure 13-17. Path to Original Data

At this point, the rendering can easily be displayed again, since its data still exists. When a second rendering is done on this object, a second set of rendering data replaces the first set (Figure 13-18).

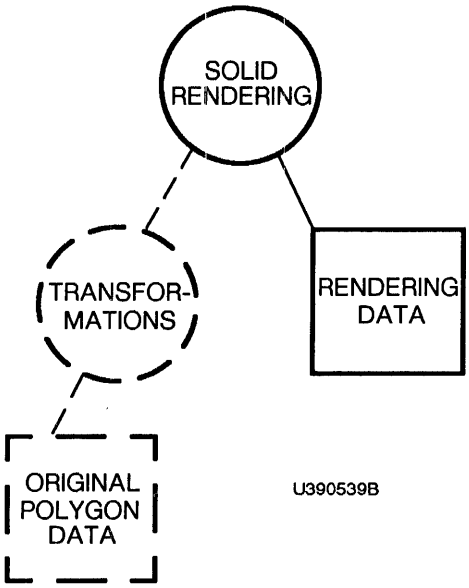


Figure 13-18. Path to Second Rendering

The "current rendering" is always the one most recently created, even if it is not currently displayed. Each rendering node has its own current rendering.

After requesting a rendering operation, commands may be entered through the keyboard or through a function network. These commands are processed at a slightly slower rate because of the overhead caused by the rendering operation.

If a command causing the screen viewport to change is issued during a rendering, the rendering will occur in the newly selected area of the screen viewport without clearing the screen.

4.3 Rendering Node Connections

Rendering nodes have five inputs. Inputs are similar for SOLID_rendering and SURFACE_rendering (Figure 13-19).

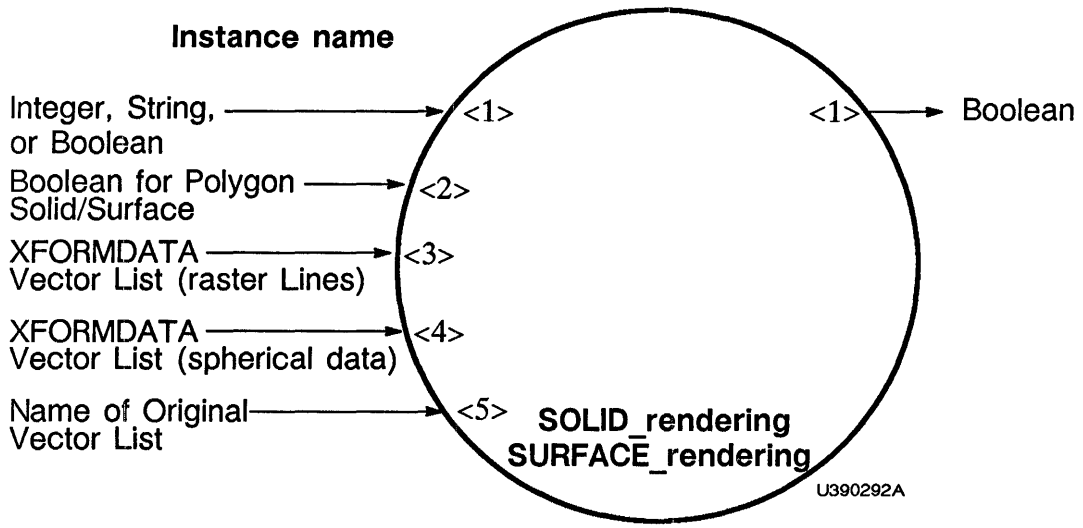


Figure 13-19. Rendering Node Connections

4.3.1 Input <1>

Input <1> of the rendering nodes accept 9 different numerical values, or a string, or a Boolean. The following summarizes valid inputs for input <1>.

Sending fix(0) to input <1> establishes a toggle between a current rendering and the original object in a dynamic viewport.

Sending fix(1) to input <1> creates and displays a cross-section of a solid object as defined by a sectioning plane in a dynamic viewport.

Sending fix(2) to input <1> creates and displays a sectioned rendering in a dynamic viewport.

Sending fix(3) to input <1> creates and displays a rendering of a solid with backfaces removed in a dynamic viewport.

Sending fix(4) to input <1> creates and displays a rendering with hidden-lines removed in the static viewport.

Sending fix(5) generates an object with the wash shading style in a static viewport.

Sending fix(6) generates an object with the flat shading style in a static viewport.

Sending fix(7) generates an object with the Phong shading style in a static viewport.

Sending fix(8) generates an object with the Gouraud shading style in a static viewport.

Sending a string to input <1> causes the current rendering to be saved under a name defined by the string.

Sending the Boolean FALSE to input <1> causes the original unrendered descendent structure of the rendering operation node to be displayed.

Sending the Boolean TRUE to input <1> causes the rendered view of the rendering operation node to be displayed.

4.3.2 Input <2>

Input <2> of the rendering nodes accept a Boolean which defines the displayed object as a surface or a solid.

Sending a TRUE to input <2> defines the descendent object as a solid.

Sending a FALSE to input <2> defines the descendent object as a surface.

4.3.3 Input <3> Through Input <5>

Input <3> accepts a transformed vector list from the function F:XFORMDATA to define raster lines. (Used with CPK–Molecular Modeling.)

Input <4> accepts a transformed vector list from the function F:XFORMDATA to define spherical centers. (Used with CPK–Molecular Modeling.)

Input <5> accepts the original vector list to enable accurate spherical scaling. (Used with CPK–Molecular Modeling.)

4.3.4 Output <1>

Rendering nodes, unlike most other display structure nodes, generates an output. A TRUE is output upon successful completion of the rendering process, and a FALSE is output if the rendering failed.

For example, the commands

```
A := SOLID RENDERING APPLIED TO B;
CONNECT A<1>:<1>C;
```

cause the output of a rendering node to be sent to input <1> of C.

Any input to input <1> of a rendering node causes an output. If output <1> has not been connected, and an integer, string, or Boolean value is sent to input <1>, a system generated function network will cause a message to appear on the screen upon successful completion of the rendering operation. An error message also appears if the rendering was not completed.

4.4 Establishing a Sectioning Plane

Defining, displaying, and positioning a sectioning plane are the first steps in producing a sectioned rendering of an object. The SECTIONING_PLANE command creates a sectioning plane node which indicates that a descendant POLYGON is a sectioning plane. The syntax is:

```
Name := SECTIONING_PLANE APPLIED TO Name1;
```

where Name1 names either (a) a POLYGON command or (b) an ancestor of a POLYGON command.

4.5 The Data Definition of a Sectioning Plane

A sectioning plane is the plane in which a specified polygon lies. Only the plane need intersect the object to be sectioned; the actual polygon that defines the plane does not need to.

The data which defines a sectioning plane is contained in a POLYGON node; SECTIONING_PLANE indicates that a given POLYGON node represents a sectioning plane.

The sectioning plane is the plane containing the polygon defined by the first POLYGON clause of the first polygon node encountered after a sectioning plane node. Additional polygon clauses defining other polygons have no effect on actual sectioning operations, but are displayed along with the defining sectioning plane polygon. This can be put to use in designing an indicator which shows the side of the plane at which sectioning will remove (or preserve) polygon data. For example, the command

```
SPdata :=
POLYGON -.9,-.9,0  -.9,.9,0   .9,.9,0   .9,-.9,0
POLYGON  .1,0,0     .1,0,-.3  .15,0,-.3  0,0,-.45
        -.15,0,-.3 -.1,0,-.3 -.1,0,0
POLYGON   0,.1,0    0,.1,-.3   0,.15,-.3 0,0,-.45
          0,-.15,-.3  0,-.1,-.3 0,-.1,  0 ;
```

defines a sectioning plane with two polygonal arrow indicators as shown in Figure 13-20.

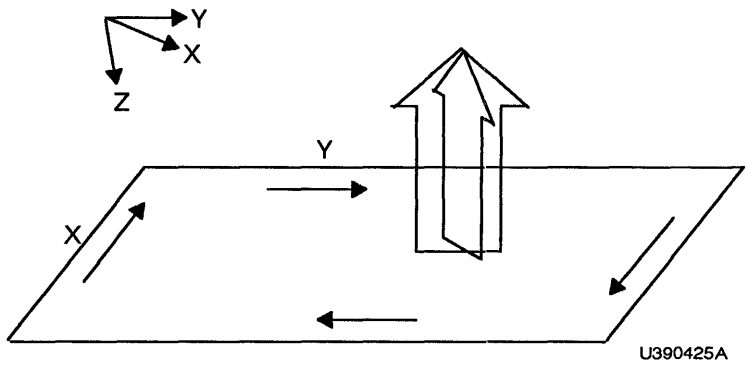


Figure 13-20. Sectioning Plane Definition

Sectioning preserves those parts of an object lying in front of and removes those parts lying in back of the sectioning plane. The front side of a sectioning plane is the side on which the vertices of the defining polygon run clockwise.

No SOLID_RENDERING or SURFACE_RENDERING operation node may be an ancestor of a sectioning plane's defining polygon. The PS 390 interprets polygons with SOLID_RENDERING or SURFACE_RENDERING ancestors as objects to be rendered rather than as sectioning plane definitions, and issues a "sectioning plane not found" message when a sectioning attempt is made.

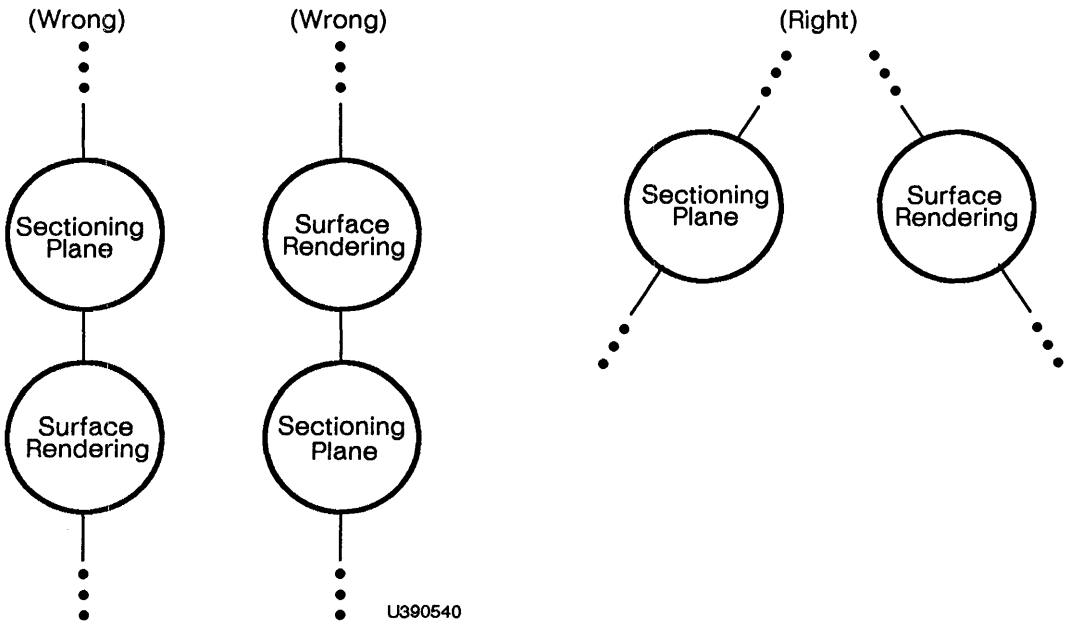


Figure 13-21. Data Structure of Sectioning Plane

Other nodes which do not represent viewing transformations, such as TRANSLATE, may be placed either above or below the sectioning plane node as needed.

Typically, you will want to orient the plane interactively by connecting interactive devices via function networks to translate and rotate the sectioning plane.

4.6 Displaying Sectioning Plane Nodes

Before an object can be sectioned, the sectioning plane node must be part of a structure which is displayed. In order to section an object, you must first create the sectioning plane, and then display it along with the object to be sectioned.

For example, if the command sequence

```
A := SECTIONING_PLANE APPLIED TO B;
B := POLYGON ... ;
```

has been entered, the DISPLAY command should be DISPLAY A; and not DISPLAY B;.

4.7 Cross-sectioning

Cross-sections can only be created for objects defined as solids. The cross-sectioning operation makes use of a sectioning plane to create a cross section of the object. When this operation is used, both sides of the object are thrown away and only the slice of the object defined by the sectioning plane remains. Essentially, the object is sectioned and only the capping polygons remain.

This operation proceeds within 1–3 seconds. During this time the display blanks momentarily while the object is sectioned.

Sending fix(1) to input <1> of the rendering node creates a cross section in the working storage area, and displays it on the screen in the dynamic viewport.

4.8 Toggling Between the Rendered Object and the Original Object

It is often useful to compare objects before and after rendering operations have been applied. The toggle operation that occurs when you send a TRUE or a fix(0) to input <1> of the rendering node allows you to alternate the display between the rendered object and the original object. Both the rendering and the original object are left intact and can be redisplayed until they are overwritten or saved.

Sending a FALSE to input <1> of a rendering operation node causes the original descendent structure of the node to be displayed. The rendered view is not affected, other than being removed from the display. The rendered view can be restored and displayed again by sending TRUE or fix(0) to the rendering operation node.

4.9 Changing the Definition of the Object

Sending a Boolean value to input <2> of the rendering node controls whether the descendant polygons are to be treated as a solid or a surface. This allows a solid rendering node to be converted to a surface rendering node and vice versa. TRUE sent to input <2> defines the node as a solid rendering node whatever the original state was. FALSE defines the node as a surface rendering node. The default is determined by the word SOLID or SURFACE in the command that created the node.

5. Saving and Compounding Renderings

Saving a rendering is done by giving it a name by which it can be referenced. Dynamic viewport renderings are the only renderings which may be saved and compounded. Static viewport renderings cannot be saved, and cannot have further rendering operations performed on them.

This process establishes a rendering as a separate named data node. Requesting and displaying a rendering creates rendering data, but does not create a node in the normal sense. The rendering cannot be referenced nor subjected to further rendering operations until it is saved. Saving the rendering is, therefore, a prerequisite to making further renderings of the object. After a rendering is saved, it is no longer considered a current rendering. Therefore, the toggle operation (Boolean values and a fix(0) sent to the rendering node) no longer affect the rendering.

5.1 How to Save a Rendering

To save a rendering, send the name (a string message) to input <1> of the SOLID_RENDERING or SURFACE_RENDERING operation node. All illegal PS 390 names are rejected and an error message is generated.

The string should specify the name of the node which is to contain the saved rendering data. If the named node does not exist, it is created; if it does exist, the saved rendering data replace the original contents of the node.

All polygons in the rendering are taken into account in the saved rendering. It is not possible to exclude selected polygons or polygon data nodes from saved renderings.

5.2 Contents of a Saved Rendering

Backface removal and sectioned renderings are saved as polygon lists. When a sectioned rendering is saved, all transformations between the rendering operation node and the polygon data node are applied to the polygon data. The result is stored in the new data node. When a backface rendering is saved, all ancestor transformations of the polygon data node are applied to the polygon data before the result is stored in the new node. This occurs even if those transformations are above the rendering operation node.

5.3 Common Uses of Saved Renderings

The most common reason for saving a rendering is to create a compound rendering from it.

Common types of compound renderings are: (a) re-sectioning of a sectioned rendering and (b) creating a static rendering of an object which has been sectioned.

Backface renderings, which are useful mainly for previewing time-consuming hidden-line operations on complex objects, are not generally rendered further. Hidden-line renderings cannot be rendered further because they are static raster images.

6. Displaying Shaded Images

The PS 390 can be used as an "image buffer" to display host generated images (by using run length encoding), or it can display shaded images derived locally from PS 390 polygonal models.

When using the display as an image or frame buffer, the PS 390 is only used as a communications link between the host and the raster system. No standard PS 390 commands or data structures are used to display host generated images.

This section deals only with displaying shaded images derived locally from PS 390 polygonal models. Run length encoding, the process of displaying host generated images, is documented in Section *GT14 Raster Programming*.

6.1 Specifying Attributes

In Section 2, Defining Polygonal Objects, you were introduced to the WITH ATTRIBUTES option. Attributes are applied to a collection of polygons by specifying the name of the attribute node after WITH ATTRIBUTES in the POLYGON command. If the WITH ATTRIBUTES option is not used in the polygon clause, the default attributes 0,0,1 for COLOR, 0.75 for DIFFUSE, 4 for SPECULAR, and 1 for OPAQUE (transparency) are assumed. Refer to Sections 6.2.1 through 6.2.4 for details on specific attributes.

6.2 Using the ATTRIBUTES Command

The ATTRIBUTES command specifies the various characteristics of polygons used in the creation of shaded renderings. Attribute nodes are created with the ATTRIBUTES command and exist in mass memory but are not part of a data structure. The attributes specified in an ATTRIBUTES command are assigned to polygons which include a WITH ATTRIBUTES clause.

When the display processor traverses the data structure with a polygon node containing a WITH ATTRIBUTES Name1, the attributes in Name1 are assigned to all polygons in the node until superseded with another WITH ATTRIBUTES clause. The various attributes may be changed from a function network via inputs to an attribute node or by reassigning the name, but the changes have no affect until a new rendering is created. No type checking is done by the shading process to ensure that WITH ATTRIBUTES

indeed refers to an attribute node and not some other entity. If it does refer to some other entity, the display processor will interpret any values in that node as attributes, and display the object incorrectly.

The ATTRIBUTES command is:

```
Name := ATTRIBUTES <attr> [ AND <attr> ] ;
```

Given:

```
<attr> = [ COLOR h [ ,s [ ,i ]]]
         [ DIFFUSE d]
         [ SPECULAR s]
         [ OPAQUE t]
```

A second set of attributes may be given after the word AND in the ATTRIBUTES command which apply to the obverse side of the polygon(s) concerned. In other words, the two sides of an object may have different attributes. The polygons considered on the obverse (back facing) side by the system are those having counterclockwise ordered vertices for the view in which the rendering is carried out. The second set of attributes will only be applied in surface renderings (not solid). The attributes defined for the first <attr> specify attributes for front facing polygons. The attributes after the AND specify the attributes of back facing polygons.

You are not required to include the AND <attr> to specify different attributes for back facing polygons. If the AND specifier is not included, the backfacing polygons will have the same attributes as the front. The command syntax for specifying just one set of attributes is:

```
Name := ATTRIBUTES <attr> ;
```

6.2.1 COLOR Component

The COLOR component of the ATTRIBUTE command sets the basic color for the surface of a polygon. This component pertains only to shaded renderings in a static viewport. It has no effect on the color of the edges of a polygon in a dynamic viewport or the outlines of polygons in static renderings (these are changed using SET COLOR command and the WITH OUTLINE clause respectively). Color is given as hue (h), saturation (s), and intensity (i). It also changes according to such variables as shading style, light sources, orientation, depth cueing, ambient lighting, and specular lights.

Hue specifies degrees around the color circle with 0 being pure blue, 120 pure red, and 240 pure green. Saturation varies from 0 for no color saturation (grays) to 1 for full color saturation. Intensity varies from 0 for no intensity (black) to 1 for full intensity.

If COLOR is not specified, the default hue is white (s=0, i=1). If not specified, saturation and intensity default to 1. If only hue and saturation are specified, intensity defaults to 1. Values greater than 1 or less than 0 for saturation or intensity are rounded to 1 or 0. Hue and saturation correspond to hue and saturation in the SET COLOR command but have greater precision.

6.2.2 DIFFUSE Component

The diffuse component of the ATTRIBUTE command determines the proportion of color contributed by diffuse reflection versus that contributed by specular reflection to smooth shaded renderings. Decreasing d (diffuseness) makes the surface more shiny; increasing d reduces the intensity of specular highlights, making the surface more matte, with a value of 1 eliminating specular highlights entirely. Values larger than 1 or less than 0 are rounded to 1 or 0. If no DIFFUSE component is given, it defaults to 0.75.

6.2.3 SPECULAR Component

The SPECULAR component of the ATTRIBUTE command adjusts the concentration of specular highlights in smooth shaded renderings. Specular highlighting is a property of the object such that the size of the highlight spot is not influenced by the light source, only by the s value. Higher concentrations of specular highlights result in more metallic looking objects. In reality, objects are never completely specular (or completely diffuse), so you get artificial effects if these values are at a maximum. Acceptable values of s are integers between 0 and 10, with values outside that rounded to 0 or 10. The default is 4.

6.2.4 OPAQUE Component

The OPAQUE component of the ATTRIBUTE command specifies the transparency of a polygon. Increasing values of **t** (transparency) increase the polygon's opacity. Acceptable values for **t** are real numbers in the range from 0 to 1 where 1 indicates that the polygon is fully opaque and 0 indicates the polygon is fully transparent (invisible).

As t decreases from 1 to 0, more of the color of the obscured object(s) will show through. At t=0, the transparent polygon becomes completely invisible. If no opaque attribute is specified, the default is 1 (fully opaque).

Polygons that are rendered as transparent have no color of their own, but merely filter the color of objects appearing behind them. This is according to the rule that each of the red, green, and blue components of the object behind is multiplied by the red, green, and blue components of the transparent polygon. This means that a transparent object rendered over a black background will be invisible. This also means that a purely blue transparent object rendering over a purely red object, will make the red object look more black (depending on the value of the Opaque specifier).

There are no specular highlights available on transparent objects.

To show polygon orientation relative to the eye point, the color which is transmitted through the transparent object is darkened according to the z-component of a surface normal. This means that with Phong, Gouraud, and flat shading, as the object bends away from the user, the transmitted color becomes darker.

To render any objects as transparent, you must at some time prior to rendering send a TRUE to input<11> of SHADINGENVIRONMENT. This input that allows you to turn transparency on and off.

6.2.5 ATTRIBUTE Node Inputs

Inputs to the attribute node are as follows:

- Input <1> accepts a real number as hue, a 2D vector as hue and saturation, or a 3D vector as hue, saturation, and intensity to specify COLOR for the front of the appropriate polygon(s) or both sides if no obverse attributes are given.
- Input <2> accepts a real number to set DIFFUSE
- Input <3> accepts an integer to set SPECULAR
- Input <4> accepts a real number to update the polygon's OPAQUE value.
- Inputs <5>...<10> are undefined
- Inputs <11>, <12>, <13>, and <14> correspond to <1>, <2>, <3>, and <4> but affect the obverse attributes if they exist.

If you send values that only change the hue to input <1> or input <11>, the saturation and intensity return to the default values of s=1 and i=1. You cannot change just one value and keep the remaining values as they were before you made the change. Essentially, if you do not send a 3D vector, default values are assumed for the missing variables.

For example, with the data definition

```
Dim_Red := ATTRIBUTES COLOR 130,1,.5 DIFFUSE .75 SPECULAR 8;
Object :=  WITH ATTRIBUTES Dim_Red
           POLYGON
              .
              .
              .
           POLYGON ;
```

If you sent 200 to input <1> of Dim_Red the resulting color parameter in the attribute node would be 200,1,1. To keep the saturation and intensity the same and change only the hue, you would send 200,1,.5 to input <1> of Dim_Red. This is the same if you want to change hue, saturation or intensity individually by sending a new value to the attribute node.

After changing the values in the attribute node, the changes will not be reflected until another rendering is requested.

6.2.6 Examples of the ATTRIBUTES Command

In the following example, an attribute node is created that defines the object to be blue. Since only the hue is specified for the color parameter, the default values for saturation and intensity (s=1, i=1) are assumed. The defaults for DIFFUSE and SPECULAR (d=.75, s=0) are also assumed.

```
Blue := ATTRIBUTES COLOR 120;
Object := WITH ATTRIBUTES Blue
          POLYGON
             .
             .
             .
          POLYGON ;
```

All the polygons in the object are blue since the attribute clause assigns the attributes defined by Blue for all polygons until superseded by another

WITH ATTRIBUTES clause. In the following example, the attributes before AND specify attributes for front facing polygons in the object and the attributes after AND specify the attributes for all back facing polygons.

```
Red_Green := ATTRIBUTES COLOR 120,.5,.75 DIFFUSE .25 SPECULAR 1
              AND COLOR 240,1,.25;
Object :=    WITH ATTRIBUTES Red_Green
              POLYGON
                 .
                 .
                 .
              POLYGON ;
```

All front facing polygons are colored red with .5 saturation and .75 intensity. The value for DIFFUSE is .25 and the value for SPECULAR is 1. All back facing polygons are green with 0 saturation and .25 intensity. Since no values for SPECULAR or DIFFUSE are given in the second set of attributes, the defaults are assumed.

The following object definition specifies attributes for display in the static viewport and also specifies the color of the polygon borders.

```
Pastel_Blue := ATTRIBUTES COLOR 3,.5,1 DIFFUSE .75 SPECULAR 5;
Object :=  WITH ATTRIBUTES Pastel_Blue OUTLINE 30
               POLYGON
                  .
                  .
                  .
               POLYGON ;
```

In this example, the shaded polygons on the raster display would be blue, with full saturation and .5 intensity. The specular value is .75 and the diffuse value is 5. The polygon edges are magenta (OUTLINE 30) when rendered in the static viewport with edges on, or in a hidden-line rendering.

6.3 Specifying Light Sources

Lights sources are specified with the ILLUMINATION command which creates illumination nodes. Illumination nodes can specify stationary lights, lights that can rotate with the object, or both. Illumination nodes are ignored during refresh in a dynamic viewport, and only those illumination nodes occurring in the descendent structure of a triggered solid or surface rendering operation node have any effect in shaded renderings. An

unlimited number of light sources are valid for flat and smooth shaded renderings. Light sources are not used in wash shaded (area filled) images.

All light sources are presumed to be an infinite distance from the object; however, you can specify the direction from which the lights illuminate the object. This direction is multiplied by the current rotation matrix to determine the direction to the light in image space. If, after transformation, the light source appears to originate from behind the object, it will cause the whole object to be unilluminated (appear black) except, perhaps, "glancing" specular highlights near the silhouette.

If no ILLUMINATION command is given, a default white light at (0,0,–1) with an ambient proportion of 1.0 is assumed. If not specified, intensity and saturation default to 1. If only hue and saturation are specified, intensity defaults to 1.

Syntax:

```
Name := ILLUMINATION X,Y,Z  [COLOR h[,s[,i]]]  [AMBIENT a] ;
```

where the **X,Y,Z** component is a vector from the origin pointing toward the light source.

The COLOR component specifies the color of the light source by defining hue, saturation, and intensity. The COLOR specification in this command is identical to the COLOR specification in the ATTRIBUTE command (refer to Section 6.2.1). The defaults are also the same.

The AMBIENT component controls the contribution of a light source to the ambient light. The net ambient lighting is determined by taking the sum of the products of the color and ambient proportion of each active light, dividing by the total number of active lights, then combining the result with the ambient input of the SHADINGENVIRONMENT function (discussed in the next section). AMBIENT is defined by a real number between 0 and 1. Increasing the value of *a* (ambient) for one light increases its contribution to ambient light. Values outside this range are changed to 0 or 1. The default value for *a* is 1.0.

Changing the values of the SHADINGENVIRONMENT (refer to Section 6.4) allows you to increase or decrease the intensity and color of the ambient light without the need to change each light source. Whatever the values, if all active light sources have the same specified proportion, then

all lights contribute equally to the ambient light. Decreasing a value for one light decreases its contribution to ambient light. Values outside the range [0..1] are changed to 0 or 1. The default value is 1.

In the following example, the ILLUMINATION command

```
Light := ILLUMINATION 1,1,-1 COLOR 180;
```

creates a node which defines a yellow light coming from the upper right. Since saturation and intensity are not specified, the defaults s=1 and i=1 are assumed. A default of 1.0 for the ambient proportion is also assumed.

Since the illumination node occurs in the data structure (unlike the attribute node which exists alone in mass memory), it is not explicitly referenced by the polygon data node.

The hierarchy with an illumination node is shown in Figure 13-22.

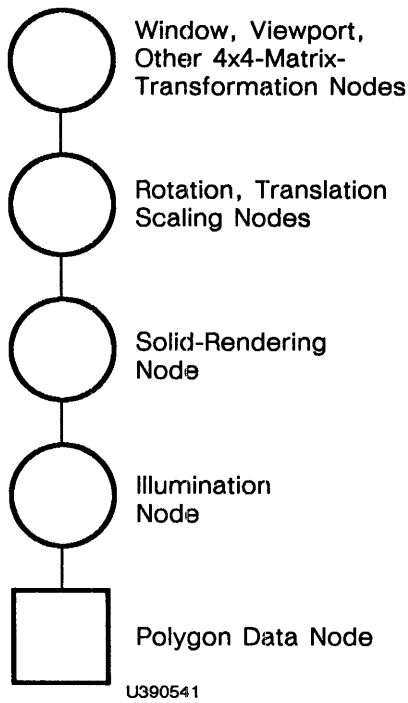


Figure 13-22. Hierarchy With Illumination Node

The illumination node must be under the rendering node in the display structure of the object.

The following is an example of how to use illumination nodes. There are two lights in the example: Sun.Light, which can be rotated independently of the object, and Moon.Light, which rotates with the object. To achieve this:

1. Both lights are underneath the rendering node in the structure.

2. Placing the Illumination nodes underneath the rendering node implies that the nodes will have the transformations of the object also applied to them. This is what happens for Moon (sending a rotation to Moon.Rot will concatenate with the transformations of the object).

3. The effect in step 2 is not desired for the sun, so a FIELD_OF_VIEW is inserted before the illumination node of Sun. This causes a rotation matrix sent to Sun.Rot to be the only matrix applied to Sun.Light.

4. Inserting a 4D matrix (caused by the FIELD_OF_VIEW) underneath a rendering node is not recommended. To avoid problems, the 4D matrix defined by Sun.Persp is identical to the 4D matrix defined by World.Persp and any change made to one (e.g., by a function net work) should be made to both. Failure to follow this suggestion may result in bad renderings.

```
Sun := BEGIN_STRUCTURE {light which can be rotated independently}
       Persp := FIELD_OF_VIEW 90 FRONT=2.2 BACK=3.6;
       LOOK AT 0,0,0 FROM 0,0,-3;
       Rot := SCALE BY 1;
       Light := ILLUMINATION 0,0,-1;
       END_STRUCTURE;


Moon := BEGIN_STRUCTURE  {light which rotates with the object}
        Rot := SCALE BY 1;
        Light := ILLUMINATION 0,0,-1;
        END_STRUCTURE;


World := BEGIN_STRUCTURE
         Persp := FIELD_OF_VIEW 45 FRONT=2.2 BACK=3.6;
         LOOK AT 0,0,0 FROM 0,0,-3;
         VIEWPORT HORIZONTAL=-1:1 VERTICAL=-1:1 INTENSITY=1:0;
```

```
            SET DEPTH_CLIPPING ON;
            Trans := TRANSLATE BY 0,0,0;
            Rot   := SCALE BY 1;
            Rendering := SURFACE_RENDERING;  {rendering node}
            INSTANCE OF Object, Sun, Moon;
            END_STRUCTURE;

DISPLAY World;
```

6.3.1 Illumination Node Inputs

Inputs to the illumination node are the following:

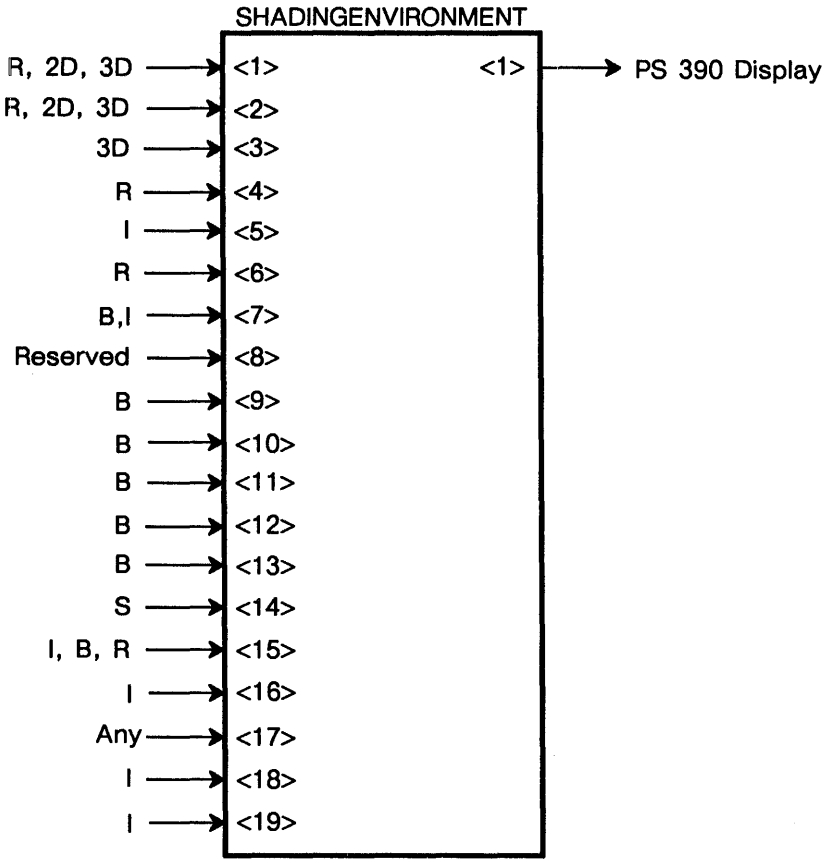
- Input <1> accepts a 3D vector as direction
- Input <2> accepts a real number as hue, a 2D vector as hue and saturation, and a 3D vector as hue, saturation, and intensity.
- Input <3> accepts a real number as the ambient proportion

Like the attribute node, if you send a real number to input <2> to change only the hue, the saturation and intensity return to the default values of s=1 and i=1. You cannot change just one value and keep the remaining values as they were before you made the change. If you do not send a 3D vector, the defaults for the variables not specified are assumed.

6.4 The SHADINGENVIRONMENT Function

An initial function instance called SHADINGENVIRONMENT allows you to control various static aspects of shaded renderings. These aspects affect the total environment in which shading operations are performed.

Sending values to the SHADINGENVIRONMENT function generally sets a parameter for the next requested shaded rendering rather than taking immediate effect. Inputs <7> (Screen Wash) and <17> (Restore look-up table) are the only inputs which cause an immediate visual effect. Note that SHADINGENVIRONMENT is different from other PS 390 functions in that any input will activate the function independent of the other inputs.



The inputs to the SHADINGENVIRONMENT function are discussed below.

6.4.1 Input <1> Ambient Color

Input <1> accepts a real number as hue, a 2D vector as hue and saturation, and a 3D vector as hue, saturation, and intensity, to specify the ambient color. Refer to the COLOR parameter of the ATTRIBUTES command for the meaning of the values (Section 6.2.1). The ambient color is combined with the result obtained from the light sources to determine the color of ambient light. The default ambient color is white, with a default intensity of 0.25.

6.4.2 Input <2> Background Color

Input <2> accepts a real number as hue, a 2D vector as hue and saturation, and/or a 3D vector as hue, saturation, and intensity to specify the background color. Refer to the COLOR parameter of the ATTRIBUTES

command for the meaning of the values (Section 6.2.1). The current static viewport specified by input <3> of SHADINGENVIRONMENT is colored with the background color prior to any shaded rendering done in the refresh mode (refer to input <9>). The default background color is black (0,0,0). For information on how to change the background color for dynamic viewports, refer to notes on input <2> of the initial function instance PS390ENV.

6.4.3 Input <3> Static Viewport

Input <3> accepts a 3D vector which specifies physical pixel locations for the viewport where shaded renderings are displayed. Static raster viewports are always square, the lower left corner being given by the X and Y coordinates of the vector, and its size given by the Z coordinate, such that the upper right corner is at (X+Z,Y+Z). Values are rounded to the nearest pixel. The default viewport is V3D(80,0,863).

The viewport can be used for rendering multiple images side by side on the display. For example, sending V3D(0,-80,1023) would be a valid command to specify the largest recommended value for the static viewport. This viewport encompasses the entire displayable screen as well as the undisplayable area in Y that is in excess of 863. Images in this viewport are clipped to the physical raster for which 0<=X<1024 and 0<=Y<864.

6.4.4 Input <4> Exposure

Input <4> accepts a real number as the exposure, controlling the overall brightness of the picture. The exposure is similar to the exposure control of a camera. If a picture is taken of an object with a very bright specular highlight, it may be so bright that the rest of the object is darkened. If three light sources exist, the object would be about three times brighter, making the object too bright. The exposure can be brought down to control this.

The exposure is multiplied by the intensity at each pixel and the result clipped to the maximum intensity. This enables the overall brightness of a rendering to be increased without causing bright spots to exceed maximum intensity (instead forming "plateaus" of maximum intensity). Recommended exposure values may vary between 0.3 and 3.0. The default exposure is 1.

6.4.5 Input <5> Anti-aliasing control (Edge smoothing)

Input <5> accepts an integer which allows users to chose between having a relatively fast rendering with jagged edges along the polygons or having slower renderings with smoother edges and correct interpretations of inter-penetrating polygons. Anti-aliasing is accomplished by taking 16 samples per pixel instead of only one. You are given the choice of having no edge smoothing at all, smoothing along the edges only, or sampling 16 times within every pixel for every polygon. The default value for this input is 0.

Sending fix(0) to this input produces no smooth edges, and produces the fastest renderings. Polygons are rendered with one sample per pixel.

Sending fix(1) produces smooth edges, but may not correctly resolve visibility between surfaces that are extremely close in their Z values or that are interpenetrating. The 16 samples are taken only where the edges of the polygon touch a pixel. The interior of the polygons is rendered with one sample per pixel. This method has a speed intermediate between a fix(0) and a fix(2).

Sending fix(2) to input <5> produces full anti-aliasing. This rendering method is the slowest, but it produces full visibility resolution for inter-penetrating polygons. Sixteen samples are taken for every pixel in every polygon.

6.4.6 Input <6> Depth Cueing

Input <6> accepts a real number in the range of 0 to 1 to control depth cueing in the shaded image (1 specifying no depth cueing and 0 specifying maximum depth cueing). As perceived depth from the viewer increases, the colors are mixed with the ambient light color. Thus, if a 3D vector with a value of black (0,0,0) is sent to the ambient input <1> and a 0 is sent to the depth clipping input <6>, objects are rendered with a ramp ending in black at the back clipping plane. A 1 sent to input <6>turns off depth cueing. The default value for this input is 0.2.

6.4.7 Input <7> Screen Wash

Input <7> accepts a Boolean value or an integer and causes an immediate visual effect. Sending a TRUE to this input clears the entire screen to static and causes a screen wash with the current static background color. Sending a FALSE to this input clears the currently specified static viewport and causes the viewport to be filled with the current static background color.

Sending fix(0) to input <7> has the same effect as sending TRUE.

Sending fix(1) to input<7> has the same effect as sending FALSE.

Sending fix(2) to input<7> clears the entire screen to a dynamic screen and causes a screen wash with the current dynamic background color set by input <2> of PS390ENV. This may be done to clear a shaded image before displaying a new dynamic image.

Sending fix(3) to input <7> clears the currently specified static viewport with the current dynamic background color.

6.4.8 Input <8> Reserved

6.4.9 Input <9> Refresh/Overlay Control

Input <9> accepts a Boolean which determines whether the screen is cleared with the current background color before the rendering is performed. Sending a TRUE to this input causes the current object to be rendered on top of the image currently displayed in the static viewport. Sending FALSE to this input causes the static viewport to be cleared with the current background color before an object is rendered. The default value is FALSE.

6.4.10 Input <10> Color By Vertex Control

Input <10> accepts a Boolean which controls the use of vertex colors. Color by vertex is accomplished by defining a color for each vertex in the polygon. A TRUE to this input enables the colors defined at each vertex. A FALSE to this input enables the color(s) specified in the ATTRIBUTES command. The default value for this input is FALSE.

6.4.11 Input <11> Opaque (Transparency) Control

Input <11> accepts a Boolean which enables or disables the transparency assigned to the polygon. Transparency is assigned to the polygon with the OPAQUE clause of the ATTRIBUTES command. Transparent polygons are created by modifying the ATTRIBUTES command as follows:

```
Name := ATTRIBUTE [COLOR h[,s[,i]]] [OPAQUE t]
                  [DIFFUSE d] [SPECULAR s];
```

where **t** refers to a value between 0 and 1, with 1 being fully opaque and 0 being fully transparent. When t=0, the object is completely invisible. As **t**

decreases from 1 to 0, more of the color of the obscured object(s) will show through. The default value for this input is FALSE (fully opaque).

6.4.12 Input <12> Specular Highlight Control

Input <12> accepts a Boolean which turns specular highlights on and off. Flat, Gouraud and Phong shading use a shading equation that can process multiple light sources and calculate specular highlights. The default value is TRUE which means specular highlights are turned on.

6.4.13 Input <13> Special Color Blending for Spheres

Input <13> accepts a Boolean value which turns color blending on and off. The color blending is used for correct spherical rendering (used in molecular modeling). Sending a TRUE turns the color blending on. Sending a FALSE turns it off. The default is FALSE.

6.4.14 Input <14> Update Attribute Table

Input <14> accepts a string which is the name of a 3D tabulated vector list used to update the attribute table that specifies color, radii, diffuseness, and specular highlights for spheres and lines. The attribute table has 0 to 127 entries with six table components for each entry. The attribute table can be updated by encoding the table entries into a named PS 390 vector list and then sending the name of the vector list to this input. The six table components are encoded into two consecutive 3D tabulated vector list.

The table has the following components: hue, saturation, intensity, radius, diffuse, specular.

Hue is a real number in the range 0 to 360. Saturation and intensity are real numbers in the range 0 to 1. Radius is a real number greater than 0. Diffuse is a real number in the range 0 to 1. Specular is an integer in the range 0 to 255.

The table is initialized as follows:

INDEX	Hue	Sat	Intensity	Radius	Diffuse	Specular
0	0	0	0.5	1.8	0.7	4 (Gray)
1	0	0	1	1.2	0.7	4 (White)
2	120	1	1	1.35	0.7	4 (Red)
3	240	1	1	1.8	0.7	4 (Green)
4	0	1	1	1.8	0.7	4 (Blue)
5	180	1	1	1.7	0.7	4 (Yellow)
6	0	0	0.7	1.8	0.7	4 (Gray)
7	300	1	1	2.15	0.7	4 (Cyan)
8	60	1	1	1.8	0.7	4 (Magenta)
9	0	0	0	1.8	0.7	4 (Black)
10–127						(Color Wheel)

Spheres use all six of these components. Lines use only the hue, saturation, and intensity components.

The (**h**) specifier in the WITH OUTLINE clause is used as the index into this table. The color of polygon interiors does not used this table; only the color of polygon outlines in static raster rendering is done this way.

6.4.15 Input <15> Polygon Edge Enhancement

Input <15> accepts a Boolean, or a real number in the range 0–1, or an integer in the range 0–2.

A real value sent to this input adds an offset to the Z-values of lines. A number between 0.05 and 1.0 causes the lines to be displayed in front of other objects with the same Z value. This allows the enhancement of polygon edges. Numbers between 0.05 and 0.0 are clamped to 0.05, which produces shaded renderings with the lines and edges brought forward slightly in Z.

Sending a Boolean to this input allows you to toggle the display of polygon edges on and off. A TRUE causes lines to be drawn along polygon borders, thus enhancing the edges, and temporarily turns on full antialiasing. A FALSE causes polygons to be rendered normally without edge enhancement. The default is FALSE (edges off).

Sending Fix(0) to <15> causes polygons to be rendered without enhanced edges. This is the same as sending a FALSE, and is the default condition.

Sending Fix(1) causes polygon edges to be enhanced, and causes all edges including those marked as soft to be displayed. This is the same as sending a TRUE.

Sending Fix(2) causes polygon edges to be enhanced, but only those edges marked as hard edges are displayed.

6.4.16 Input <16> Algorithm

This input accepts an integer value of 1 or 0 to choose between one of two possible algorithms for resolving visibility in a rendering.

Sending Fix(0) causes a scan-line zbuffer algorithm to be applied. This algorithm is used in rendering solids; it causes all obscured polygons to remain undisplayed. This is the default shading algorithm.

Sending Fix(1) causes the painters algorithm to be applied to the rendering. This algorithm renders an image by filling (painting) each polygon from back to front Z-value. Occasionally this algorithm displays polygons which which should be obscured.

6.4.17 Input <17> Restore System Look-up Table

Any value sent to this input restores the gamma-corrected system look-up table. This is the table responsible for producing antialiased lines of good line quality. Sending a value to this input has an immediate visual effect.

6.4.18 Input <18> Vertex Normals Control

This input accepts an integer value in the range 0 to 2. Values sent to this input affect vertex normals.

Sending Fix(0) causes vertex normals to remain unchanged from their original definition. This is the default.

Sending Fix(1) inverts all vertex normals that are backwards and that are on backfacing polygons to make the polygons appear forward. This is useful for the user who knows the desired direction for normals to point, but who does not necessarily specify polygon vertices in a consistently clockwise

fashion. This is applicable to surface renderings only. The AND specifier of the ATTRIBUTES command should not be used when using this input to reverse normals.

Sending Fix(2) flips vertex normals that are backwards and are on polygons that are frontfacing to make the polygons appear forward. This is useful for performing mirrored modeling operations, i.e., using a -y scale factor to produce an image mirrored about the xz plane. Again, this is applicable only to surface renderings.

6.4.19 Input <19> Stereo

This input is used for stereo renderings. Sending Fix(1024) causes renderings to be produced on the entire display, including the (usually) missing 160 scan lines at the bottom of the screen. This input is used for rendering solid polygons, spheres, and raster lines in 3-dimensional stereo (using the Tektronix LCD screen).

7. Summary

The POLYGON command defines collections of polygons from which renderings can be created. This is a data definition command that creates a polygon data node in the data structure of the object. Objects defined as polygons are the only objects that are eligible for rendering operations.

There are two types of rendering operations: those performed in the dynamic viewport, and those performed in a static viewport. Rendering operations in the dynamic viewport can result in a cross section of a displayed object, sectioning of an object relative to a sectioning plane, or backface removal.

Rendering operations performed in the static viewport include hidden-line removal, flat shading, wash shading, and smooth shading (Gouraud and Phong).

Polygonal objects must be defined correctly to produce correct renderings.

7.1 POLYGON Command Syntax

Given,

```
<vertex> = [ S ] x,y,z [ N x,y,z ] [C h[s[i]]]
<polygon> = [WITH ATTRIBUTES name1] [WITH OUTLINE h] [COPLANAR]
             POLYGON <vertex> ... <vertex>
```

The POLYGON command is:

```
Name := <polygon> <polygon> . . . <polygon> ;
```

where:

A vertex definition has the form [S] x,y,z [N x,y,z] [C h[s[i]]]

where:

- **S** indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon. If the S specifier is used for the first vertex in a polygon definition, the edge connecting the last vertex with the first is soft.
- **N** indicates a normal to the surface with each vertex of the polygon. Normals are used only in smooth-shaded renderings. Normals must be specified for all vertices of a polygon or for none of the vertices of a polygon. Normals do not need to be present for all polygons in the object. If no normals are given for a polygon, they are defaulted to the same as the plane equation for the polygon.
- **x**, **y**, and **z** are coordinates in a left-handed Cartesian system.
- **C** indicates a color that is assigned to the vertex. During shading operations, this color is interpolated across the polygon to the other vertices.
- **h,s,i** are values in the Hue/Saturation/Intensity color system. Together these values create a color.

WITH ATTRIBUTES is an option that assigns the attributes defined by name1 for all polygons until superseded by another WITH ATTRIBUTES clause.

WITH OUTLINE is an option that specifies the color of the edges of polygons in shaded renderings or in hidden-line renderings.

COPLANAR declares that the specified polygon and the one immediately preceding it have the same plane equation.

7.2 Defining Polygonal Objects

There is no syntactical limit on the number of polygon clauses in the group.

Polygons are implicitly closed. The first vertex should not be repeated when defining a polygon.

No more than 250 vertices per polygon may be specified and no less than three.

The vertices of a polygon must be coplanar. The plane equation is determined from any three non-colinear vertices.

Concave polygons are acceptable. Degenerate polygons and polygons that intersect themselves or others are unacceptable. No specific checks are made for these conditions.

Polygons are not pickable and polygon nodes have no inputs from which they can be modified with function networks.

7.3 Constructing Surfaces and Solids

Surfaces and solids can be defined. Solids enclose a volume of space, while surfaces do not.

In a solid, every edge of every polygon must coincide with the edge of a neighboring polygon.

For surfaces and solids, polygons are defined by listing their vertices in a clockwise order in the polygon clause.

In a solid, the common edge where two polygons join must run in opposite directions. This arrangement is essential to produce correct renderings. The system does not check for this condition.

A solid cannot contain three or more polygons which have a single edge in common, although surfaces may.

The SURFACE_RENDERING and SOLID_RENDERING commands determine the nature of a polygonal object.

7.4 The COPLANAR Option

Inner contours may be defined to create objects with holes or protrusions.

Vertices of inner contours must be listed in the opposite direction to the corresponding outer contour.

An inner contour should not be defined unless it is coplanar with some surrounding outer contour.

All members of a set of consecutive coplanar polygons are taken to have the same plane equation, that of the previous polygon not containing the COPLANAR option.

If COPLANAR is specified for the first polygon in a polygon list, it has no effect.

7.5 The Soft Edge Option

The S specifier before a set of X,Y,Z coordinates indicates that the edge drawn between the previous vertex and this one represents a soft edge of the polygon.

Soft edges are positions in the original object. If either edge of a common edged pair is declared soft, the entire edge is considered soft. Soft edges are displayed the same as hard edges, however they are only drawn once. The exception to this occurs when enhanced edges have been requested on a shaded rendering and a fix(2) has been sent to input<15> of SHADINGENVIRONMENT. In this instance, polygon edges are enhanced, but those edges declared as soft are not displayed.

7.6 The Color Option in a Dynamic Viewport

Color for polygons displayed in a dynamic viewport is specified with the SET COLOR command. Color is specified for complete polygons, not individual edges.

7.7 Specifying Normals

When a polygon is used to approximate a curved surface, the smooth appearance of the surface can be restored in a smooth shaded rendering by approximating a surface using normals. A normal to the surface is given with each vertex of the polygon specified **N x,y,z.**

7.8 Memory Usage

The rendering process requires that a block of mass memory be available as working storage. This memory can be explicitly reserved with the command RESERVE_WORKING_STORAGE n, where the current working storage is replaced with another containing at least n bytes. It is also possible to allow the system to automatically calculate working storage for you. If n is less than or equal to 0, the system will automatically calculate the amount of memory needed for the rendering process and display the amount used at completion.

The best time to explicitly reserve working storage is immediately after booting; typically, you should reserve 200,000 to 400,000 bytes of working storage when you begin a session.

Working storage is not freed by the INITIALIZE command.

In addition to the working storage space, extra mass memory is needed to create static raster renderings. This memory is referred to as transient memory and is automatically allocated and deallocated by the system.

7.9 Marking an Object for Rendering

Syntaxes for the rendering commands are:

```
Name := SOLID_RENDERING APPLIED TO Name1;
Name := SURFACE_RENDERING APPLIED TO Name1;
```

where Name1 names either (a) a Polygon node, or (b) an ancestor of one or more polygon nodes. If (b) is the case, any rendering referring to Name1 is performed immediately on all of the polygon objects descended from Name1.

Only polygons nodes are used in renderings. Vector and character nodes occurring beneath a rendering node are ignored by the rendering operations.

Transformation nodes are lost in the rendering, but their effect is incorporated into the data nodes.

7.10 Establishing a Sectioning Plane

The SECTIONING_PLANE command creates a sectioning plane node which indicates that a descendant polygon is a sectioning plane. The syntax is:

```
Name := SECTIONING_PLANE APPLIED TO Name1;
```

where name1 names either (a) a POLYGON command or (b) an ancestor of a POLYGON command.

7.11 The Data Definition of the Sectioning Plane

The sectioning plane is the plane containing the polygon defined by the first polygon clause of the first polygon node encountered by the display processor as it traverses the branch beneath a sectioning plane node.

The sectioning plane is the plane in which a specified polygon lies. The polygon itself need not intersect the object to be sectioned, as long as some part of the plane does.

No SOLID_RENDERING or SURFACE_RENDERING operation node, whether below or above the sectioning plane node, may be an ancestor of the defining polygon of a sectioning plane. The PS 390 interprets such polygons as objects to be rendered rather than as sectioning plane definitions and issues a "sectioning plane not found" message when a sectioning attempt is made.

7.12 Saving a Rendering

A rendering is saved by a string sent to input <1> of the SOLID_RENDERING or SURFACE_RENDERING operation node. The string should specify the name of the node which is to contain the saved rendering data. If the named node does not exist, it is created; if it does exist, the saved rendering data replaces the original contents of the node.

All polygons in the rendering are taken into account in the saved rendering. It is not possible to exclude selected polygons or polygon data nodes from saved renderings.

7.13 Specifying Color and Highlights for Static Viewports

Specifying attributes (specular and diffuse highlights, color, and transparency), of a polygon for display in a static viewport is done via the WITH ATTRIBUTES clause of the POLYGON command.

Given the polygon syntax:

```
Name := <polygon> <polygon> . . . <polygon> ;
```

the attributes option is

```
<polygon> = [WITH ATTRIBUTES Name1] [OUTLINE h]
             POLYGON  <vertex>...<vertex>
```

The ATTRIBUTES command is:

```
Name := ATTRIBUTES <attr> [ AND <attr> ] ;
```

Given:

```
<attr> = [COLOR h [,s[,i]]]
          [DIFFUSE d]
          [SPECULAR s]
          [OPAQUE t]
```

7.14 Specifying Light Sources

Lights may be stationary or rotate with the object or both. If no ILLUMINATION command is given, a default white light at (0,0, 1) with an ambient proportion of .25 is assumed. If intensity and saturation are not specified, both values default to 1.

Syntax:

```
Name := ILLUMINATION x,y,z [COLOR h [,s[,i]]] [AMBIENT a] ;
```

Like the attribute node, if you send a real number to input <2> to change only the hue, the saturation and intensity return to the default values of s=1 and i=1.

7.15 The SHADINGENVIRONMENT Function

An initial function instance called SHADINGENVIRONMENT allows you to control various static factors of shaded renderings. This function controls factors that affect the total environment in which shading operations are performed. There are currently nineteen inputs to the function.

Sending values to the SHADINGENVIRONMENT function generally sets a parameter for the next requested shaded rendering rather than taking immediate effect. Note that SHADINGENVIRONMENT is different from other PS 390 functions in that any input will activate the function independent of the other inputs.

GT14. RASTER PROGRAMMING

GT14. RASTER PROGRAMMING

DISPLAYING HOST-GENERATED IMAGES WITH THE PS 390 RASTER SYSTEM

CONTENTS

ILLUSTRATIONS

Section GT14

Raster Programming

Displaying Host-Generated Images with the PS 390 Raster System

Introduction

The PS 390 raster system consists of a printed circuit card that outputs static images to a 1024 (column) by 864 (row) pixel raster display. Each pixel is 24 bits deep for addressing into a red-green-blue color lookup table (CLUT) that is 24 bits deep.

The PS 390 raster system can be used to display polygon wireframe models and shaded images derived locally from PS 390 polygonal models, or it can be used as a frame buffer to display host-generated images. When used as a frame buffer, the PS 390 only serves as a communications link between the host and the raster system. No standard PS 390 commands or data structures are used to display host-generated images.

This document describes how to display host-generated images using the FORTRAN and Pascal Graphics Support Routines (GSRs) and user-generated routines. Programming examples for both methods are provided.

This manual assumes that you are familiar with creating raster images. If you need background in this subject, the following books contain detailed sections on raster graphics:

J.D. Foley and A. Van Dam: *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, 1982.

William M. Newman and Robert F. Sproul: *Principles of Interactive Computer Graphics.* McGraw-Hill Book Company, 1979.

Conrac Division: *Raster Graphics Handbook.* Conrac Corporation (600 North Rimsdale Avenue, Covina CA, 91722), 1980.

Donald P. Greenburg: *Introduction to Raster Graphics.* Siggraph '83 Tutorial, 1983.

1. PS 390 Raster Concepts

The basic steps required to display a host-generated picture on the PS 390 are:

- Determine what your picture will look like (determine pixel values and the addresses into the lookup tables).
- Set the logical device coordinates to specify the proper size and position of the raster image.
- Transfer this information from the host to the PS 390 via the GSRs or user-written routines.

Three features of the PS 390 image buffer mode are:

- It is run-length encoded.
- It uses red, green, and blue color lookup tables.
- It specifies logical device coordinates to define the portion of virtual address space (the total coordinate area in which pictures can be created) that contains the raster picture. This allows flexibility in positioning a picture relative to the actual screen display.

These concepts and their application in the PS 390 raster system are discussed in detail in the following sections.

1.1 Run-Length Encoding

Some raster systems require that you encode a raster picture pixel by pixel. That is, each pixel on the raster screen must be addressed individually. In contrast, the PS 390 accepts raster data from the host in run-length encoded format. Both the GSRs and user-written routines specify run-length encoding.

In run-length encoding, a set of consecutive pixels of the same color is specified in a single command containing the number of consecutive pixels and the color value of the pixels. Since, in practice, most pictures contain many sequences of consecutive pixels of the same color, run-length encoding allows more efficient picture transmission than pixel-by-pixel encoding in all but the most complex and high-resolution raster pictures.

For example, if the bottom third of your raster picture is a background color, one run-length encoded command could specify the color for those 294,912 (1024x288) pixels. Pixel-by-pixel encoding would require 294,912 separate single-pixel commands.

1.2 Color Lookup Tables

Any displayable color is a combination of three components--red, green, and blue, the primary phosphor colors used in the raster display's additive color process. Varying the intensity of these three color components produces the wide variety of colors available to the raster display.

The PS 390 raster system does not specify colors directly, but rather refers to locations in color lookup tables (CLUTs) that contain the color entries. Each pixel on the raster display is 24 bits deep. That is, 24 bits of data address each pixel's color value in the CLUTs with 8 bits to specify entries in the CLUTs for each red, green, and blue color. Since the 24 bits of pixel data do not specify a color directly, this is sometimes referred to as "pseudocolor" specification.

There are three CLUTs on the raster card, one each of red, green, and blue. The CLUT entries (derived from the 24 bits of pixel data) contain a precise color level, or intensity for a specific color.

Each entry in the tables is 8 bits deep, providing 2^{24} potential colors. Each set of 8 bits specifies the intensity of the corresponding red, green, or blue color. Each table has 256 (0-255) possible entries (i.e., 8 bits of address per lookup table), providing 2^{24} (256^3) usable colors. This provides more displayable colors than there are actual pixels on the raster screen. Naturally, the human eye cannot distinguish between this many shades of colors. This permits "smooth shading" of host-generated raster pictures. If the eye could actually perceive the slight differences in shades of colors, you would see banding (stripes of different shades) instead of smooth shading.

The CLUTs are preloaded at boot time with a gamma-corrected lookup table. This gives the appearance on the screen of a linear change in integrity as the index changes (i.e., location 20 is twice as bright as 10).

Figure 14-1 provides a graphic representation of how the 24 bits of pixel data map to the 24 bits of the CLUTs. The top of the figure shows the image buffer memory of the system. Each pixel contains 24 bits of pixel data made up of 8 bits of red, 8 bits of green, and 8 bits of blue pixel data. These bits specify the address in the CLUTs.

The 8-bit entries in the CLUTs specify intensities of red, green, and blue (RGB). The 8-bit digital-to-analog converters change these digital values to analog signals which drive the red, green, and blue guns that stimulate the RGB triads on the raster display screen. The eye blends these intensities to generate the specified color.

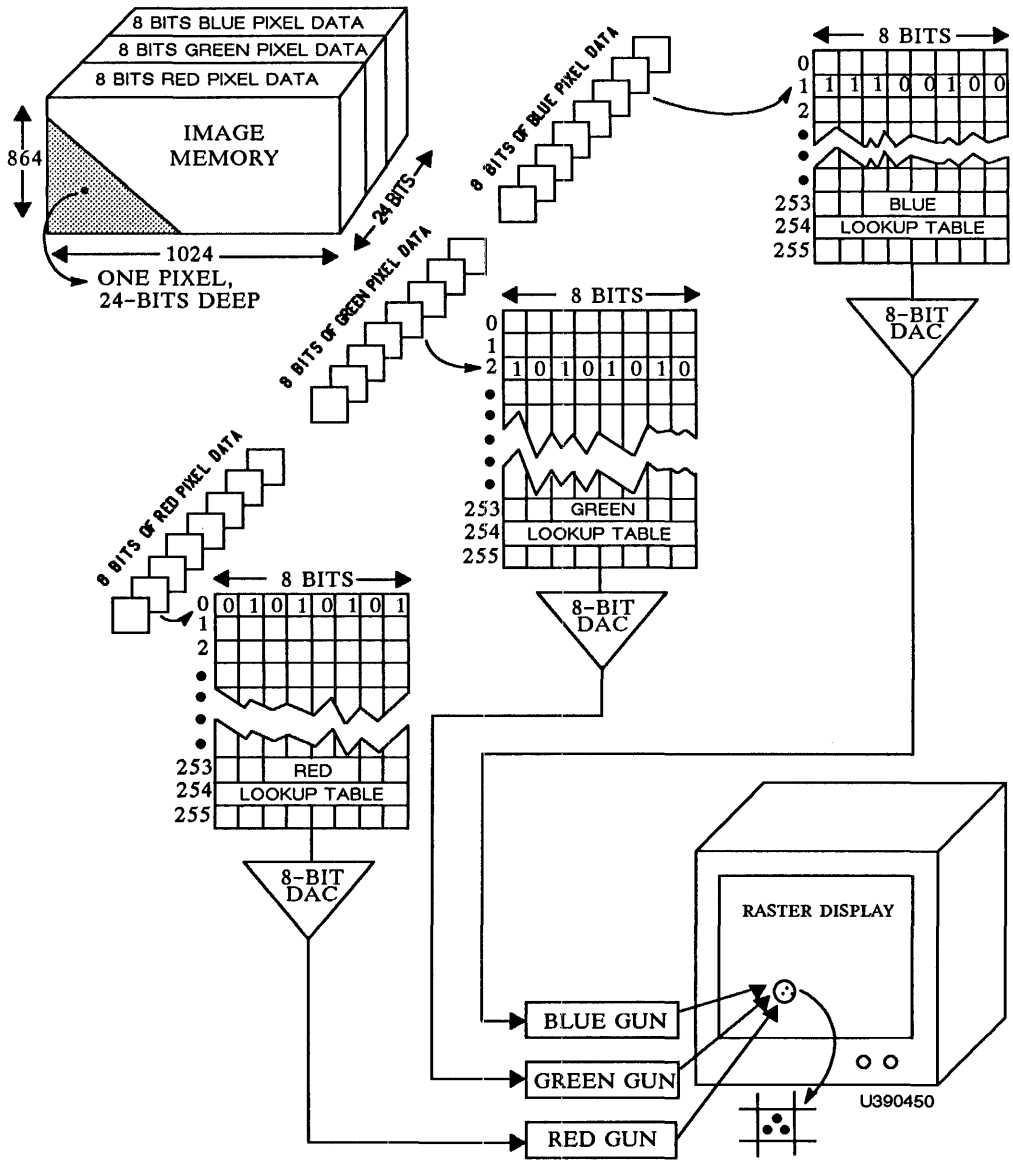


Figure 14-1. Pixel Mapping to Color Lookup Tables

2. Logical Device Coordinates

The raster option has a virtual address space from -32768 to 2047 in both X and Y (see Figure 14-2). The portion of virtual address space that is actually displayed is from 0 to 1023 in X and from 0 to 863 in Y, and is called "screen space." A picture can be placed anywhere in the virtual address space. The portion of that picture which overlaps with screen space will be displayed (see Figure 14-2).

When the raster system is booted, the logical device coordinates default to a 1024x864 screen size starting at 0,0. Of course, once you have specified a different set of logical device coordinates, this becomes the new default value.

The logical device coordinates are specified as ranges of X and Y values. They define the dimensions and position of the area that contains the picture and can be larger or smaller than screen space.

Logical device coordinates specify:

1. The size of the raster picture.
2. Where the picture will appear in virtual-address space.
3. Where "wraparound" will occur. Wraparound occurs when the run-length encoded command hits the limit of the X coordinates (the end of a row of pixels) and begins a new row of pixels.

When sending the logical device coordinates for a picture that is larger than screen space, data outside of screen space is discarded. The data can be sent again to the image buffer to display another portion of the raster image in screen space. Changing the logical device coordinates and starting position, and resending the picture, places a new portion of the image in screen space without recalculating the image. Only the portion of the logical device coordinate picture that coincides with screen space will be visible (see Figures 14-3 and 14-4).

The logical device coordinate range should correspond to the actual size of the precalculated raster image. If correctly run-length encoded, when the current pixel location reaches the right boundary of the logical device coordinates, the next pixel location automatically begins at the left boundary of the logical device coordinates with the Y value incremented by one, addressing the pixels in the next row. In other words, raster images in the PS 390

go from left to right, bottom to top. This allows you to send an entire picture with only one current pixel location rather than having to start each new row of pixels with a new pixel location.

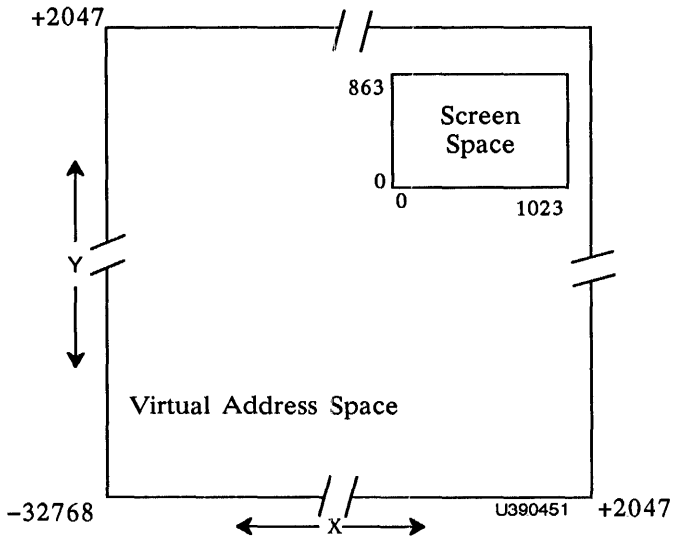


Figure 14-2. Virtual Address Space and Screen Space

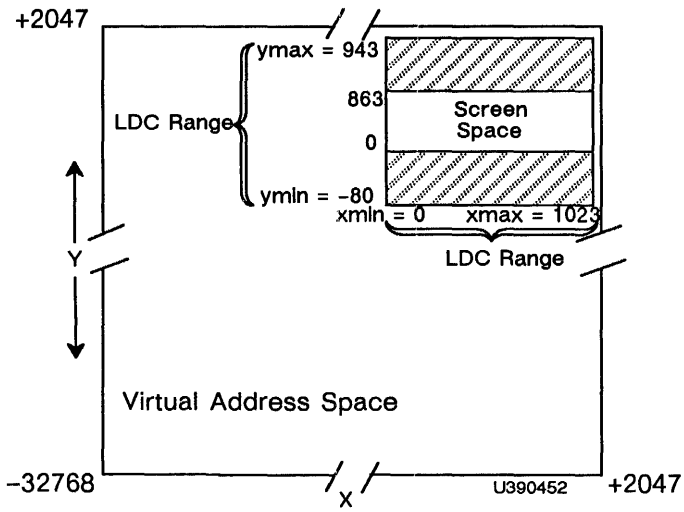


Figure 14-3. Coordinate Ranges for a Raster Display

Figure 14-3 illustrates the coordinate ranges for virtual address space (the total coordinate area in which pictures can be created), sample logical device coordinates specified by the programmer (in this case, shown in the shaded area specifying a 1024 x 1024 image), and the actual screen space that can be displayed at any given time.

Run-length encoded commands make no mention of absolute pixel location. The commands simply specify the next (n) consecutive pixels starting at the current pixel location. (The current pixel location is the point in the logical device coordinates where the run-length encoded command begins loading pixels.) It follows that an entire picture can be repositioned by changing the logical device coordinate specifications (which are the only specifications that refer to absolute pixel locations) and retransmitting the picture data that fall in the new logical device coordinates. No change to the encoded pixel data is necessary.

Figure 14-4 shows virtual address space (the entire area in which a picture can be created), the logical device coordinate range (specified by the WRPIX command described in the next section), and the screen space containing the portion of the picture that will actually be displayed on the raster monitor.

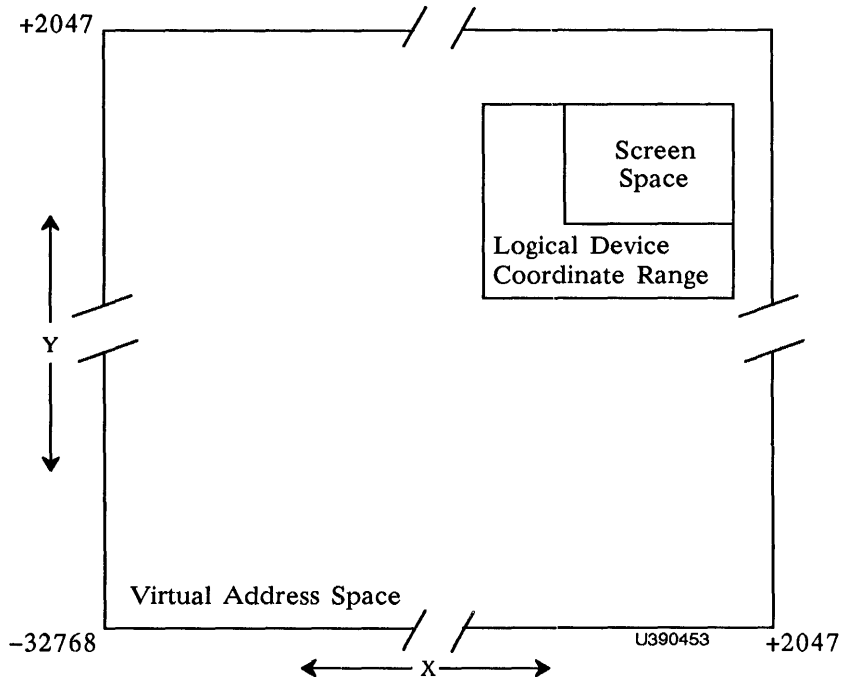


Figure 14-4. Virtual Address Space, Logical Device Coordinate Range, and Screen Space

Figure 14-5 shows that the logical device coordinate range has been changed so that the lower left-hand area of the logical device coordinate range coincides with screen space. Note that a new section of the raster image is in screen space after the picture data has been retransmitted. Also, screen space remains fixed: the new logical device coordinate range has changed what actually appears in screen space.

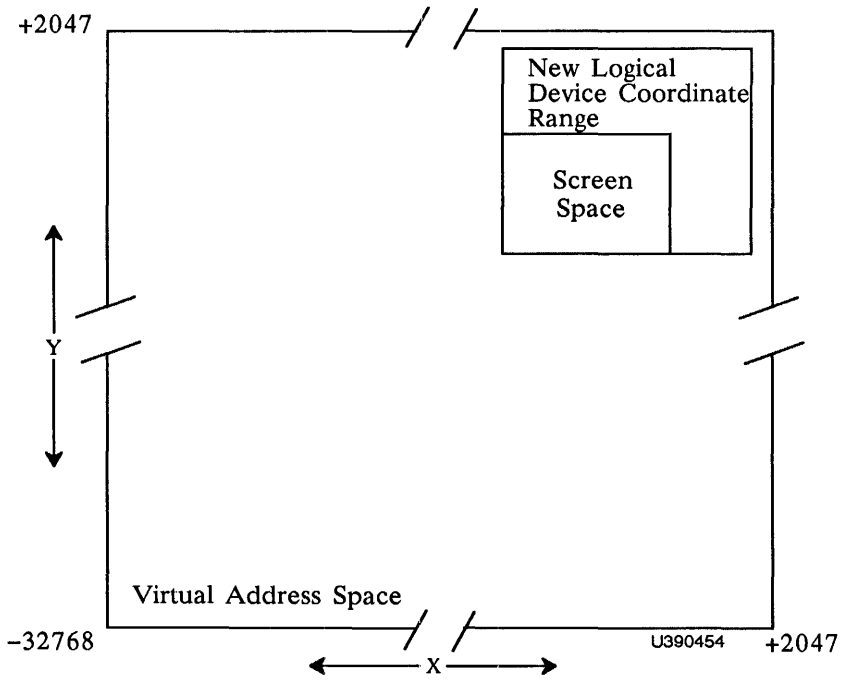


Figure 14-5. Displayed Raster Image Within Lower Left Logical Device Coordinate Range

The raster option has the ability of virtual pixel addressing of:

-32768 <= X <= 2047, -32768 <= Y <= 2047

of which the portion that is actually displayed is:

0 <= X <= 1023, 0 <= Y <= 863

The logical device coordinates (Xmin <= X <= Xmax, Ymin <= Y <= Ymax) can be any subset of this range.

To position a raster image of 200x200 on the center of the screen, the values should be:

```
Xmin = 412   ([1024-200]/2)
Xmax = 611   (Xmin + 200 -1)
Ymin = 332   ([864-200]/2)
Ymax = 531   (Ymin + 200 -1)
```

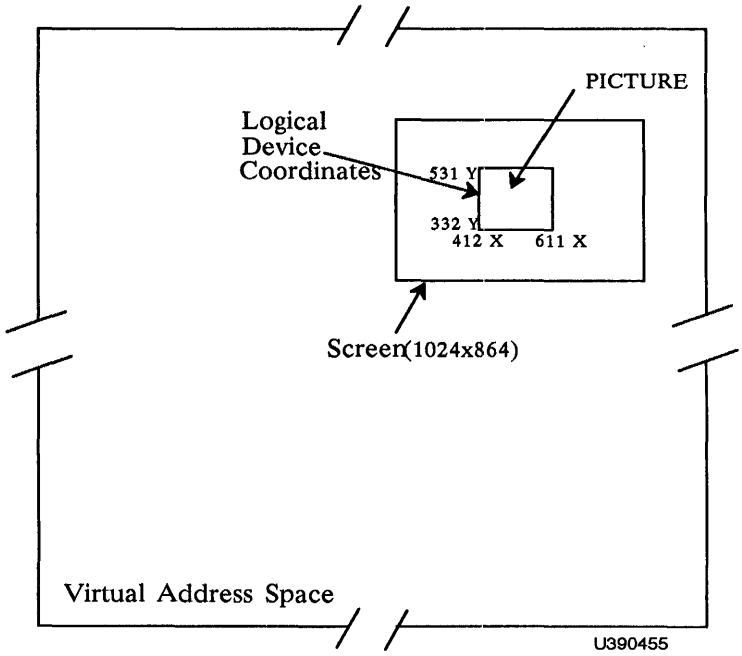


Figure 14-6. Centering a Raster Picture

To get the center of a 1024x1024 image on the physical screen, the logical device coordinates values should be:

```
Xmin =   -0  ([1024-1024]/2)
Xmax = 1023  (xmin + 1024 -1)
Ymin =  -80  ([864-1024]/2)
Ymax =  943  (ymin + 1024 -1)
```

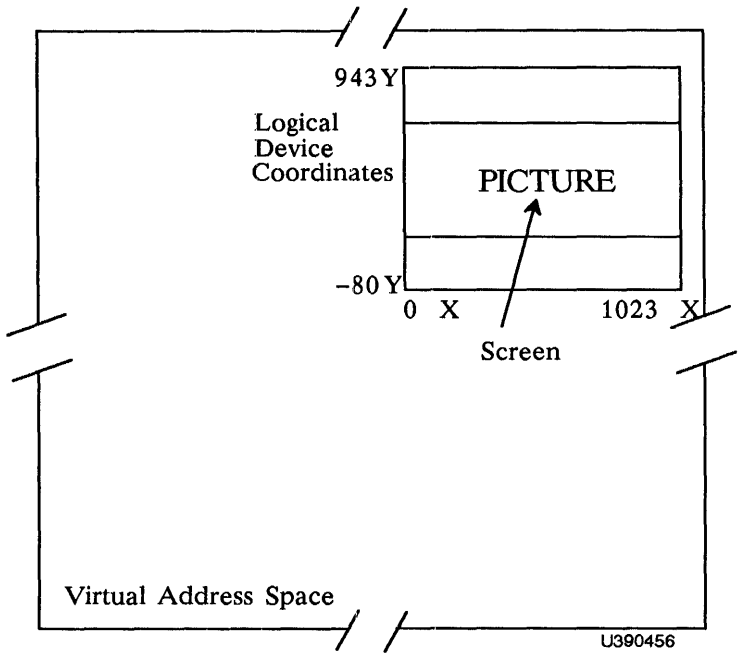


Figure 14-7. Displaying a Section of a Raster Picture

3. Encoding a Picture with the Raster Mode

This section discusses the basic raster mode of the image buffer, Write Pixel Data (WRPIX), and how it functions. The implementation is described in sections 4 and 5.

Table 14-1 in section 4 provides a quick reference to commands in WRPIX mode and shows the GSRs that implement these commands.

3.1 Writing Pixel Data

There are two basic steps to encoding a picture in WRPIX mode: setting up raster display parameters and changing the picture on the raster display.

Two WRPIX mode commands are used to establish basic operating parameters for the raster display:

- Set Logical Device Coordinates

  This command positions the picture in virtual address space.

- Set Current Pixel Location

  This command establishes a starting point (the current pixel) in the logical device coordinates where the next WRPIX mode command begins.

Two WRPIX-mode commands are used to change the picture on the raster display:

- Erase Screen

  This command fills all of pixel memory to one value, an address into the CLUTs.

- Load Pixel Data

  This command writes specific values to pixels.

NOTE

All data not in the range of the actual display are discarded. When the raster display is first set up in WRPIX mode, the X-Y position must be set to start position by using the X-Y position command. Data are stored in the pixels sequentially and wrap-around occurs at X maximum position until a new X-Y position is received. Whole picture representations require at least one X-Y position.

4. Graphics Support Routines

The GSRs provide the easiest way to send pixel information to the display and avoid the need for writing your own routines for pixel encoding. This document assumes that you are familiar with the E&S GSRs. For a description of the GSRs, refer to Section *RM4*. The routines are listed alphabetically and the raster routines all begin with "PRA."

Table 14-1 lists the mode commands, the result of using the commands, and the GSR calls that implement the command.

Following the table is an alphabetical list of the routines, their parameters, and a brief description.

Programming examples in FORTRAN and Pascal follow the list of the raster routines.

Table 14-1. Commands in WRPIX Mode

MODE COMMAND	RESULT	GSR
	Set Raster Mode	
Set Raster Mode to Write Pixel Data	Sets raster mode to write pixel data.	PRAWRP
	WRPIX Mode -- Establish Operating Parameters	
Set Logical Device Coordinates	Positions the picture in virtual address space	PRASLD
Set Current Pixel Location	Establishes the current pixel location in the Logical Device Coordinates where the next WRPIX mode command begins	PRASCP
	WRPIX Mode -- Change Raster Picture	
Erase Screen	Fills all of pixel memory to one address in the CLUTs	PRASER
Load Pixel Data	Writes specific values to specific pixels	PRASWP

4.1 List of Raster Graphics Support Routines

The following list provides the names of the routines, expected parameters, and brief descriptions. Refer to Section *RM4* for a more detailed description.

Name of Routine and Parameters	Description
PRASCP (x,y, error routine)	Establishes current pixel location relative to the logical device coordinates.
PRASER (color, error routine	Erases the entire raster screen.
PRASLD (xmin, ymin,xmax,ymax, routine)	Sets the logical device error coordinates used to position the picture in virtual address space.
PRASWP (num, pixval, error routine)	Loads current pixel location with pixel values.
PRAWRP (error routine)	Sets raster mode to write pixel data.

4.2 FORTRAN GSR Raster Programming Example

This programming example uses the GSRs to build a "tricolor" flag display surrounded by a 20-pixel-wide blank border.

Program Example

```
C
  EXTERNAL  ERR
   INTEGER*4 MAT(4,10), BACK(3)
C
   CALL Pattch ('Logdevnam=tt:/Phydevtyp=Async', Err)
C
C  ERASE SCREEN TO BLACK
C
   BACK(1) = 0
   BACK(2) = 0
   BACK(3) = 0
   CALL PRASER( BACK, ERR )
```

```
C
C  PUT ON RED RECTANGLE
C
   CALL PRASLD( 20, 20, 219, 459, ERR)
C
   MAT(1,1) = 200 * 440
   MAT(2,1) = 255
   MAT(3,1) = 0
   MAT(4,1) = 0
   CALL PRASCP( 0, 0, ERR )
   CALL PRASWP( 1, MAT, ERR )
C
C
C  PUT ON WHITE RECTANGLE
C
   CALL PRASLD( 220, 20, 419, 459, ERR)
C
   MAT(1,1) = 200 * 440
   MAT(2,1) = 255
   MAT(3,1) = 255
   MAT(4,1) = 255
   CALL PRASCP( 0, 0, ERR )
   CALL PRASWP( 1, MAT, ERR )
C
C
C  PUT ON BLUE RECTANGLE
C
   CALL PRASLD( 420, 20, 619, 459, ERR)
C
   MAT(1,1) = 200 * 440
   MAT(2,1) = 0
   MAT(3,1) = 0
   MAT(4,1) = 255
   CALL PRASCP( 0, 0, ERR )
   CALL PRASWP( 1, MAT, ERR )
C
   CALL  PDTACH ( err )
   STOP
   END
```

4.3 Pascal GSR Constant Declarations

The following definitions are provided for the Pascal Raster GSRs:

```
P_MaxRunClrSize = User-specified maximum run-length color array
P_ColorType     = RECORD
       Red      : INTEGER;
       Green    : INTEGER;
       Blue     : INTEGER;
END;
P_RunColorType = RECORD
       Count    : INTEGER
       Red      : INTEGER;
       Green    : INTEGER;
       Blue     : INTEGER;
END;
P_RunClrArrayType = ARRAY [1..P_MaxRunClrSize] OF P_RunColorType;
```

4.4 Pascal GSR Raster Programming Example

```
PROGRAM EXAMPLE  (input, output);
CONST
     %INCLUDE 'PROCONST.PAS'
TYPE
     %INCLUDE 'PROTYPES.PAS'
VAR
     MAT : P_RunClrArrayType;
     BACK : P_ColorType;
     %INCLUDE 'PROEXTRN.PAS'
PROCEDURE Error_handler( err : INTEGER );
     BEGIN
     Writeln(' Error received : ', err );
     END;

BEGIN

Pattach ('Logdevnam=tt:/Phydevtyp=Async', Error_Handler );

{ ERASE SCREEN TO BLACK  }

BACK.red   := 0;
BACK.green := 0;
BACK.blue  := 0;
PRASER( BACK, Error_Handler );
```

```
{ PUT ON RED RECTANGLE  }

PRASLD( 20, 20, 219, 459, Error_Handler );

MAT[1].count := 200 * 440;
MAT[1].Red   := 255;
MAT[1].Green := 0;
MAT[1].Blue  := 0;
PRASCP( 0, 0, Error_Handler );
PRASWP( 1, MAT, Error_Handler );

{ PUT ON WHITE RECTANGLE }

PRASLD( 220, 20, 419, 459, Error_Handler );
MAT[1].count := 200 * 440;
MAT[1].Red   := 255;
MAT[1].Green := 255;
MAT[1].Blue  := 255;
PRASCP( 0, 0, Error_Handler );
PRASWP( 1, MAT, Error_Handler );

{ PUT ON BLUE RECTANGLE  }

PRASLD( 420, 20, 619, 459, Error_Handler );
MAT[1].count := 200 * 440;
MAT[1].Red   := 0;
MAT[1].Green := 0;MAT[1].Blue  := 255;
PRASCP( 0, 0, Error_Handler );
PRASWP( 1, MAT, Error_Handler );

Pdetach ( Error_Handler );
END.
```

5. Formatting Raster Commands for User-Generated Host Routines

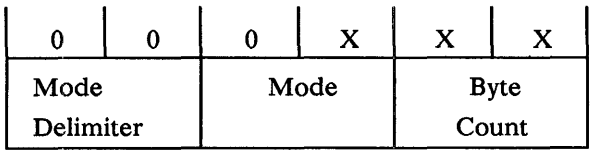
Communications between the host and the PS 390 use binary data transmission protocols. If you are writing your own host routines, you must:

- Ensure that the image buffer is in the correct mode (WRPIX).
- Ensure that all data (including the mode delimiter) are transferred out of the intrinsic user function CIROUTE using routing byte B, which sends the data out port 21.

Mode is specified by the following decimal value:

```
WRPIX = 0
```

Raster commands are strings of data that follow this format:



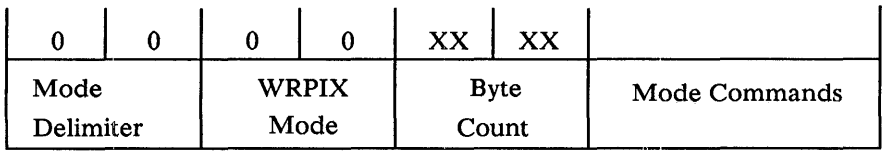
Commands

The "Mode Delimiter" is two bytes of 0 (0000000000000000), and must precede a new mode specification. No delimiter needs to follow the final mode specification.

The "Mode" is the sixteen-bit binary value for WRPIX. This determines the way the data that follow are to be interpreted. If WRPIX is specified, the information that follows the byte count is interpreted as pixel data.

The "Byte Count" determines how many of the bytes of data that follow the byte count are to be interpreted as commands in the specified mode. Until a new mode delimiter is set, all data are interpreted as being in the currently specified mode. Multiple sets of byte count and data may be sent without changing modes.

The WRPIX commands are described below:



NOTE

The maximum recommended byte count is 512 bytes.

5.1 Write Pixel Data Mode

Pixel information can be transmitted from the host to the raster display in a run-length encoding scheme.

For example, a "Load Pixel Data" command for 1-127 consecutive pixels has the following format (each character represents one bit):

```
| 0nnnnnnn | RRRRRRRR | GGGGGGGG | BBBBBBBB |
+ -------- + -------- + -------- + -------- +
31                                          0
```

Where "n" specifies the number of consecutive pixels, and "R-G-B" specifies the lookup table addresses for red, green, and blue.

Load Pixel Data for 128-16383 consecutive pixels command:

```
| 10nnnnnn | nnnnnnnn | RRRRRRRR | GGGGGGGG | BBBBBBBB |
+ -------- + -------- + -------- + -------- + -------- +
47                            31                      0
```

where "n" specifies the number of consecutive pixels and "R-G-B" specifies the lookup table addresses for red, green, and blue.

The current pixel location can be explicitly set by the "Set X-Y Position" command and is used to specify the current pixel location where the "Load Pixel Data" command will begin writing pixels. The pixel location is set relative to the values (Xmin, Ymin) of the logical device coordinates. If the logical device coordinates are $-1024 <= X <= 1024$ and $-1024 <= Y <= 1024$, then an X,Y position of (0,0) is the lower left-hand corner pixel and (2047,2047) is the upper right-hand corner pixel.

Set Current Pixel Location command:

```
| 110xxxxx | xxxxxxxx | 000yyyyy | yyyyyyyy |
+ -------- + -------- + -------- + -------- +
31                                          0
```

The entire pixel memory may be set to the same value with one command. This will erase the entire screen to the color in the specified lookup table locations.

Erase Screen command:

```
| 11100000 | RRRRRRRR | GGGGGGGG | BBBBBBBB |
+ -------- + -------- + -------- + -------- +
31                                          0
```

Logical device coordinates are specified by the Set Logical Device Coordinates command.

Set Logical Device Coordinates command:

```
| 11110XXX | XXXXXXXX | xxxxxxxx | xxxxxxxx | Where X = X maximum
+ -------- + -------- + -------- + -------- +       x = X minimum
  64                                    32

| 0000YYYY | YYYYYYYY | yyyyyyyy | yyyyyyyy | Where Y = Y maximum
+ -------- + -------- + -------- + -------- +       y = Y minimum
31                                          0
```

Restrictions:

```
x  =  twos complement integer        -32768   < = x < = 1023
X  =  unsigned integer                   0    < = X < = 2047
y  =  twos complement integer        -32768   < = y < = 863
Y  =  unsigned integer                   0    < = Y < = 2047
x < = X and y < = Y
```

5.2 Programming Example for User-Generated Host Routines

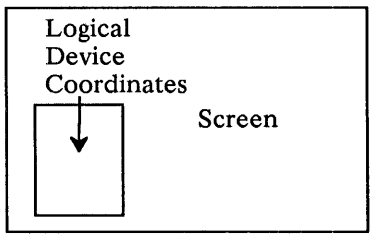
This programming example describes how to build a "tricolor" flag display surrounded by a 20-pixel-wide blank border. Numbers are specified in hexadecimal.

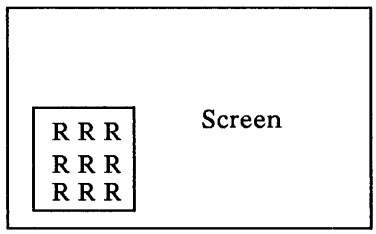
WRPIX mode is specified to:

1. Erase Screen.
2. Set Logical Device Coordinates.
3. Set Current Pixel Location.
4. Load Pixel Data.

WRPIX mode (0000) is set and 4 bytes of commands are specified. An Erase Screen command is sent.

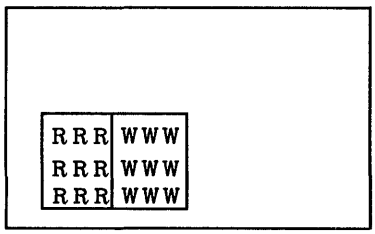
```
0000          ( Set WRPIX mode )
04            ( 4 bytes of commands )
E0000000      ( Erase Screen r = 0, g = 0, b = 0 )
```





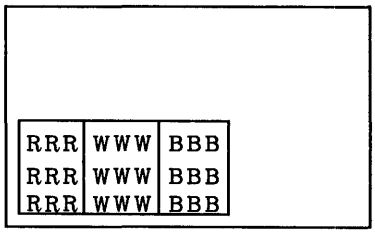

```
30                ( 48 bytes of commands )
WRPIX
F0DB001401CB0014  ( Set LDC 20 <= x <= 219, 20 <= y <= 459: Sets LDCs
                    to the left one-third of the screen )
WRPIX             ( Fill LDC area with red )
C0000000          ( Set Current Location X = 0, Y = 0 )
BFFF00FF0000      ( 16383 pixels red )
BFFF00FF0000      ( 16383 pixels red )
BFFF00FF0000      ( 16383 pixels red )
BFFF00FF0000      ( 16383 pixels red )
BFFF00FF0000      ( 16383 pixels red )
97C500FF0000      (  6085 pixels red )
```

WRPIX (Fill new LDC with white)




```
30                  ( 48 bytes of commands )
F1A300DC01CB0014    ( Set LDC 220 <= x <= 419, 20 <= y <= 459 )
C0000000            ( Set Current Location X = 0, Y = 0 )
BFFF00FFFFFF        ( 16383 pixels white )
BFFF00FFFFFF        ( 16383 pixels white )
BFFF00FFFFFF        ( 16383 pixels white )
BFFF00FFFFFF        ( 16383 pixels white )
BFFF00FFFFFF        ( 16383 pixels white )
97C500FFFFFF        (  6085 pixels white )
```

WRPIX (Fill new LDC with blue)




```
30                  ( 48 bytes of commands )
F26B01A401CB0014    ( Set LDC 420 <= x <= 619, 20 <= y <= 459 )
C0000000            ( Set Current Location X = 0, Y = 0 )
BFFF000000FF        ( 16383 pixels blue )
BFFF000000FF        ( 16383 pixels blue )
BFFF000000FF        ( 16383 pixels blue )
BFFF000000FF        ( 16383 pixels blue )
BFFF000000FF        ( 16383 pixels blue )
97C5000000FF        ( 6085  pixels blue)
```

(End of Example)

GT15. SAMPLE PROGRAMS

GT15. SAMPLE PROGRAMS

CONTENTS

ILLUSTRATIONS

Section GT15
Sample Programs

The sample programs in this section illustrate various applications of the PS 390 for design and analysis. A program with a .300 extension is a data structure file, and a program with a .FUN extension is a function network file. A header section in each file explains what the application does. General practices illustrated in the sample programs can give you ideas for your own application programs.

A great deal of care has been taken to make these programs examples of good PS 390 programming practices. In the data structure files, notice particularly the use of BEGIN_STRUCTURE ... END_STRUCTURE versus explicit naming. Notice also that the code is tabbed and commented in a way that makes it very easy to read.

The sample programs are listed in this section and also distributed in loadable form on magnetic tape. A selection in the command file TUTORIALS.COM lets you load the sample programs individually from the host.

1. Articulated Anthropoid Robot Example

1.1. ADAM.300

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: April 21, 1983
Last update:

Data Structure for an articulated anthropoid robot called ADAM (A Dial Activated Man). The data nodes (vector lists) for the sphere and the cylinder are not included in this file. The sphere has a radius of 1 and is centered at the origin. The base of the cylinder is at the origin lying in the XZ plane with the cylinder centered about the positive Y axis. The cylinder has a radius of 1 and a height of 1.

ADAM.FUN is the function network file that will articulate this structure.

```
INIT DISP;
DISP Adam;

Adam := BEGIN_S
          WINDOW X=-8.5:8.5 Y=-8.5:5.5
            FRONT=0 BACK=10;
          LOOK AT 0,0,0 FROM 0,0,-1;
Tran :=   TRAN 0,0,0;
Rot :=    ROT Y 0;
Scale :=  SCALE 1;
2Pick :=  SET PICKING OFF;
          INST Upper_Body,Lower_Body;
        END_S;

Upper_Body := BEGIN_S
                SET PICK ID = B;
Rot :=          ROT 0;
{Chest}       SCALE .8,2.4,.7 THEN Cylinder;
                INST Right_Arm,Left_Arm,Head;
              END_S;

Right_Arm  := BEGIN_S
                TRAN -1.15,2.4,0;
{ Right Shoulder Joint }
                SET PICK ID = C;
Rot :=          ROT 0;
                 INST Upper_Arm,Right_Lower_Arm;
              END_S;

Upper_Arm  := BEGIN_S
{Shoulder Ball} SCALE .3,.2,.2 THEN Sphere;
                TRAN 0,-2.1,0;
                SCALE .25,2.1,.25 THEN Cylinder;
              END_S;

Right_Lower_Arm := BEGIN_S
                     TRAN 0,-2.2,0;
Rot :=             ROT 0;
                     INST Lower_Arm,Right_Hand;
                   END_S;

Lower_Arm  := BEGIN_S
{Elbow}        SCALE .219 THEN Sphere; {7/32 rad.}
                TRAN 0,-1.8,0;
                SCALE .225,1.7,.225 THEN Cylinder;
              END_S;
```

```
Right_Hand := BEGIN_S
                TRAN 0,-1.9,0;
                SET PICK ID = D;
Rot :=          ROT 0 THEN Hand;
              END_S;

Hand := BEGIN_S
{Wrist}   SCALE .175 THEN Sphere;
{Hand}    TRAN 0,-.4,0;
          SCALE .15,.4,.25 THEN Sphere;
        END_S;

Left_Arm := BEGIN_S
              TRAN 1.15,2.4,0;
              SET PICK ID = C;
Rot :=        ROT 0;
              INST Upper_Arm,Left_Lower_Arm;
            END_S;

Left_Lower_Arm  := BEGIN_S
                     TRAN 0,-2.2,0;
Rot :=               ROT 0;
                     INST Lower_Arm,Left_Hand;
                   END_S;

Left_Hand := BEGIN_S
               TRAN 0,-1.9,0;
               SET PICK ID = D;
Rot :=         ROT 0 THEN Hand;
             END_S;

Head := BEGIN_S
          TRAN 0,2.4,0;
          SET PICK ID = A;
Rot :=    ROT 0;
{Neck}    SCALE .3,.6,.3 THEN Cylinder;
{Head}    TRAN 0,1.5,0;
          SCALE .6,1,.6 THEN Sphere;
        END_S;

Lower_Body := BEGIN_S
                SET PICK ID = B;
Rot :=          ROT 0;
                TRAN 0,-1,0;
                INST Right_Leg,Left_Leg;
{Waist & Hips}  SCALE .8,1,.7 THEN Cylinder;
              END_S;
```

```
Right_Leg := BEGIN_S
               TRAN -.45,-.25;
               SET PICK ID = E;
Rot :=         ROT 0;
               INST Upper_Leg,Right_Lower_Leg;
             END_S;

Upper_Leg := BEGIN_S
{Hip Joint}    SCALE .3 THEN Sphere;
               TRAN 0,-2.5,0;
               SCALE .35,2.5,.35 THEN Cylinder;
             END_S;

Right_Lower_Leg := BEGIN_S
                     TRAN 0,-2.6,0;
Rot :=               ROT x 0;
                     INST Lower_Leg,Right_Foot;
                   END_S;

Lower_Leg  := BEGIN_S
                INST Knee;
                TRAN 0,-2.6,0;
{Limb}          SCALE .3,2.5,.3 THEN Cylinder;
              END_S;

Knee := BEGIN_S
          ROT 90;
          TRAN 0,-.3,0;
          SCALE .15,.6,.15 THEN Cylinder;
        END_S;

Right_Foot := BEGIN_S
                TRAN 0,-2.75,0;
                SET PICK ID = F;
Rot :=          ROT 0 THEN Foot;
              END_S;

Foot := BEGIN_S
{Ankle}   SCALE .2 THEN Sphere;
          TRAN 0,-.2,.2;
          ROT x -90;
          SCALE .3,1,.2 THEN Cylinder;
        END_S;

Left_Leg := BEGIN_S
              TRAN .45,-.25;
              SET PICK ID = E;
Rot :=        ROT 0;
              INST Upper_Leg,Left_Lower_Leg;
            END_S;
```

```
Left_Lower_Leg := BEGIN_S
                    TRAN 0,-2.6,0;
Rot :=              ROT x 0;
                    INST Lower_Leg,Left_Foot;
                  END_S;

Left_Foot := BEGIN_S
               TRAN 0,-2.75,0;
               SET PICK ID = F;
Rot :=         ROT 0 THEN Foot;
             END_S;
```

1.2. ADAM.FUN

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: October, 1982
Last update: February, 1985

Network to modify the structure in ADAM.300. Point at the joint you want to rotate and the dials will be routed to modify that joint and others associated in that mode. If you want to rotate and translate the whole robot, point at the head.

```
{ Code generated by Network Editor 1.07 }
{ ADAM }
{ Frame-Prefix Macro-Prefix  }
{ Frame2:F2_ }
F2_P4:=F:CROUTE(6);
F2_P5:=F:CROUTE(6);
F2_P6:=F:DXROTATE;
F2_P7:=F:DXROTATE;
F2_P8:=F:DXROTATE;
F2_P9:=F:DXROTATE;
CONN F2_P4<3>:<1>F2_P6;
CONN F2_P4<5>:<1>F2_P7;
CONN F2_P5<3>:<1>F2_P8;
CONN F2_P5<5>:<1>F2_P9;
CONN F2_P6<1>:<1>Right_Lower_Arm.Rot;
CONN F2_P7<1>:<1>Right_Lower_Leg.Rot;
CONN F2_P8<1>:<1>Left_Lower_Arm.Rot;
CONN F2_P9<1>:<1>Left_Lower_Leg.Rot;
SEND 200 TO <3>F2_P7;
SEND 200 TO <3>F2_P8;
```

```
SEND 200 TO <3>F2_P9;
SEND 200 TO <3>F2_P6;
SEND 0 TO <2>F2_P7;
SEND 0 TO <2>F2_P8;
SEND 0 TO <2>F2_P9;
SEND 0 TO <2>F2_P6;
{ Frame3:F3_ }
F3_P11:=F:MULC;
F3_P12:=F:MULC;
F3_P13:=F:MULC;
F3_P14:=F:XROTATE;
F3_P15:=F:YROTATE;
F3_P16:=F:ZROTATE;
F3_P17:=F:CROUTE(6);
F3_P18:=F:MULC;
F3_P19:=F:MULC;
F3_P20:=F:MULC;
F3_P21:=F:MULC;
F3_P22:=F:MULC;
F3_P23:=F:MULC;
CONN F3_P11<1>:<1>F3_P14;
CONN F3_P12<1>:<1>F3_P15;
CONN F3_P13<1>:<1>F3_P16;
CONN F3_P14<1>:<2>F3_P17;
CONN F3_P15<1>:<2>F3_P17;
CONN F3_P16<1>:<2>F3_P17;
CONN F3_P17<1>:<1>F3_P18;
CONN F3_P17<2>:<1>F3_P19;
CONN F3_P17<3>:<1>F3_P20;
CONN F3_P17<4>:<1>F3_P21;
CONN F3_P17<5>:<1>F3_P22;
CONN F3_P17<6>:<1>F3_P23;
CONN F3_P18<1>:<1>Head.Rot;
CONN F3_P18<1>:<2>F3_P18;
CONN F3_P19<1>:<1>Upper_Body.Rot;
CONN F3_P19<1>:<2>F3_P19;
CONN F3_P20<1>:<1>Right_Arm.Rot;
CONN F3_P20<1>:<2>F3_P20;
CONN F3_P21<1>:<1>Right_Hand.Rot;
CONN F3_P21<1>:<2>F3_P21;
CONN F3_P22<1>:<1>Right_Leg.Rot;
CONN F3_P22<1>:<2>F3_P22;
CONN F3_P23<1>:<1>Right_Foot.Rot;
CONN F3_P23<1>:<2>F3_P23;
SEND 200 TO <2>F3_P11;
SEND 200 TO <2>F3_P12;
SEND 200 TO <2>F3_P13;
```

```
{ Frame4:F4_ }
F4_P24:=F:MULC;
F4_P25:=F:MULC;
F4_P26:=F:MULC;
F4_P27:=F:XROTATE;
F4_P28:=F:YROTATE;
F4_P29:=F:ZROTATE;
F4_P30:=F:CROUTE(6);
F4_P31:=F:CMUL;
F4_P32:=F:MULC;
F4_P33:=F:MULC;
F4_P34:=F:MULC;
F4_P35:=F:MULC;
F4_P36:=F:MULC;
CONN F4_P24<1>:<1>F4_P27;
CONN F4_P25<1>:<1>F4_P28;
CONN F4_P26<1>:<1>F4_P29;
CONN F4_P27<1>:<2>F4_P30;
CONN F4_P28<1>:<2>F4_P30;
CONN F4_P29<1>:<2>F4_P30;
CONN F4_P30<1>:<2>F4_P31;
CONN F4_P30<2>:<1>F4_P32;
CONN F4_P30<3>:<1>F4_P33;
CONN F4_P30<4>:<1>F4_P34;
CONN F4_P30<5>:<1>F4_P35;
CONN F4_P30<6>:<1>F4_P36;
CONN F4_P31<1>:<1>Adam.Rot;
CONN F4_P31<1>:<1>F4_P31;
CONN F4_P32<1>:<1>Lower_Body.Rot;
CONN F4_P32<1>:<2>F4_P32;
CONN F4_P33<1>:<1>Left_Arm.Rot;
CONN F4_P33<1>:<2>F4_P33;
CONN F4_P34<1>:<1>Left_Hand.Rot;
CONN F4_P34<1>:<2>F4_P34;
CONN F4_P35<1>:<1>Left_Leg.Rot;
CONN F4_P35<1>:<2>F4_P35;
CONN F4_P36<1>:<1>Left_Foot.Rot;
CONN F4_P36<1>:<2>F4_P36;
SEND 200 TO <2>F4_P25;
SEND 200 TO <2>F4_P26;
SEND 200 TO <2>F4_P24;
{ Picking Network:F5_ }
F5_P3:=F:PICKINFO;
F5_P39:=F:CHARCONVERT;
F5_P40:=F:SUBC;
CONN TABLETIN<4>:<1>Adam.Pick;
CONN TABLETIN<6>:<1>PICK;
```

```
CONN PICK<1>:<1>F5_P3;
CONN PICK<2>:<1>Adam.Pick;
CONN PICK<3>:<1>Adam.Pick;
CONN F5_P3<2>:<1>F5_P39;
CONN F5_P39<1>:<1>F5_P40;
SEND FIX(64) TO <2>F5_P40;
SEND FIX(1) TO <2>F5_P3;
{ Frame1:F1_ }
{ Setup cness true <2-3>P10 }
F1_P10:=F:SYNC(3);
SETUP CNESS TRUE <2>F1_P10;
SETUP CNESS TRUE <3>F1_P10;
CONN F1_P10<2>:<2>F2_P6;
CONN F1_P10<2>:<2>F2_P7;
CONN F1_P10<2>:<2>F2_P8;
CONN F1_P10<2>:<2>F2_P9;
CONN F1_P10<3>:<1>Right_Lower_Arm.Rot;
CONN F1_P10<3>:<1>Right_Lower_Leg.Rot;
CONN F1_P10<3>:<1>Left_Lower_Arm.Rot;
CONN F1_P10<3>:<1>Left_Lower_Leg.Rot;
CONN F1_P10<3>:<2>F3_P18;
CONN F1_P10<3>:<2>F3_P19;
CONN F1_P10<3>:<2>F3_P20;
CONN F1_P10<3>:<2>F3_P21;
CONN F1_P10<3>:<2>F3_P22;
CONN F1_P10<3>:<2>F3_P23;
CONN F1_P10<3>:<1>Head.Rot;
CONN F1_P10<3>:<1>Upper_Body.Rot;
CONN F1_P10<3>:<1>Right_Arm.Rot;
CONN F1_P10<3>:<1>Right_Hand.Rot;
CONN F1_P10<3>:<1>Right_Leg.Rot;
CONN F1_P10<3>:<1>Right_Foot.Rot;
CONN F1_P10<3>:<1>F4_P31;
CONN F1_P10<3>:<2>F4_P32;
CONN F1_P10<3>:<2>F4_P33;
CONN F1_P10<3>:<2>F4_P34;
CONN F1_P10<3>:<2>F4_P35;
CONN F1_P10<3>:<2>F4_P36;
CONN F1_P10<3>:<1>Adam.Rot;
CONN F1_P10<3>:<1>Lower_Body.Rot;
CONN F1_P10<3>:<1>Left_Arm.Rot;
CONN F1_P10<3>:<1>Left_Hand.Rot;
CONN F1_P10<3>:<1>Left_Leg.Rot;
CONN F1_P10<3>:<1>Left_Foot.Rot;
CONN FKEYS<1>:<1>F1_P10;
CONN DIALS<1>:<1>F3_P11;
CONN DIALS<2>:<1>F3_P12;
```

```
CONN DIALS<3>:<1>F3_P13;
CONN DIALS<4>:<2>F2_P4;
CONN DIALS<5>:<1>F4_P24;
CONN DIALS<6>:<1>F4_P25;
CONN DIALS<7>:<1>F4_P26;
CONN DIALS<8>:<2>F2_P5;
CONN F5_P40<1>:<1>F2_P4;
CONN F5_P40<1>:<1>F2_P5;
CONN F5_P40<1>:<1>F3_P17;
CONN F5_P40<1>:<1>F4_P30;
SEND FIX(1) TO <1>F2_P4;
SEND FIX(1) TO <1>F2_P5;
SEND FIX(1) TO <1>F3_P17;
SEND FIX(1) TO <1>F4_P30;
SEND 0 TO <2>F1_P10;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <3>F1_P10;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>F1_P10;
```

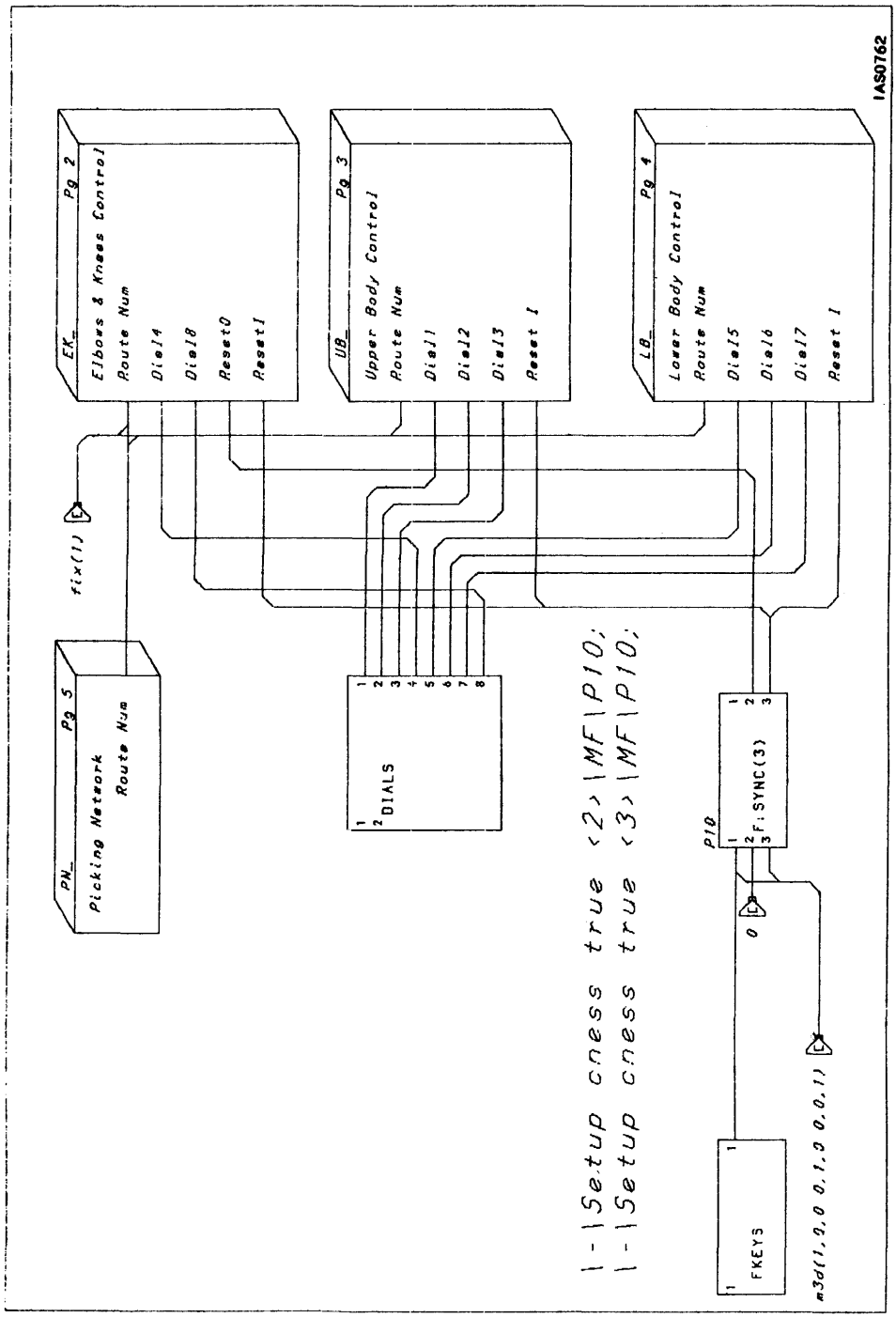


Figure 15-1. ADAM.FUN (Sheet 1 of 5)
(Function Network for ADAM.300)

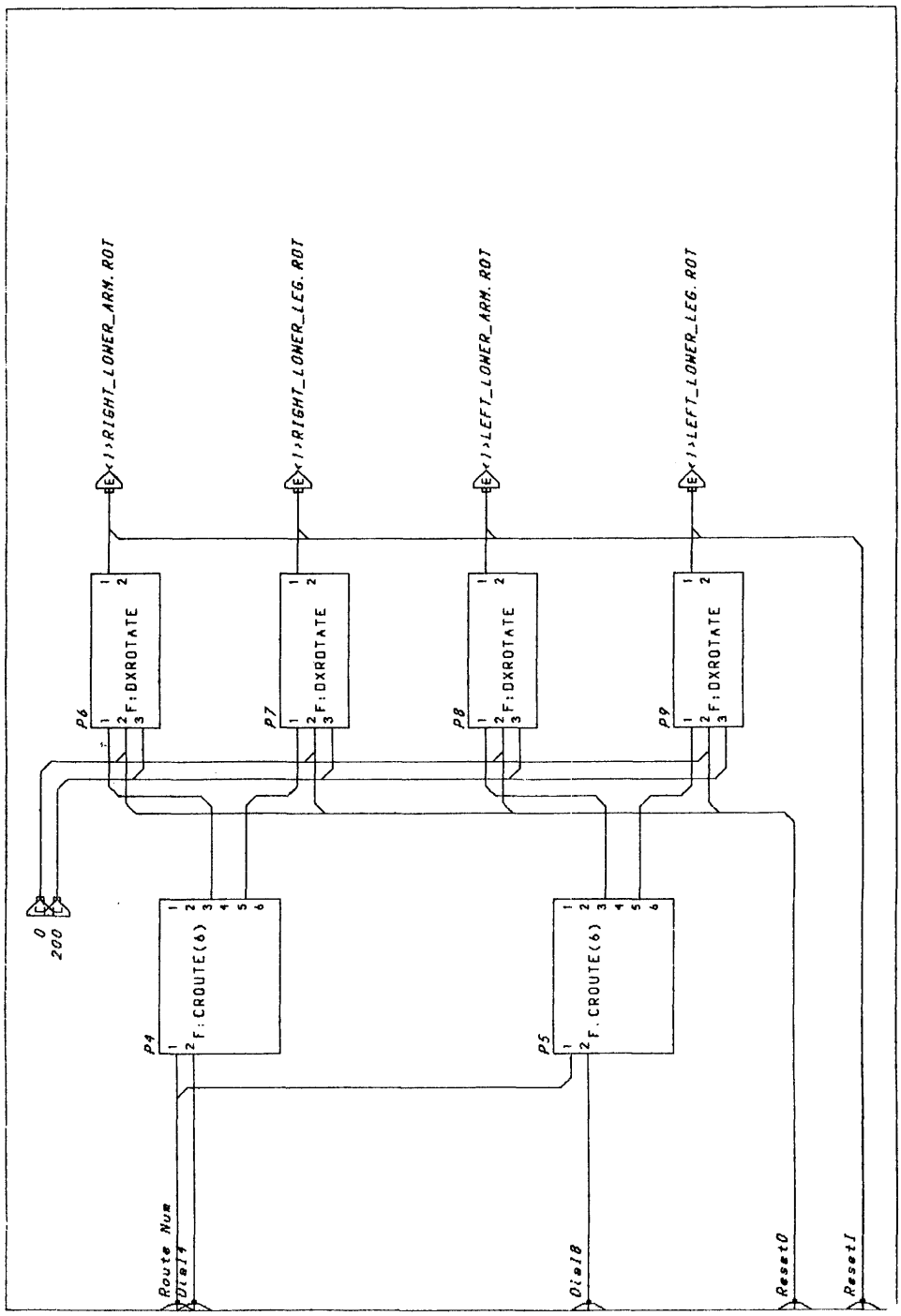


Figure 15-1. ADAM.FUN (Sheet 2 of 5)

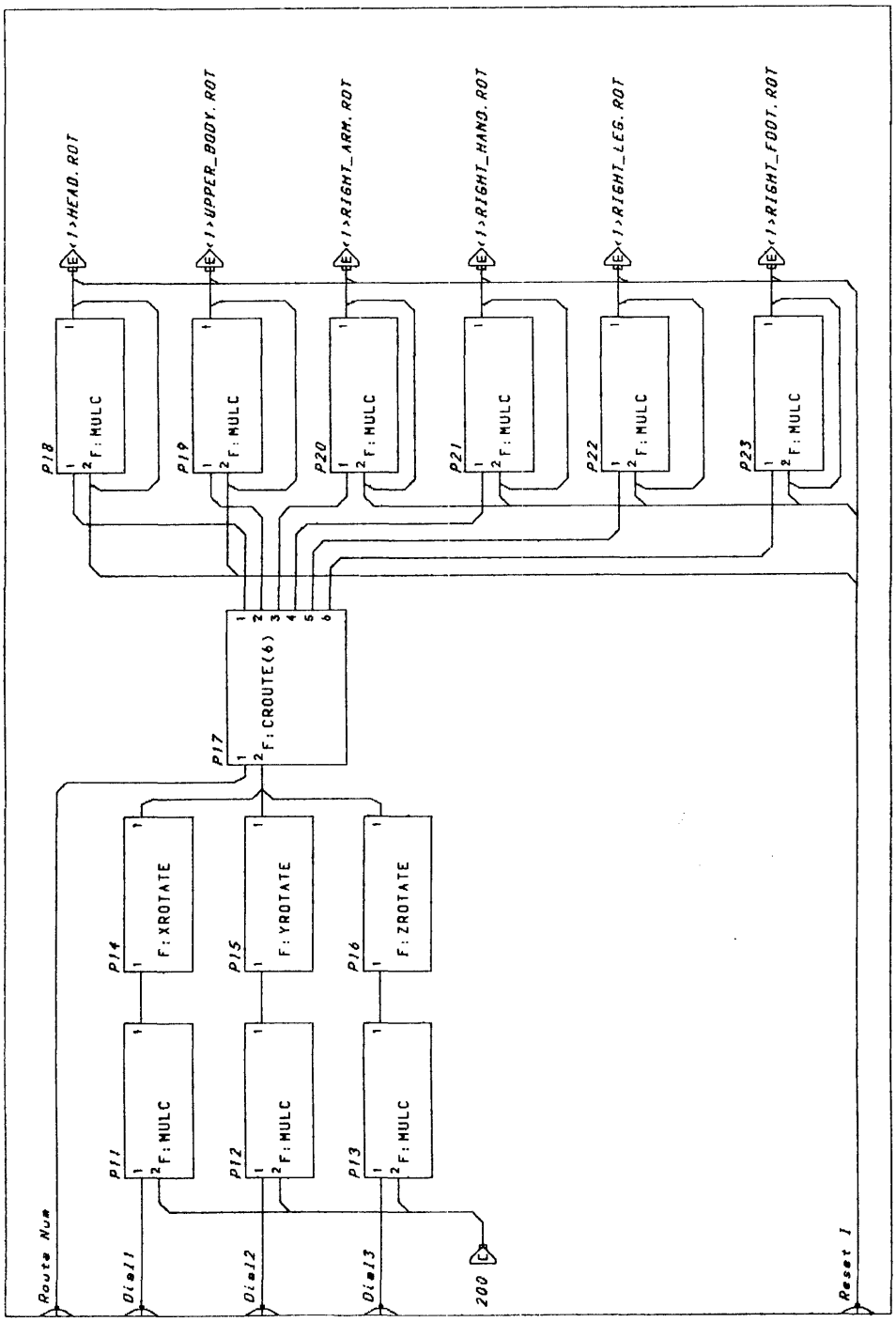


Figure 15-1. ADAM.FUN (Sheet 3 of 5)

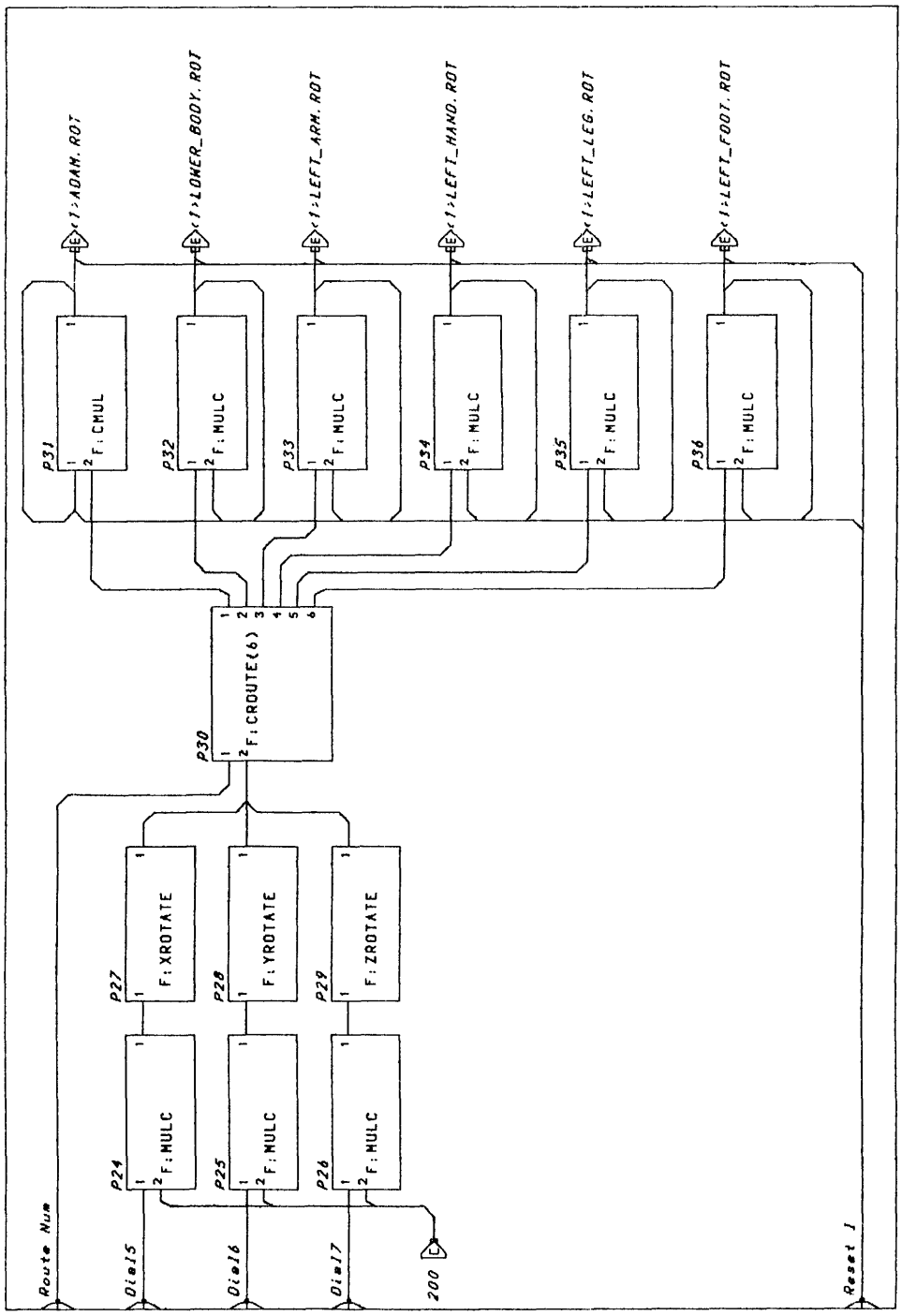


Figure 15-1. ADAM.FUN (Sheet 4 of 5)

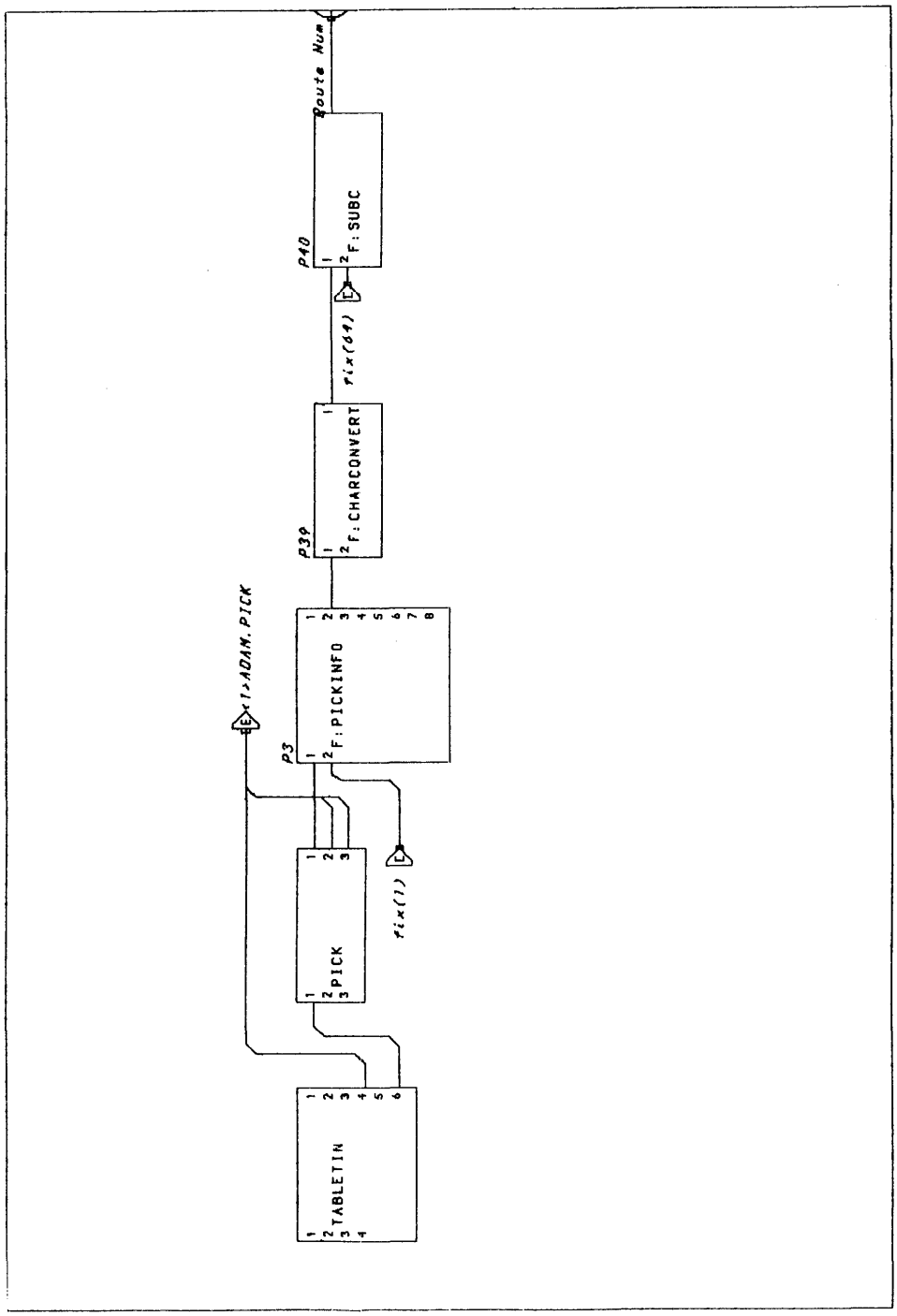


Figure 15-1. ADAM.FUN (Sheet 5 of 5)

2. Bouncing Ball Example

2.1. COLLISION.300

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: October, 1984
Last update: February, 1985

PS 390 data structure, consisting of a ball in a box. The function network in COLLISION.FUN modifies this structure to simulate the ball bouncing in the box with no gravity and elastic collisions.

```
INIT DISP;
DISP Collision;

Collision := BEGIN_S
               SET INTENSITY ON .75:1;
               SET DEPTH_CLIPPING ON;
               FOV 70 FRONT = 1.4 BACK = 5;
               LOOK AT 0,0,0 FROM 1.5,1.3,-2.4;
Yrot :=        ROT 0;
               SET COLOR 240,1 THEN Box;
               SET COLOR 120,1 THEN Ball;
               SET COLOR 0,1 THEN Path;
             END_S;

Box := SCALE 1 THEN Cube;

Ball := BEGIN_S
Tran :=   TRAN 0,0,0;
Rot :=    ROT 0;
Scale :=  SCALE .1 THEN Sphere;
        END_S;

Path := VEC n=10000 0,0,0;

Cube := VEC Item n=16
           P -1, 1,-1   L  1, 1,-1
           L  1,-1,-1   L -1,-1,-1
           P  1, 1,-1   L  1, 1, 1
           L  1,-1, 1   L  1,-1,-1
           P  1, 1, 1   L -1, 1, 1
           L -1,-1, 1   L  1,-1, 1
           P -1, 1, 1   L -1, 1,-1
           L -1,-1,-1   L -1,-1, 1;
```

2.2. COLLISION.FUN

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: October, 1984
Last update: February, 1985

Network to modify structure created in COLLISION.300. See description in that file.

```
{ Code generated by Network Editor 1.07 }
{ COLLISION }
{ Frame-Prefix Macro-Prefix  }
{ Frame1:M1$F1_ }
M1$F1_P1:=F:INPUTS_CHOOSE(13);
M1$F1_P2:=F:ROUTE(12);
CONN M1$F1_P1<1>:<2>M1$F1_P2;
SEND TRUE TO <1>M1$F1_P1;
SEND TRUE TO <2>M1$F1_P1;
SEND TRUE TO <3>M1$F1_P1;
SEND TRUE TO <4>M1$F1_P1;
SEND TRUE TO <5>M1$F1_P1;
SEND TRUE TO <6>M1$F1_P1;
SEND TRUE TO <7>M1$F1_P1;
SEND TRUE TO <8>M1$F1_P1;
SEND TRUE TO <9>M1$F1_P1;
SEND TRUE TO <10>M1$F1_P1;
SEND TRUE TO <11>M1$F1_P1;
SEND TRUE TO <12>M1$F1_P1;
{ Motion Control:F2_ }
F2_P2:=F:SYNC(4);
F2_P6:=F:LIMIT;
F2_P7:=F:LIMIT;
F2_P8:=F:LIMIT;
F2_P9:=F:BROUTEC;
F2_P10:=F:BROUTEC;
F2_P11:=F:BROUTEC;
F2_P12:=F:MULC;
F2_P13:=F:MULC;
F2_P14:=F:MULC;
F2_P15:=F:XVECTOR;
F2_P16:=F:YVECTOR;
F2_P17:=F:ZVECTOR;
F2_P18:=F:ADD;
```

```
F2_P19:=F:ADD;
F2_P20:=F:ADD;
F2_P41:=F:ACCUMULATE;
F2_P42:=F:ACCUMULATE;
F2_P43:=F:ACCUMULATE;
F2_P38:=F:ADD;
F2_P39:=F:ADD;
CONN F2_P2<2>:<1>F2_P18;
CONN F2_P2<3>:<1>F2_P19;
CONN F2_P2<4>:<1>F2_P20;
CONN F2_P6<1>:<1>F2_P15;
CONN F2_P6<1>:<2>F2_P18;
CONN F2_P6<3>:<1>F2_P9;
CONN F2_P7<1>:<1>F2_P16;
CONN F2_P7<1>:<2>F2_P19;
CONN F2_P7<3>:<1>F2_P10;
CONN F2_P8<1>:<1>F2_P17;
CONN F2_P8<1>:<2>F2_P20;
CONN F2_P8<3>:<1>F2_P11;
CONN F2_P9<1>:<2>F2_P2;
CONN F2_P9<2>:<1>F2_P12;
CONN F2_P10<1>:<3>F2_P2;
CONN F2_P10<2>:<1>F2_P13;
CONN F2_P11<1>:<4>F2_P2;
CONN F2_P11<2>:<1>F2_P14;
CONN F2_P12<1>:<2>F2_P2;
CONN F2_P12<1>:<2>F2_P9;
CONN F2_P12<1>:<2>F2_P41;
CONN F2_P13<1>:<3>F2_P2;
CONN F2_P13<1>:<2>F2_P10;
CONN F2_P13<1>:<2>F2_P42;
CONN F2_P14<1>:<4>F2_P2;
CONN F2_P14<1>:<2>F2_P11;
CONN F2_P14<1>:<2>F2_P43;
CONN F2_P15<1>:<1>F2_P38;
CONN F2_P16<1>:<2>F2_P38;
CONN F2_P17<1>:<2>F2_P39;
CONN F2_P18<1>:<1>F2_P6;
CONN F2_P19<1>:<1>F2_P7;
CONN F2_P20<1>:<1>F2_P8;
CONN F2_P38<1>:<1>F2_P39;
CONN F2_P39<1>:<1>Ball.Tran;
CONN F2_P41<1>:<2>F2_P9;
CONN F2_P42<1>:<2>F2_P10;
CONN F2_P43<1>:<2>F2_P11;
SEND -.9 TO <3>F2_P6;
SEND -.9 TO <3>F2_P7;
```

```
SEND -.9 TO <3>F2_P8;
SEND .9 TO <2>F2_P6;
SEND .9 TO <2>F2_P7;
SEND .9 TO <2>F2_P8;
SEND 0 TO <6>F2_P41;
SEND 0 TO <6>F2_P42;
SEND 0 TO <6>F2_P43;
SEND 10 TO <5>F2_P41;
SEND 10 TO <5>F2_P42;
SEND 10 TO <5>F2_P43;
SEND .1 TO <4>F2_P41;
SEND .1 TO <4>F2_P42;
SEND .1 TO <4>F2_P43;
SEND 0 TO <3>F2_P41;
SEND 0 TO <3>F2_P42;
SEND 0 TO <3>F2_P43;
SEND .03 TO <4>F2_P2;
SEND .03 TO <2>F2_P11;
SEND .03 TO <2>F2_P43;
SEND .02 TO <3>F2_P2;
SEND .02 TO <2>F2_P10;
SEND .02 TO <2>F2_P42;
SEND .01 TO <2>F2_P2;
SEND .01 TO <2>F2_P9;
SEND .01 TO <2>F2_P41;
SEND 0 TO <2>F2_P18;
SEND 0 TO <2>F2_P19;
SEND 0 TO <2>F2_P20;
SEND -1 TO <2>F2_P12;
SEND -1 TO <2>F2_P13;
SEND -1 TO <2>F2_P14;
{ Clock Control:F3_ }
F3_P1:=F:CLFRAMES;
F3_P22:=F:CONSTANT;
F3_P23:=F:EDGE_DETECT;
F3_P25:=F:ACCUMULATE;
F3_P27:=F:FIX;
F3_P28:=F:XOR;
F3_P65:=F:XROTATE;
CONN F3_P1<2>:<1>F3_P22;
CONN F3_P1<2>:<1>F3_P65;
CONN F3_P1<2>:<5>F3_P1;
CONN F3_P22<1>:<1>F3_P23;
CONN F3_P25<1>:<1>F3_P27;
CONN F3_P27<1>:<1>F3_P1;
CONN F3_P28<1>:<6>F3_P1;
CONN F3_P28<1>:<2>F3_P28;
```

```
CONN F3_P65<1>:<1>Ball.Rot;
SEND FIX(0) TO <2>F3_P1;
SEND FALSE TO <3>F3_P1;
SEND FIX(1) TO <4>F3_P1;
SEND FIX(0) TO <5>F3_P1;
SEND FALSE TO <6>F3_P1;
SEND FIX(1) TO <1>F3_P1;
SEND FALSE TO <1>F3_P23;
SEND TRUE TO <2>F3_P22;
SEND TRUE TO <2>F3_P23;
SEND 1 TO <2>F3_P25;
SEND 1 TO <3>F3_P25;
SEND 10 TO <4>F3_P25;
SEND 60 TO <5>F3_P25;
SEND 1 TO <6>F3_P25;
SEND FALSE TO <2>F3_P28;
{ Frame1:M2$F1_ }
{ Box Size }
M2$F1_P1:=F:ACCUMULATE;
M2$F1_P2:=F:XVECTOR;
M2$F1_P3:=F:YVECTOR;
M2$F1_P4:=F:ZVECTOR;
M2$F1_P5:=F:CONSTANT;
M2$F1_P6:=F:NOP;
CONN M2$F1_P2<1>:<1>M2$F1_P1;
CONN M2$F1_P3<1>:<1>M2$F1_P1;
CONN M2$F1_P4<1>:<1>M2$F1_P1;
CONN M2$F1_P5<1>:<2>M2$F1_P1;
SEND V3D(.01,.01,.01) TO <6>M2$F1_P1;
SEND 1 TO <4>M2$F1_P1;
SEND V3D(1,1,1) TO <2>M2$F1_P1;
SEND V3D(1,1,1) TO <2>M2$F1_P5;
SEND V3D(1,1,1) TO <5>M2$F1_P1;
SEND V3D(1,1,1) TO <1>M2$F1_P6;
SEND 0 TO <3>M2$F1_P1;
{ Box/Ball Size:F4_ }
F4_P31:=F:SUBC;
F4_P32:=F:SCALE;
F4_P33:=F:PARTS;
F4_P34:=F:PARTS;
F4_P35:=F:MULC;
F4_P44:=F:DSCALE;
F4_P45:=F:VEC;
F4_P46:=F:VEC;
F4_P47:=F:FETCH;
VAR Box_Size;
CONN M2$F1_P1<1>:<1>F4_P32;
```

```
CONN M2$F1_P1<1>:<1>F4_P31;
CONN M2$F1_P1<1>:<1>Box_Size;
CONN M2$F1_P5<1>:<1>F4_P32;
CONN M2$F1_P5<1>:<1>F4_P31;
CONN M2$F1_P5<1>:<1>Box_Size;
CONN M2$F1_P6<1>:<1>F4_P32;
CONN M2$F1_P6<1>:<1>F4_P31;
CONN M2$F1_P6<1>:<1>Box_Size;
CONN F4_P31<1>:<1>F4_P33;
CONN F4_P31<1>:<1>F4_P35;
CONN F4_P32<1>:<1>Box;
CONN F4_P35<1>:<1>F4_P34;
CONN F4_P44<1>:<1>Ball.Scale;
CONN F4_P44<2>:<3>F4_P44;
CONN F4_P44<2>:<1>F4_P45;
CONN F4_P44<2>:<2>F4_P45;
CONN F4_P44<2>:<2>F4_P46;
CONN F4_P45<1>:<1>F4_P46;
CONN F4_P46<1>:<1>F4_P47;
CONN F4_P46<1>:<2>F4_P31;
CONN F4_P47<1>:<1>F4_P31;
SEND V3D(1,1,1) TO <1>Box_Size;
SEND 'Box_Size' TO <2>F4_P47;
SEND .05 TO <5>F4_P44;
SEND 1 TO <4>F4_P44;
SEND .1 TO <2>F4_P44;
SEND .1 TO <3>F4_P44;
SEND V3D(.1,.1,.1) TO <2>F4_P31;
SEND -1 TO <2>F4_P35;
{ Path:F5_ }
F5_P49:=F:CBROUTE;
F5_P50:=F:XOR;
CONN F5_P49<1>:<append>Path;
CONN F5_P50<1>:<2>F5_P50;
CONN F5_P50<1>:<1>F5_P49;
SEND TRUE TO <2>F5_P50;
SEND TRUE TO <1>F5_P49;
{ Labels:F6_ }
SEND 'RESET' TO <1>FLABEL11;
SEND 'STRT/STP' TO <1>FLABEL10;
SEND 'SLOWER' TO <1>FLABEL4;
SEND 'FASTER' TO <1>FLABEL3;
SEND 'CLR PATH' TO <1>FLABEL2;
SEND 'TRACE?' TO <1>FLABEL1;
SEND 'BALLSIZE' TO <1>DLABEL8;
SEND 'Z VEL' TO <1>DLABEL7;
SEND 'Y VEL' TO <1>DLABEL6;
```

```
SEND 'X VEL' TO <1>DLABEL5;
SEND 'OS Y ROTATE' TO <1>DLABEL4;
SEND 'Z SIZE' TO <1>DLABEL3;
SEND 'Y SIZE' TO <1>DLABEL2;
SEND 'X SIZE' TO <1>DLABEL1;
{ Frame1:F1_ }
F1_P48:=F:DYROTATE;
CONN DIALS<1>:<1>M2$F1_P2;
CONN DIALS<2>:<1>M2$F1_P3;
CONN DIALS<3>:<1>M2$F1_P4;
CONN DIALS<4>:<1>F1_P48;
CONN DIALS<5>:<1>F2_P41;
CONN DIALS<6>:<1>F2_P42;
CONN DIALS<7>:<1>F2_P43;
CONN DIALS<8>:<1>F4_P44;
CONN M1$F1_P2<1>:<1>F5_P50;
CONN M1$F1_P2<2>:<clear>Path;
CONN M1$F1_P2<3>:<1>F3_P25;
CONN M1$F1_P2<4>:<1>F3_P25;
CONN M1$F1_P2<10>:<1>F3_P28;
CONN M1$F1_P2<11>:<1>M2$F1_P5;
CONN FKEYS<1>:<13>M1$F1_P1;
CONN FKEYS<1>:<1>M1$F1_P2;
CONN F1_P48<1>:<1>Collision.Yrot;
CONN F2_P2<1>:<1>F3_P23;
CONN F2_P39<1>:<2>F5_P49;
CONN F3_P23<2>:<1>F2_P2;
CONN F4_P33<1>:<2>F2_P6;
CONN F4_P33<2>:<2>F2_P7;
CONN F4_P33<3>:<2>F2_P8;
CONN F4_P34<1>:<3>F2_P6;
CONN F4_P34<2>:<3>F2_P7;
CONN F4_P34<3>:<3>F2_P8;
SEND 2 TO <4>M1$F1_P1;
SEND -2 TO <3>M1$F1_P1;
SEND FIX(10000) TO <2>M1$F1_P1;
SEND 200 TO <3>F1_P48;
SEND 0 TO <2>F1_P48;
```

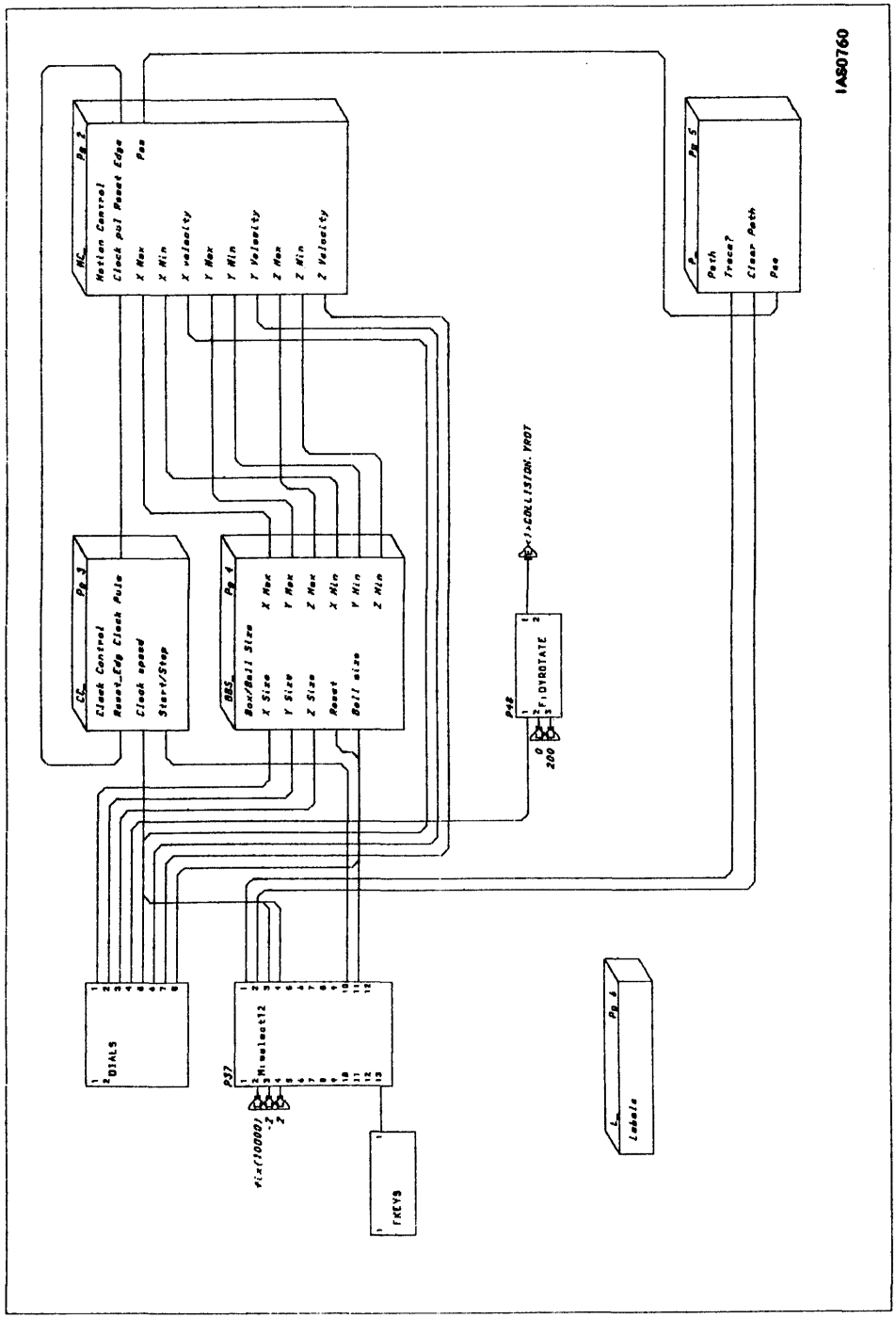


Figure 15-2. COLLISION.FUN (Sheet 1 of 6)
(Function Network for COLLISION.300)

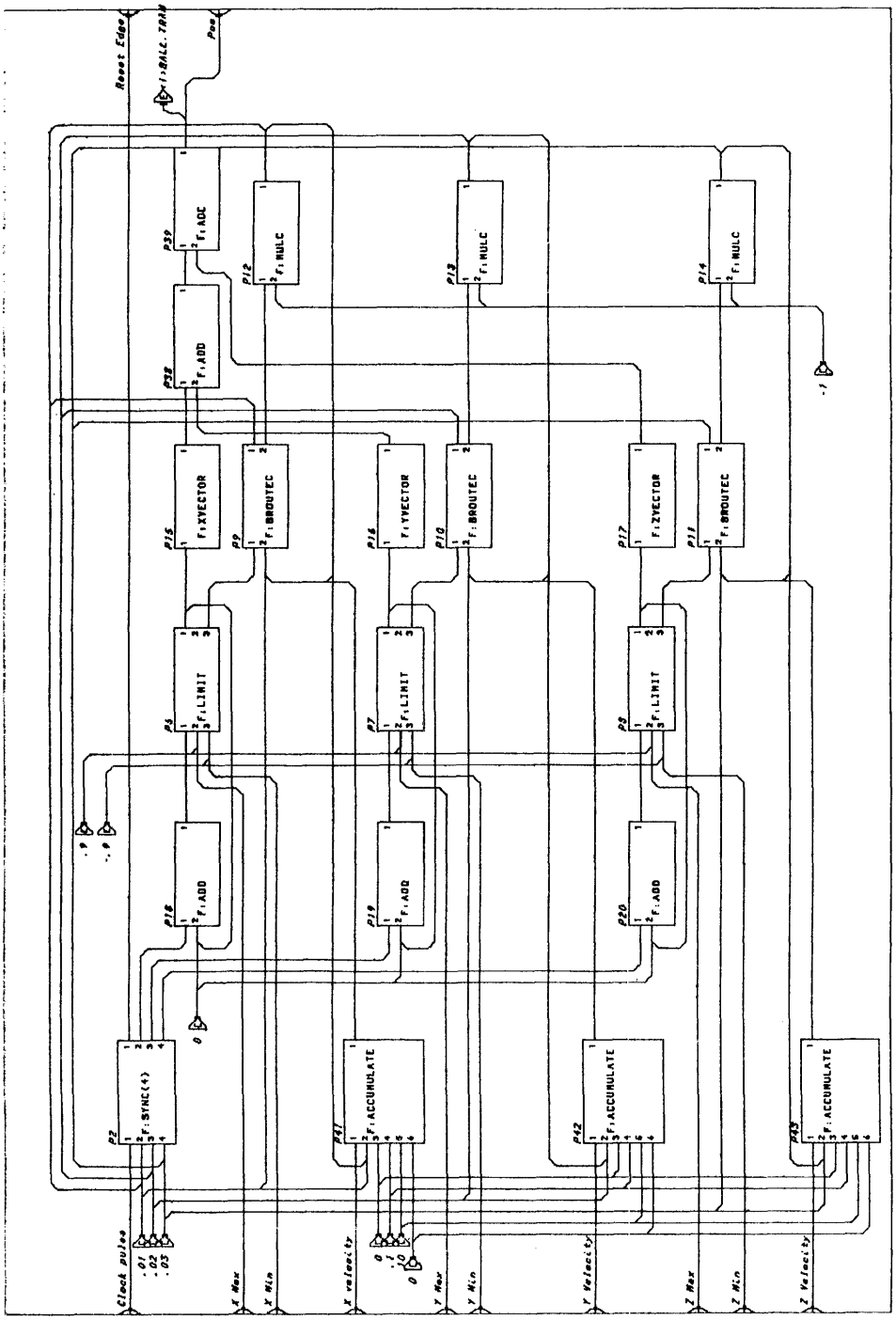


Figure 15-2. COLLISION.FUN (Sheet 2 of 6)

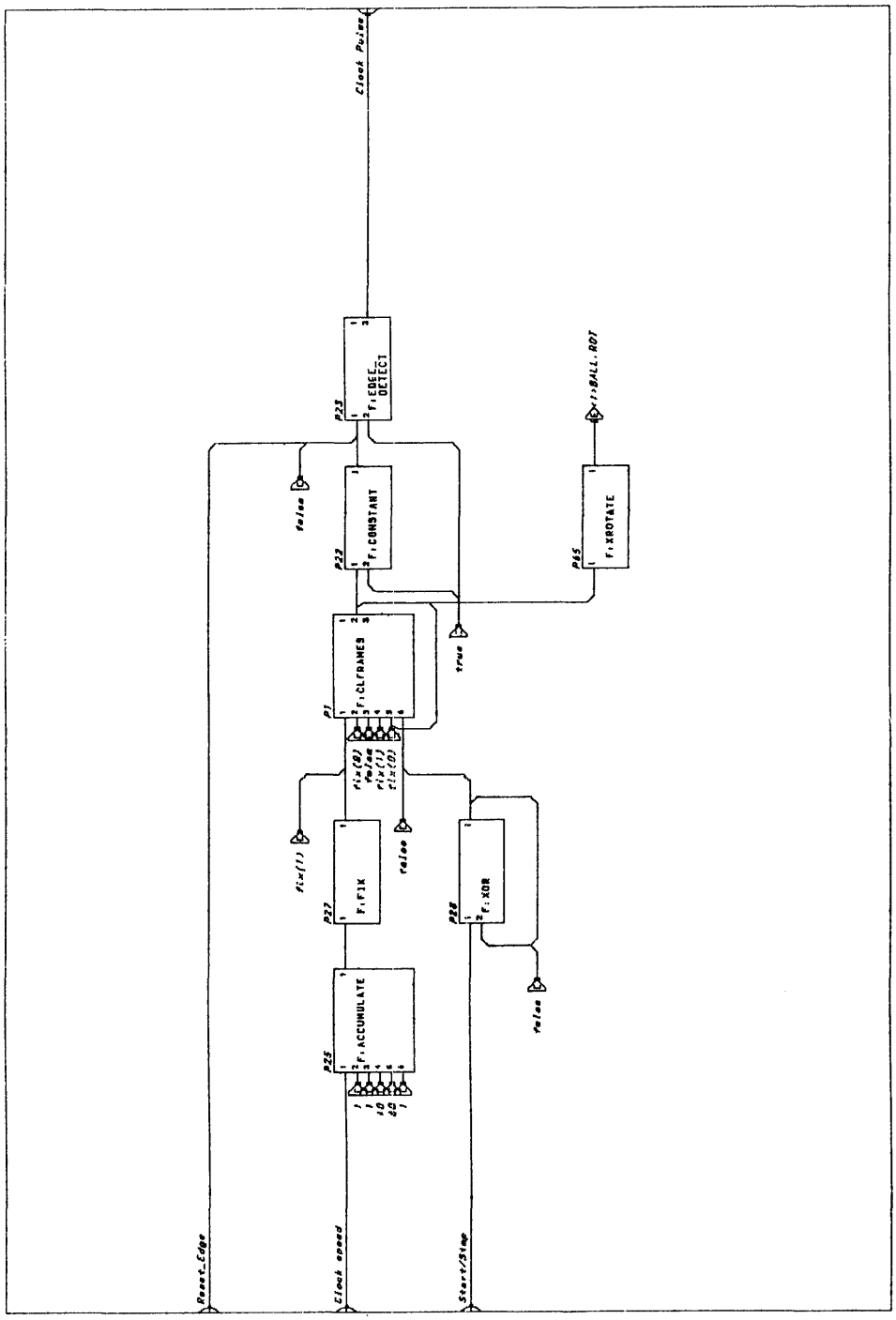


Figure 15-2. COLLISION.FUN (Sheet 3 of 6)

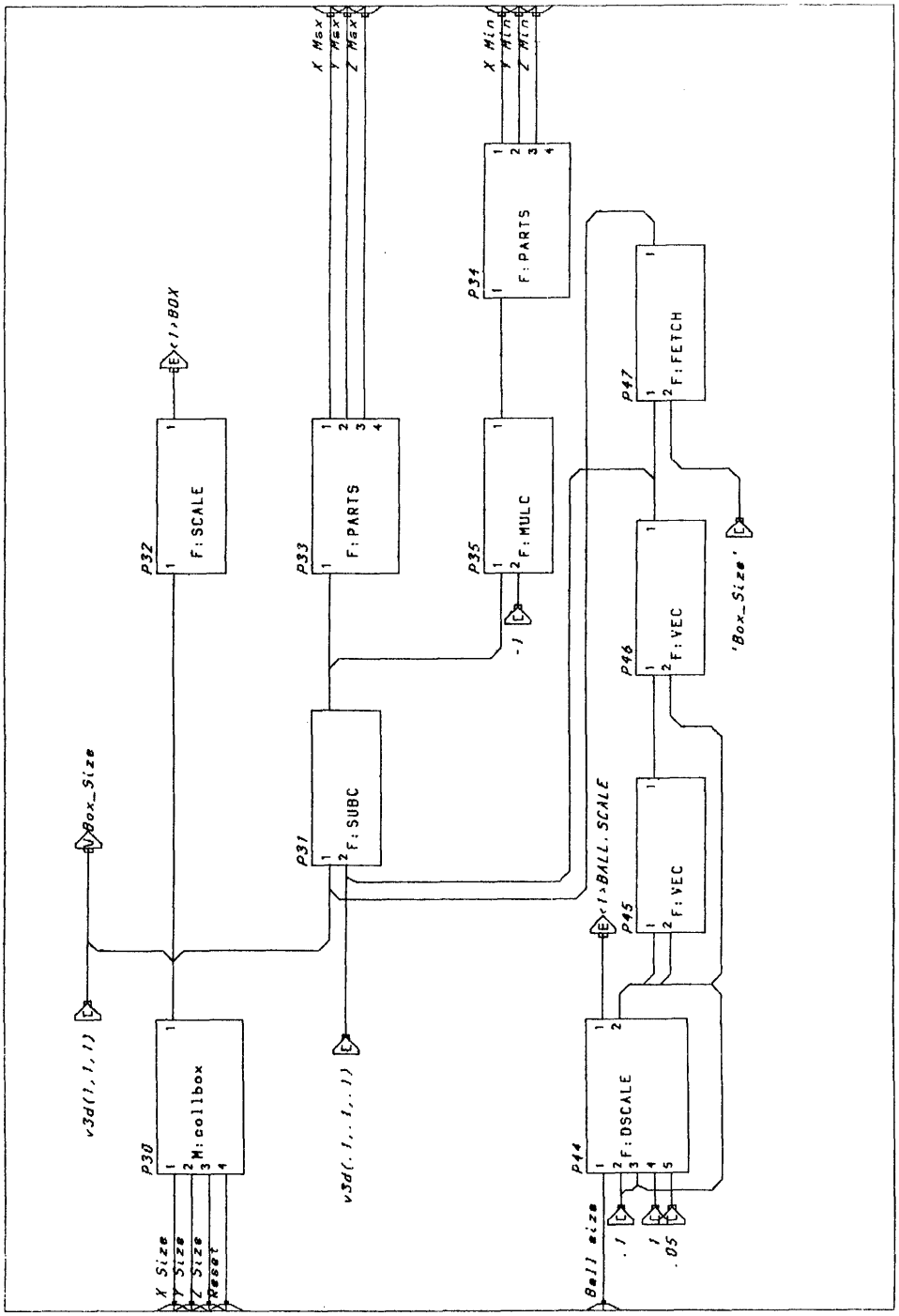


Figure 15-2. COLLISION.FUN (Sheet 4 of 6)

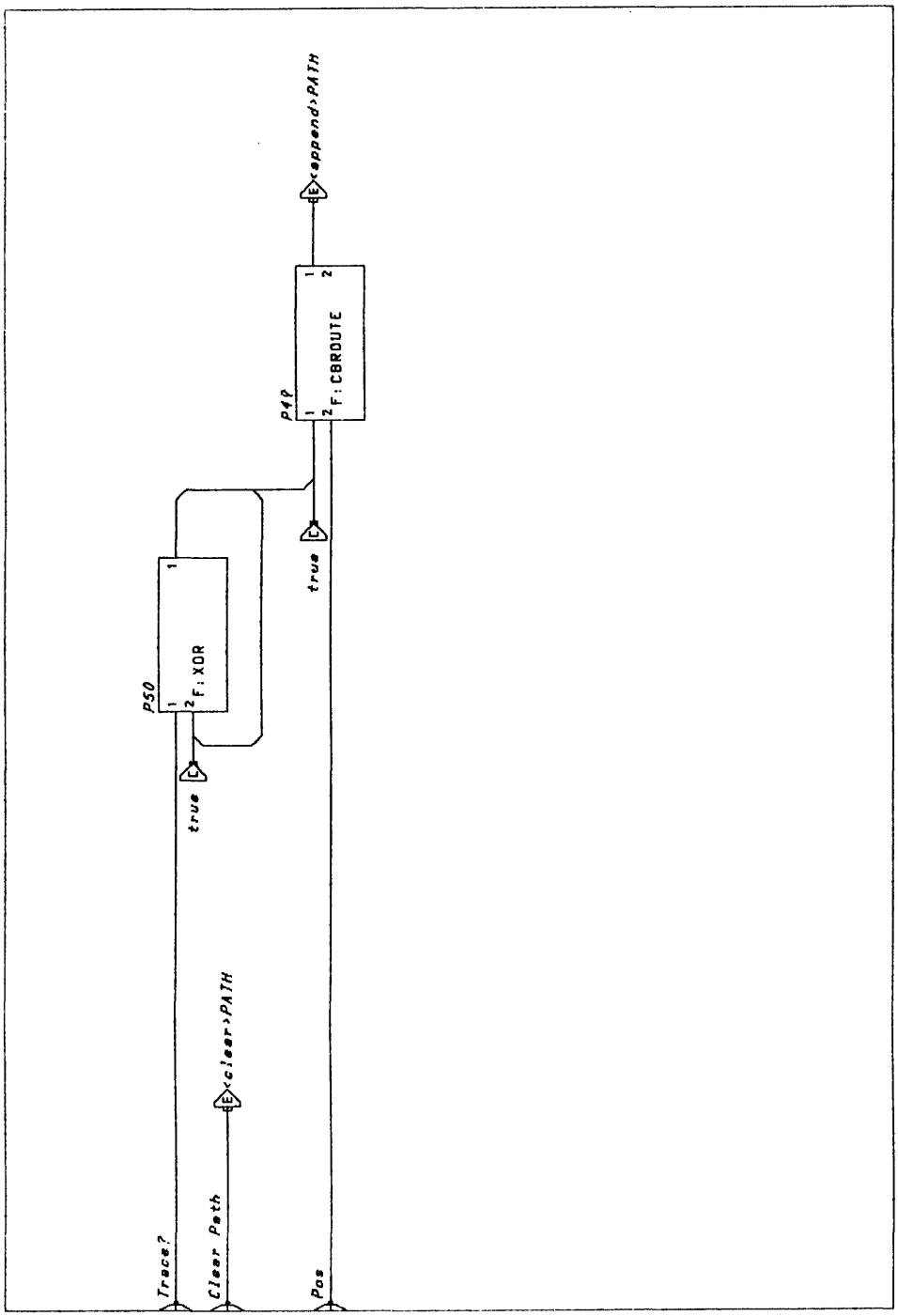


Figure 15-2. COLLISION.FUN (Sheet 5 of 6)

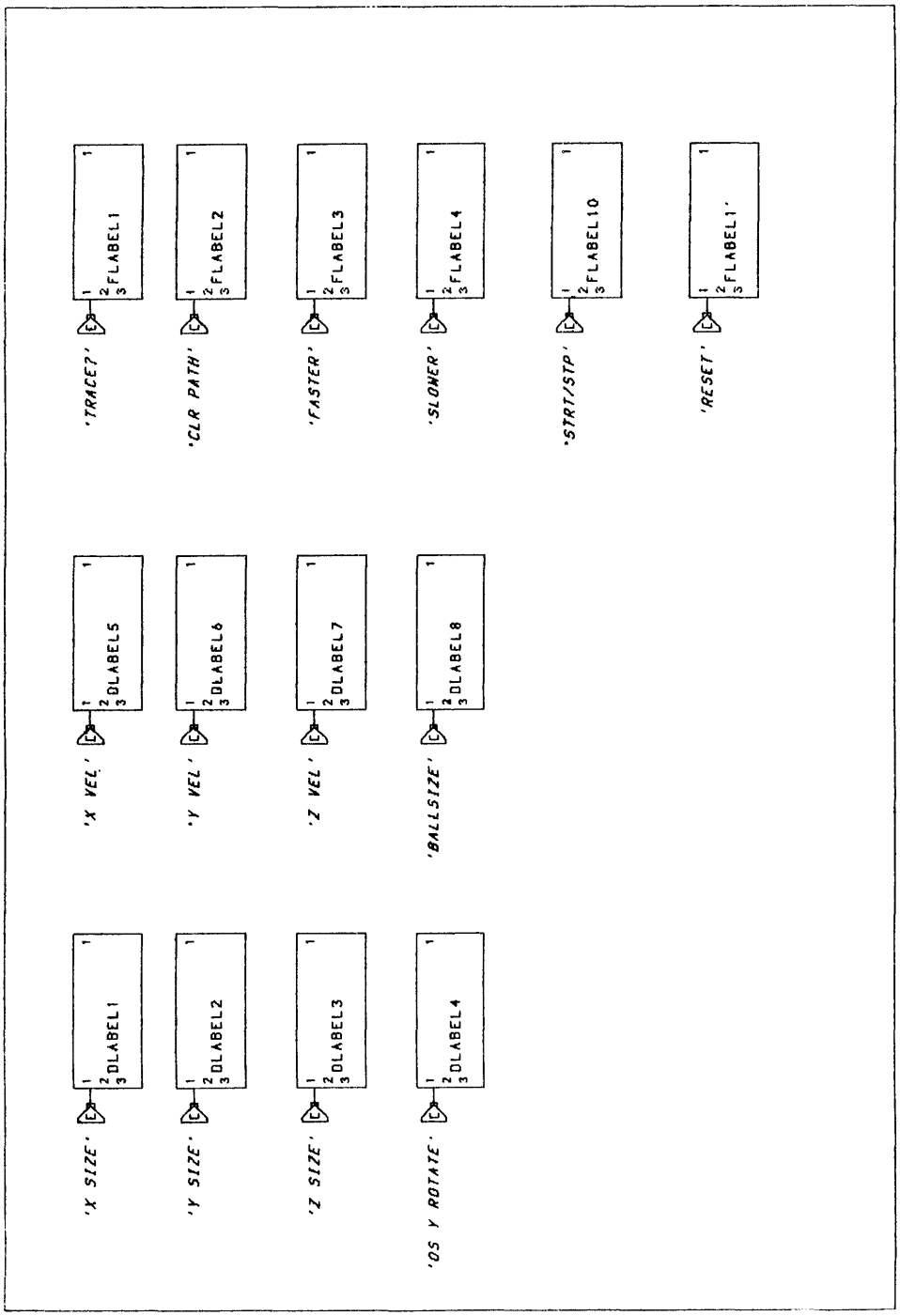


Figure 15-2. COLLISION.FUN (Sheet 6 of 6)

3. Planar Projection Example

3.1. PROJECTN.300

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: July, 1982
Last update: February, 1985

Demonstrate X,Y, and Z planar projections using Matrix_3x3 command. The vector list data node for SPHERE, which is referred to in this structure, is not included in this file.

```
INIT DISP;
DISP Projection;

Projection := BEGIN_S
                CHAR SCALE .65;
                FONT Complex_Roman;
                INST Isometric_View;
                WINDOW x=-7.2:7.2 y=-7.2:7.2;
                INST Front_View,Side_View,Top_View;
              END_S;

Front_View := BEGIN_S
                VIEWPORT HOR=-1:0 VERT=-1:0;
                LOOK AT 3,2,0 FROM 3,2,-12 THEN Object;
              END_S;

Side_View := BEGIN_S
               VIEWPORT HOR=0:1 VERT=-1:0;
               LOOK AT 0,2,3 FROM 12,2,3 THEN Object;
             END_S;

Top_View := BEGIN_S
              VIEWPORT HOR=-1:0 VERT=0:1;
              LOOK AT 3,0,1 FROM 3,12,1 THEN Object;
            END_S;

Isometric_View  := BEGIN_S
                     VIEWPORT HOR=0:1 VERT=0:1;
                     WINDOW x=-7:9 y=-7:9;
Rot :=               ROT 0;
                     ROT X -30;
                     ROT Y 40 THEN Object;
                   END_S;
```

```
Object := BEGIN_S
            SET COLOR 240,1;
            SCALE 8 THEN WS_Gnomon;
            SET COLOR 0,0;
            INST Globe,Xplane,Yplane,Zplane;
          END_S;

Globe := BEGIN_S
Rot :=     ROT 0;
           SCALE 1.5;
           SET COLOR 0,1 THEN Sphere;
           SET COLOR 120,1;
           SCALE 1.5 THEN Os_Gnomon;
         END_S;

Xplane := BEGIN_S
            TRAN 5,0,0;
            INST Xprojection_Matrix;
            ROT Y -90;
            INST Square;
            LABELS -2.5,-2.5 'YZ Plane';
          END_S;

Yplane := BEGIN_S
            TRAN 0,5,0;
            INST Yprojection_Matrix;
            ROT X 90;
            INST Square;
            LABELS -2.5,-2.5 'XZ Plane';
          END_S;

Zplane := BEGIN_S
            TRAN 0,0,-5;
            INST Zprojection_Matrix,Square;
            LABELS -2.5,-2.5 'XY Plane';
          END_S;

XProjection_Matrix := MATRIX_3X3  0,0,0
                                  0,1,0
                                  0,0,1 THEN Globe;

YProjection_Matrix := MATRIX_3X3   1,0,0
                                   0,0,0
                                   0,0,1 THEN Globe;

ZProjection_Matrix := MATRIX_3X3   1,0,0
                                   0,1,0
                                   0,0,0 THEN Globe;
```

```
Square := VEC n=5 3,3 -3,3 -3,-3 3,-3 3,3;

WS_Gnomon := BEGIN_S
               TEXT SIZE .05;
               SET CHARACTERS Screen_Oriented;
               FONT Triplex_Roman;
               LABELS
                 1.1,-.05 'Wx'
                 -.05,1.1 'Wy'
                 -.05,-.05,1.1 'Wz';
               VEC ITEM n=5 P 0,.8,0 L 0,0,0 L .8,0,0
                 P 0,0,0 L 0,0,.8;
               TRAN .8,0 THEN Xarrow;
               TRAN 0,.8 THEN Arrow;
               TRAN 0,0,.8 THEN Zarrow;
             END_S;

Xarrow := ROT z -90 THEN Arrow;
Arrow := SCALE .025,.2,.025 THEN Pyramid;
Zarrow := ROT x 90 THEN Arrow;

OS_Gnomon := BEGIN_S
               CHARACTER SCALE .0375;
               SET CHARACTERS Screen_Oriented;
               FONT Triplex_Roman;
               LABELS
                 1.1,-.05 'Ox'
                 -.05,1.1 'Oy'
                 -.05,-.05,1.1 'Oz';
               WITH PATTERN 1 1 LEN .1
               VEC ITEM n=5 P 0,.8,0 L 0,0,0 L .8,0,0
                 P 0,0,0 L 0,0,.8;
               TRAN .8,0 THEN Xarrow;
               TRAN 0,.8 THEN Arrow;
               TRAN 0,0,.8 THEN Zarrow;
             END_S;

Pyramid := VEC BLOCK ITEM n=10
   P 1,0, 1 L -1,0,1 L -1,0,-1 L 1,0,-1 L 1,0,1 L 0,1,0 L 1,0,-1
   P -1,0,-1 L  0,1,0 L -1,0,1;
```

3.2. PROJECTN.FUN

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: July, 1982
Last update: February, 1985

Demonstrate X,Y, and Z planar projections using Matrix_3x3 command. The vector list data node for SPHERE, which is referred to in this structure, is not included in this file.

```
{ Code generated by Network Editor 1.07 }
{ PROJECTN }
{ Frame-Prefix Macro-Prefix  }
{ Frame1:M2$F1_ }
M2$F1_P1:=F:MULC;
M2$F1_P2:=F:MULC;
M2$F1_P3:=F:MULC;
M2$F1_P4:=F:XROTATE;
M2$F1_P5:=F:YROTATE;
M2$F1_P6:=F:ZROTATE;
CONN M2$F1_P1<1>:<1>M2$F1_P4;
CONN M2$F1_P2<1>:<1>M2$F1_P5;
CONN M2$F1_P3<1>:<1>M2$F1_P6;
SEND 200 TO <2>M2$F1_P2;
SEND 200 TO <2>M2$F1_P1;
SEND 200 TO <2>M2$F1_P3;
{ Frame1:M1$F1_ }
{World Space Rotations}
M1$F1_P2:=F:CMUL;
M1$F1_P3:=F:CONSTANT;
CONN M2$F1_P5<1>:<2>M1$F1_P2;
CONN M2$F1_P4<1>:<2>M1$F1_P2;
CONN M2$F1_P6<1>:<2>M1$F1_P2;
CONN M1$F1_P2<1>:<1>M1$F1_P2;
CONN M1$F1_P3<1>:<1>M1$F1_P2;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>M1$F1_P3;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>M1$F1_P2;
{ Frame1:M3$F1_ }
M3$F1_P1:=F:INPUTS_CHOOSE(13);
M3$F1_P2:=F:ROUTE(12);
CONN M3$F1_P1<1>:<2>M3$F1_P2;
SEND TRUE TO <1>M3$F1_P1;
SEND TRUE TO <2>M3$F1_P1;
```

```
SEND TRUE TO <3>M3$F1_P1;
SEND TRUE TO <4>M3$F1_P1;
SEND TRUE TO <5>M3$F1_P1;
SEND TRUE TO <6>M3$F1_P1;
SEND TRUE TO <7>M3$F1_P1;
SEND TRUE TO <8>M3$F1_P1;
SEND TRUE TO <9>M3$F1_P1;
SEND TRUE TO <10>M3$F1_P1;
SEND TRUE TO <11>M3$F1_P1;
SEND TRUE TO <12>M3$F1_P1;
{ Labels:F2_ }
SEND 'RESET' TO <1>FLABEL11;
SEND 'OS ROT' TO <1>FLABEL2;
SEND 'WS ROT' TO <1>FLABEL1;
SEND 'OBJ ZROT' TO <1>DLABEL7;
SEND 'OBJ YROT' TO <1>DLABEL6;
SEND 'OBJ XROT' TO <1>DLABEL5;
SEND 'VIEWZROT' TO <1>DLABEL3;
SEND 'VIEWYROT' TO <1>DLABEL2;
SEND 'VIEWXROT' TO <1>DLABEL1;
{ Frame1:F1_ }
F1_P2:=F:CROUTE(2);
F1_P3:=F:MULC;
F1_P4:=F:MULC;
F1_P5:=F:MULC;
F1_P6:=F:XROTATE;
F1_P7:=F:YROTATE;
F1_P8:=F:ZROTATE;
F1_P9:=F:CMUL;
F1_P10:=F:MULC;
F1_P14:=F:CONSTANT;
CONN M1$F1_P2<1>:<1>Isometric_View.Rot;
CONN M1$F1_P3<1>:<1>Isometric_View.Rot;
CONN F1_P2<1>:<2>F1_P9;
CONN F1_P2<2>:<1>F1_P10;
CONN F1_P3<1>:<1>F1_P6;
CONN F1_P4<1>:<1>F1_P7;
CONN F1_P5<1>:<1>F1_P8;
CONN F1_P6<1>:<2>F1_P2;
CONN F1_P7<1>:<2>F1_P2;
CONN F1_P8<1>:<2>F1_P2;
CONN F1_P9<1>:<1>Globe.Rot;
CONN F1_P9<1>:<2>F1_P10;
CONN F1_P9<1>:<1>F1_P9;
CONN F1_P10<1>:<1>F1_P9;
CONN F1_P10<1>:<1>Globe.Rot;
CONN F1_P10<1>:<2>F1_P10;
```

```
CONN M3$F1_P2<1>:<1>F1_P2;
CONN M3$F1_P2<2>:<1>F1_P2;
CONN M3$F1_P2<11>:<1>F1_P14;
CONN M3$F1_P2<11>:<1>M1$F1_P3;
CONN FKEYS<1>:<13>M3$F1_P1;
CONN FKEYS<1>:<1>M3$F1_P2;
CONN DIALS<1>:<1>M2$F1_P1;
CONN DIALS<2>:<1>M2$F1_P2;
CONN DIALS<3>:<1>M2$F1_P3;
CONN DIALS<5>:<1>F1_P3;
CONN DIALS<6>:<1>F1_P4;
CONN DIALS<7>:<1>F1_P5;
CONN F1_P14<1>:<2>F1_P10;
CONN F1_P14<1>:<1>F1_P9;
CONN F1_P14<1>:<1>Globe.Rot;
SEND FIX(2) TO <2>M3$F1_P1;
SEND FIX(1) TO <1>M3$F1_P1;
SEND FIX(1) TO <13>M3$F1_P1;
SEND FIX(1) TO <1>M3$F1_P2;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>F1_P14;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <1>F1_P9;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <2>F1_P10;
SEND 200 TO <2>F1_P3;
SEND 200 TO <2>F1_P4;
SEND 200 TO <2>F1_P5;
```

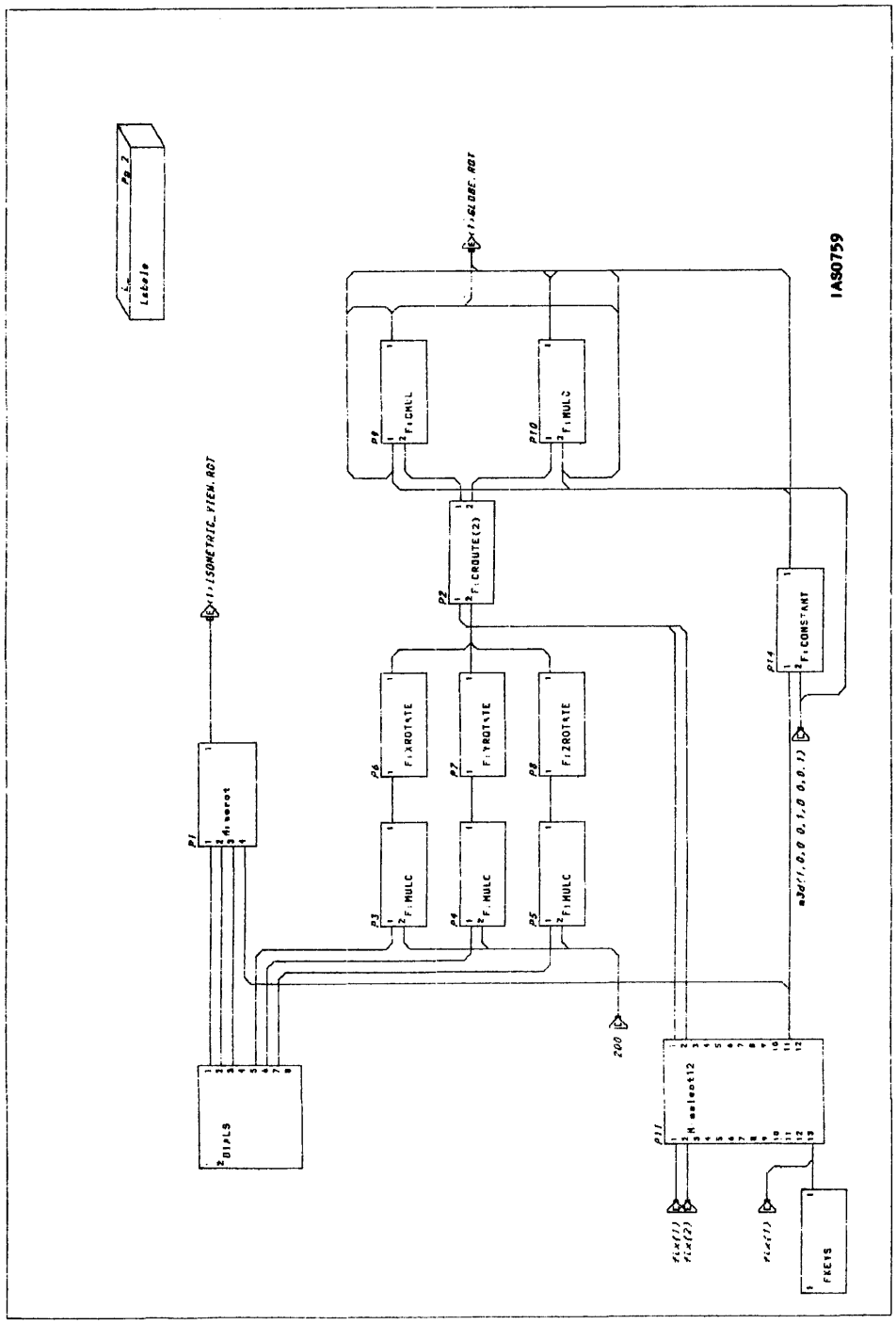


Figure 15-3. PROJECTN.FUN (Sheet 1 of 2)
(Function Network for PROJECTN.300)

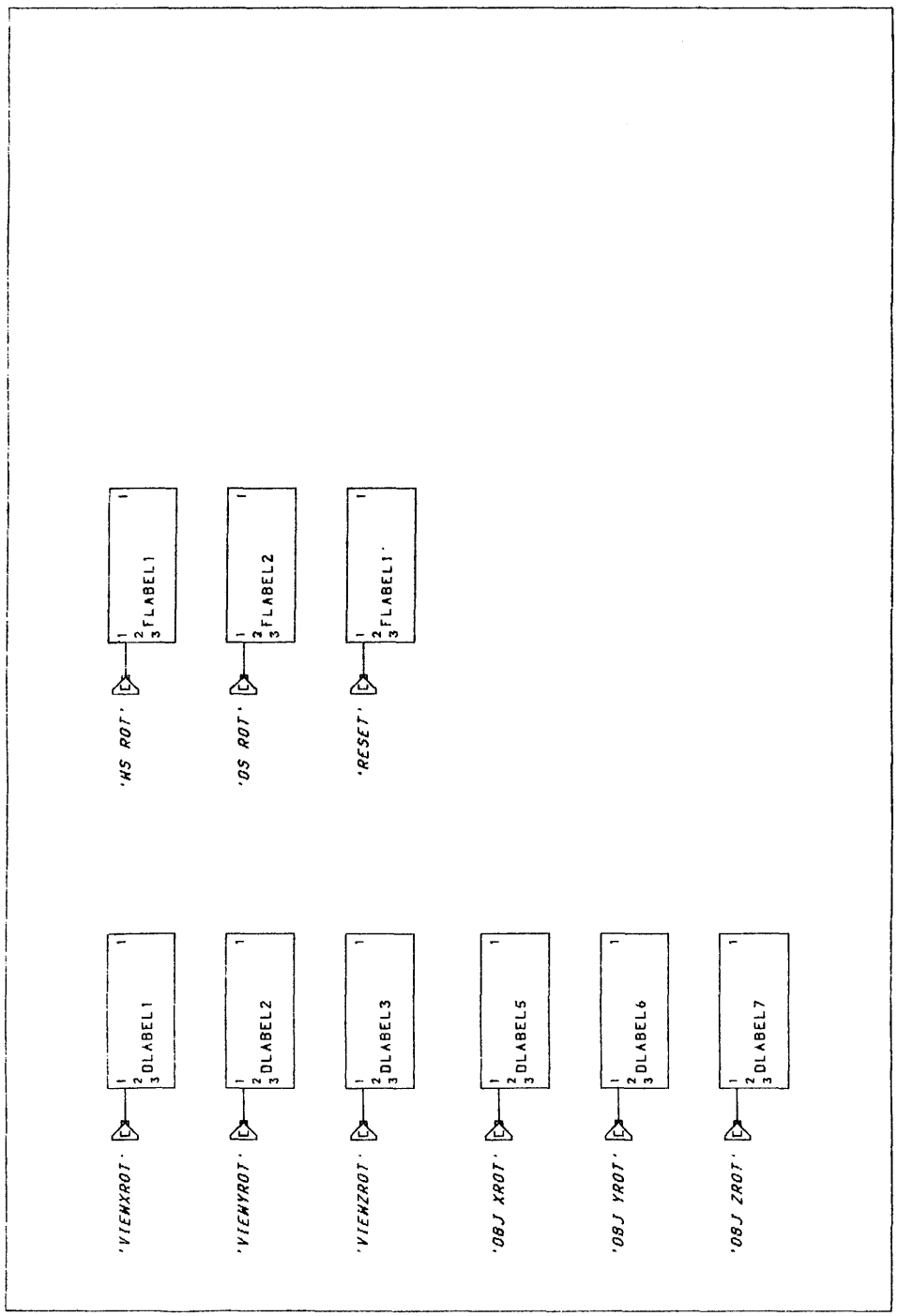


Figure 15-3. PROJECTN.FUN (Sheet 2 of 2)

4. Transformation Example

4.1. TRISQUARE.300

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: December, 1983
Last update: February, 1985

Demonstration to transform four pieces from an equilateral triangle to a square, and vice versa. Can be done either manually with the dials or automatically by starting a clock. The control network is in TRISQUARE.FUN.

```
INIT DISP;
DISP TriSquare;

TriSquare := BEGIN_S
                WINDOW X=-5:5 Y=-5:5;
                TRAN -2,2;
Rot :=          ROT 0;
                SET COLOR 0,1 THEN Part1;
                TRAN 1,-1.268;
P2_Rot :=       ROT 0;
                SET COLOR 90,1 THEN Part2;
                TRAN -1,-1.732;
P3_Rot :=       ROT 0;
                SET COLOR 180,1 THEN Part3;
                TRAN -1.5,.866;
P4_Rot :=       ROT 0;
                SET COLOR 240,1 THEN Part4;
             END_S;

PART1 :=   VEC n=5 0,.4641 -.5,-.4019 -.2857,-1.5151 1,-1.268 0,.4641;

PART2 :=   VEC n=5 0,0 -1.2857,-.2471 -1,-1.732 1,-1.732 0,0;

PART3 :=   VEC n=5 0,0 -.2142,1.1135 -1.5,.866 -2,0 0,0;

PART4 :=   VEC n=4 0,0 1.2858,.2475 1,1.732 0,0;
```

4.2. TRISQUARE.FUN

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: December, 1983
Last update: February, 1985

Network to control the structure created by TRISQUARE.300.

```
{ Code generated by Network Editor 1.07 }
{ TRISQUARE }
{ Frame-Prefix Macro-Prefix  }
{ Clock Motion:F2_ }
{ first in que ---> }
F2_P13:=F:EQC;
F2_P14:=F:XOR;
F2_P15:=F:CLFRAMES;
F2_P16:=F:BROUTEC;
F2_P17:=F:SYNC(2);
CONN FKEYS<1>:<1>F2_P13;
CONN F2_P13<1>:<1>F2_P14;
CONN F2_P14<1>:<2>F2_P14;
CONN F2_P14<1>:<6>F2_P15;
CONN F2_P15<2>:<5>F2_P15;
CONN F2_P15<3>:<1>F2_P16;
CONN F2_P16<2>:<2>F2_P15;
CONN F2_P16<2>:<1>F2_P17;
CONN F2_P17<2>:<4>F2_P15;
CONN F2_P17<2>:<2>F2_P17;
SEND FIX(1) TO <2>F2_P13;
SEND FALSE TO <2>F2_P14;
SEND FIX(-1) TO <2>F2_P17;
SEND FIX(1) TO <2>F2_P17;
SEND FIX(179) TO <2>F2_P16;
SEND FIX(0) TO <5>F2_P15;
SEND FIX(1) TO <4>F2_P15;
SEND FALSE TO <3>F2_P15;
SEND FALSE TO <6>F2_P15;
SEND FIX(179) TO <2>F2_P15;
SEND FIX(6) TO <1>F2_P15;
{ Labels:F3_ }
SEND 'STRT/STP' TO <1>FLABEL1;
SEND 'JOINT 3' TO <1>DLABEL3;
SEND 'JOINT 2' TO <1>DLABEL2;
SEND 'JOINT 1' TO <1>DLABEL1;
```

```
{ Frame1:F1_ }
F1_P1:=F:ACCUMULATE;
F1_P2:=F:ACCUMULATE;
F1_P3:=F:ACCUMULATE;
F1_P4:=F:ZROTATE;
F1_P5:=F:ZROTATE;
F1_P6:=F:ZROTATE;
CONN F1_P1<1>:<1>F1_P4;
CONN F1_P2<1>:<1>F1_P5;
CONN F1_P3<1>:<1>F1_P6;
CONN F1_P4<1>:<1>Trisquare.P2_Rot;
CONN F1_P5<1>:<1>Trisquare.P3_Rot;
CONN F1_P6<1>:<1>Trisquare.P4_Rot;
CONN DIALS<1>:<1>F1_P1;
CONN DIALS<2>:<1>F1_P2;
CONN DIALS<3>:<1>F1_P3;
CONN F2_P15<2>:<1>F1_P4;
CONN F2_P15<2>:<1>F1_P5;
CONN F2_P15<2>:<1>F1_P6;
SEND 180 TO <5>F1_P1;
SEND 180 TO <5>F1_P2;
SEND 180 TO <5>F1_P3;
SEND 200 TO <4>F1_P1;
SEND 200 TO <4>F1_P2;
SEND 200 TO <4>F1_P3;
SEND 0 TO <2>F1_P1;
SEND 0 TO <3>F1_P1;
SEND 0 TO <6>F1_P1;
SEND 0 TO <2>F1_P2;
SEND 0 TO <3>F1_P2;
SEND 0 TO <6>F1_P2;
SEND 0 TO <2>F1_P3;
SEND 0 TO <3>F1_P3;
SEND 0 TO <6>F1_P3;
```

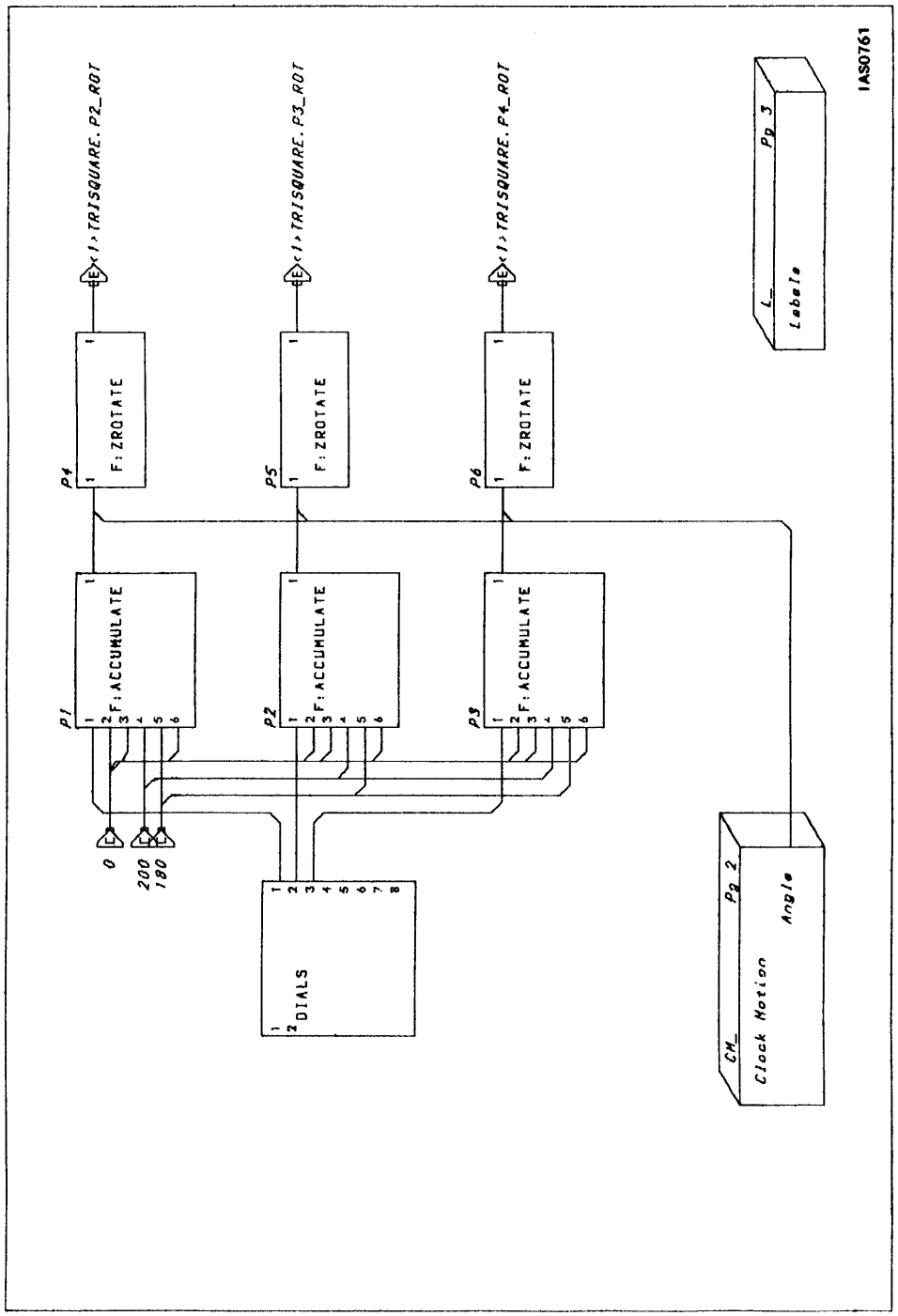


Figure 15-4. TRISQUARE.FUN (Sheet 1 of 3)
(Function Network for TRISQUARE.300)

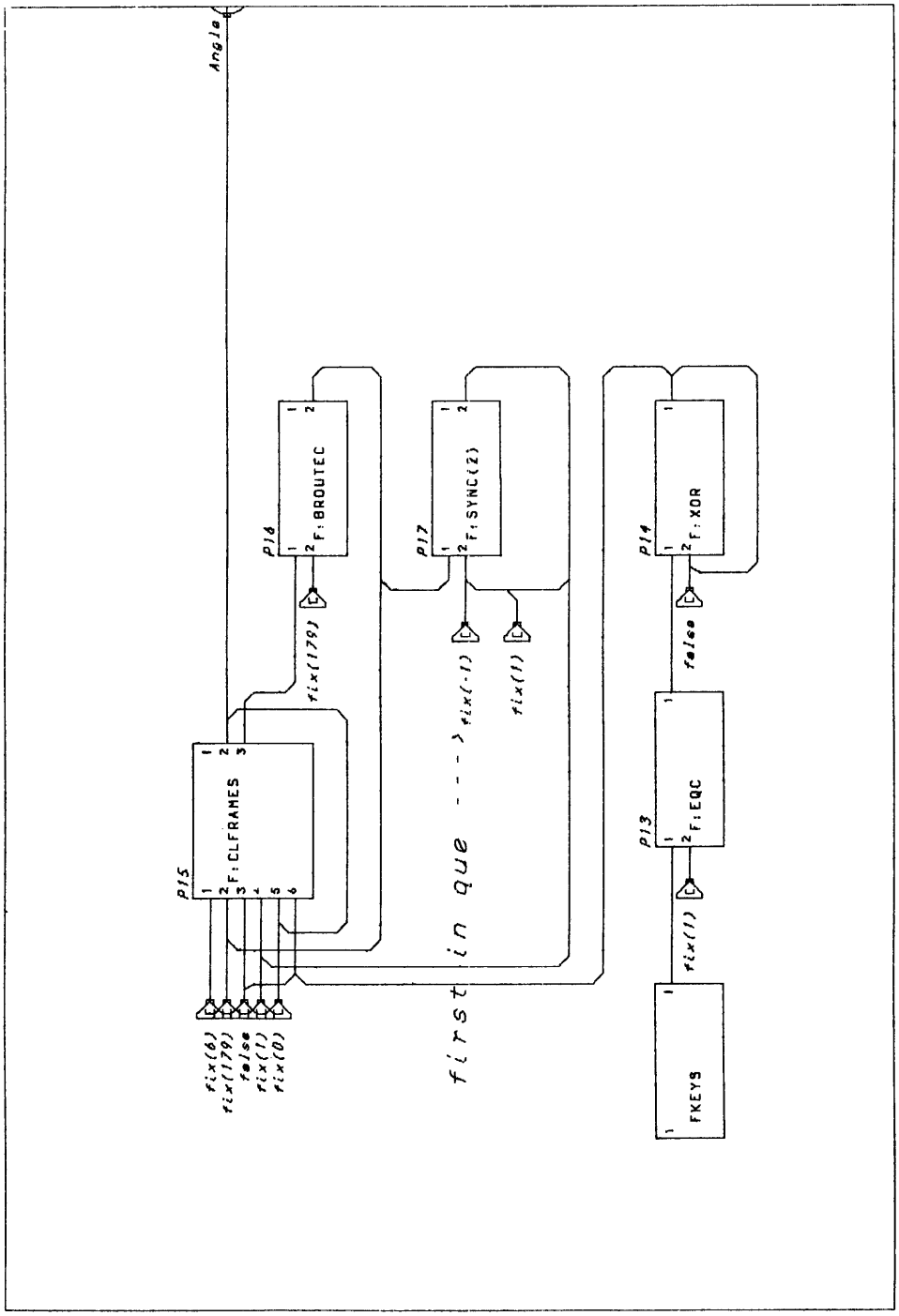


Figure 15-4. TRISQUARE.FUN (Sheet 2 of 3)

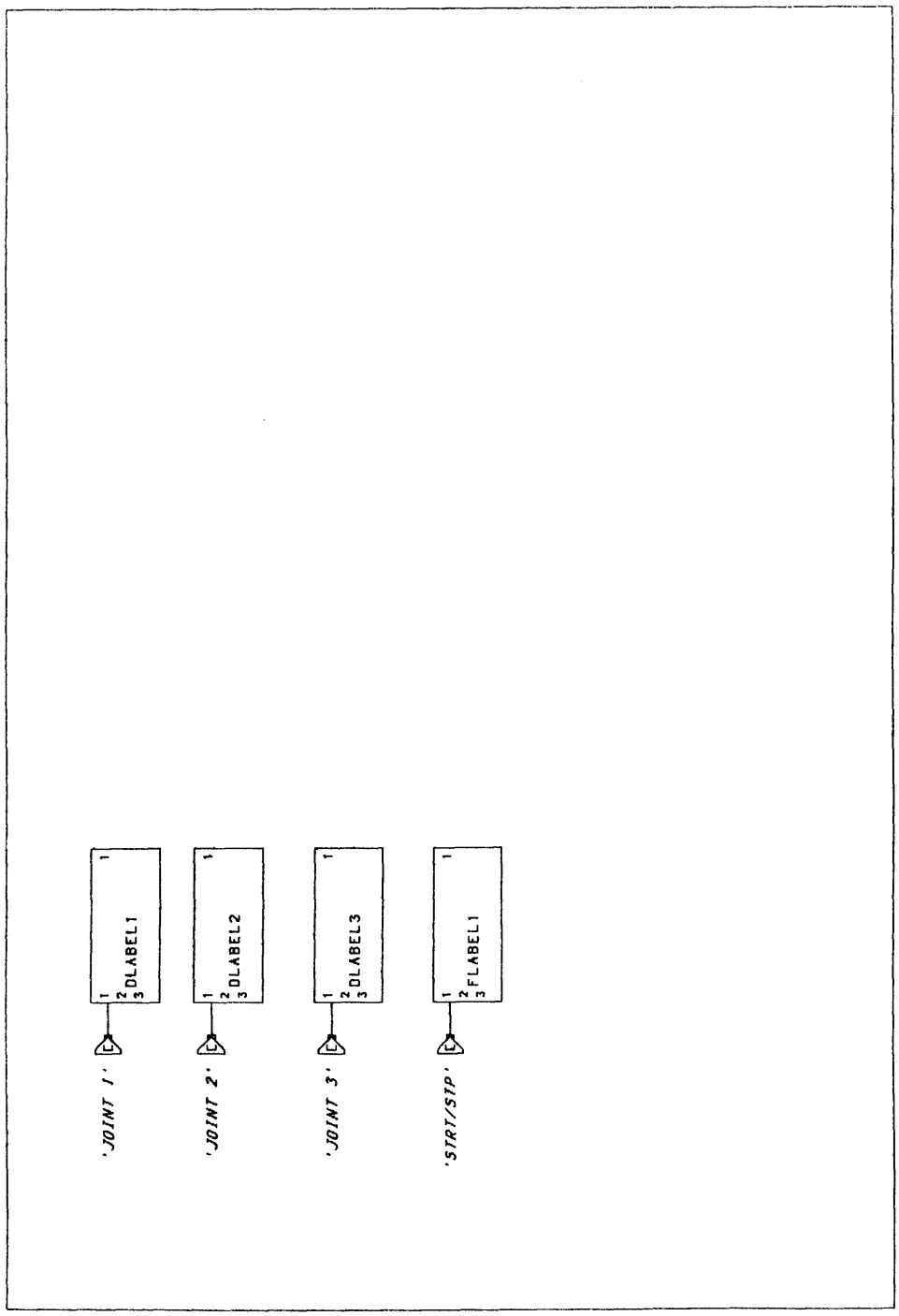


Figure 15-4. TRISQUARE.FUN (Sheet 3 of 3)

5. SET RATE Programming Example

Programmed by: Neil Harrington
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: November, 1984
Last update: February, 1985

PS 390 Set Rate programming example using the GSRs. Pascal version of BLKLEVF.FOR created originally by A. Kerry Evans. To run this program compile it and link it with the Pascal GSR library.

This program creates a PS 390 display structure that has many SET RATE nodes cascaded by offsetting the starting time of each SET RATE node. This structure is another way to create an animated sequence on the PS 390. A function network is not needed, since the PHASE attribute value is modified by the Display Processor as a function of the number of times a SET RATE node is traversed.

```
CONST
   DTheta  = 0.1745329;  { PI/18 }
   %INCLUDE 'gsrlib:ProConst.pas/nolist'

TYPE
   %INCLUDE 'gsrlib:ProTypes.pas/nolist'

VAR
   Theta    : REAL;
   Tran     : P_VectorType;
   I        : INTEGER;
   Vecs     : P_VectorListType;
   Front    : P_VectorListType;
   Name     : P_VaryingType;

%INCLUDE 'gsrlib:ProExtrn.pas/nolist'

PROCEDURE ErrHnd ( Error : INTEGER);
   BEGIN
       WRITELN  ( 'Error: ',Error:3);
   END; { ErrHnd }
```

```
PROCEDURE Calc_Wave;
   VAR
       I,J   : INTEGER;
       VecNum   : INTEGER;
       VecNum2   : INTEGER;

   BEGIN
       VecNum   := -1;
       FOR I := 0 TO 49 DO BEGIN
           VecNum   := VecNum + 2;
           VecNum2   := VecNum + 1;

           Vecs[VecNum].v4[1] := I/50;
           Vecs[VecNum].v4[2] :=
0.8*EXP(-0.02*I)*COS(Theta-0.2513274123*I);
           Vecs[VecNum].v4[3] := 0;
           Vecs[VecNum].v4[4] := 1 - I/150;

           Vecs[VecNum2].v4[1] := Vecs[VecNum].v4[1];
           Vecs[VecNum2].v4[2] := 0;
           Vecs[VecNum2].v4[3] := 0.5;
           Vecs[VecNum2].v4[4] := Vecs[VecNum].v4[4];

           FOR J := 1 TO 4 DO
               Front[I+1].v4[J] := Vecs[VecNum].v4[J];

       END; { FOR I }
   END; { Calc_Wave }

BEGIN
   PAttach  ('LogDevNam=tt:/PhyDevTyp=async',ErrHnd);
   PInitC  (ErrHnd);
   PInitD  (ErrHnd);
   Tran.v4[1]   := -0.5;
   Tran.v4[2]   := 0;
   Tran.v4[3]   := 0;
   PTransBy  ('Sine_Wave',Tran,'Inst',ErrHnd);
   PInst  ('Inst','',ErrHnd);
   Theta   := -DTheta;

   FOR I := 10 TO 46 DO BEGIN
       Theta   := Theta + DTheta;
       WriteV  (Name,'VecList',I:2);
       WRITELN  (Name);
```

```
        Calc_Wave;
        PBeginS  (Name, ErrHnd);
            PSetR ('',1,35,FALSE,I,'',ErrHnd);
            PIfPhase ('',TRUE,'',ErrHnd);
            PVecBegn ('',100,TRUE,FALSE,3,P_Sepa,ErrHnd);
            PVecList (100,Vecs,ErrHnd);
            PVecEnd (ErrHnd);
            PVecBegn ('',50,TRUE,FALSE,3,P_Conn,ErrHnd);
            PVecList (50,Front,ErrHnd);
            PVecEnd (ErrHnd);
        PEndS  (ErrHnd);
        PIncl  (Name,'Inst',ErrHnd);
    END; { FOR I }

    PDisplay  ('Sine_Wave',ErrHnd);
    PDetach  (ErrHnd);
END. { SetRate }
```

6. PS 390 Rendering Example

6.1. RENDER.300

Programmed by: Scott Goodyear
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: May 31, 1987
Last update:

This data structure creates a model and allows the user to put the model in two dynamic viewports (one full screen and one in the lower right-hand corner) and three static viewports in the other corners. Data structure in LIGHT.300 creates two independent light sources to illuminate model. The function network in RENDER.FUN renders dynamic wireframe and static shaded images of the model.

```
{ reserve working storage to transform polygons }
{ put this in the site.dat file on the system disk }
{ reserve 150 - 180 bytes / phong shaded fully anti-aliased polygon }
{ example: reserve reserve_working_storage 800000; for a 4500 polygon
model }

{ set up initial display structure }
universe := begin_s
   split := set level to 0;
   if level = 0 then full;
   if level = 1 then split;
end_s;

{ full screen dynamic window }
full := begin_s
      persp := fov 45 front=2.2 back=3.6;
      look := look at 0,0,0 from 0,0,-3.25;
      vport := viewport horizontal=-1:1 vertical=-1:1 intensity = 0:1
         then world;
end_s;
```

```
{ 1/4 static viewport and 1/4 dynamic viewport }
split := begin_s
      persp := fov 45 front=2.2 back=3.6;
      look := look at 0,0,0 from 0,0,-3.25;
      vport := viewport horizontal=0:1 vertical=-1:0 intensity = 0:1
               then world;
end_s;

{ define two light sources }
{ the world }
world := begin_s
      bits := set bit 1 on;
      b := if bit 1 on;
      set depth_clipping on;
      tran := translate by 0,0,0;
      rot   := rot 0;
      scale := scale by 1;
      rendering := surface;
      if bit 2 on then sun;
      if bit 3 on then moon;
      inst of objects;
end_s;

display universe;

{ multiple objects in memory, to attach your model to this network }
{ put these statements in your network or type them locally }
{ local: }
{  @@  object  := INSTANCE my_model }
{  @@  object1 := INSTANCE my_first_model }
{  @@  object2 := INSTANCE my_second_model }
{    . . .etc. }

objects := begin_s
   select := set level_of_detail to 1;
   if level_of_detail = 1 then OBJECT;
   if level_of_detail = 2 then OBJECT1;
   if level_of_detail = 3 then OBJECT2;
   if level_of_detail = 4 then OBJECT3;
   if level_of_detail = 5 then OBJECT4;
end_s;

{ create a bit bucket to dump unused things }
VAR BitBucket;
```

6.2. LIGHT.300

Programmed by: Scott Goodyear
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: May 31, 1987
Last update:

This data structure creates two light sources for the model in RENDER.300. The two light sources, called Sun and Moon, can be rotated independently. The function network in RENDER.FUN renders dynamic wireframe and static shaded images of the model.

```
{ sun - define a light which can be rotated independently }
SunLight := illumination 1,0,0 color 0,0,1 ambient 1.0;

Sun    := Begin_S
   persp := fov 45 front=2.3 back=4.3;
   look := look at 0,0,0 from 0,0,-3.3;
   Set Intensity on 0:.9;
   Set depth_clipping off;
   rot in x -10;
   Scale := scale by 1;
   azimuth := Rot y 0;
   ra   := Rot z 0;
   Inst SUN_LIGHT_MARKER,SUN_ICON,SUNLIGHT;
End_S;

Sun_Light_Marker := Begin_S
   set color 180,1 then ccircle;
   Set color 120,1;
   Vec n=2 .6,0 1,0;
   tran .6,0;
   rot 90;
   scale .025,.2,.025 then PYRAMID;
End_S;

Sun_Icon := Begin_S
   color := Set color 0,0;
   Tran 1,0;
   Rot y -90;
```

```
   Rational polynomial
    .2,0,8 -.2,-.2,-8 0,.1,4 chords=15;
   Rational polynomial
    .2,0,-8 -.2,-.2,8 0,.1,-4 chords=15;
   Vec sep n=15
    -.1,0 -.05,0 .05,0 .1,0
 0,-.1 0,-.05 0,.05 0,.1
 -.0707,-.0707 -.0354,-.0354
 .0354,.0354 .0707,.0707
 -.0707,.0707 -.0354,.0354
 .0354,-.0354 .0707,-.0707;
End_S;

{ moon - define a light which can be rotated independently }
MoonLight := illumination 1,0,0 color 0,0,1 ambient 1.0;

Moon := begin_structure
   persp := fov 45 front=2.3 back=4.3;
   look := look at 0,0,0 from 0,0,-3.3;
   Set Intensity on 0:.9;
   Set depth_clipping off;
   rot in x -10;
   Scale := scale by 1;
   azimuth := Rot y 90;
   ra  := Rot z 0;
   instance MoonLight,Moon_Icon,Moon_Light_Marker;
end_s;

Moon_Icon := begin_s
   Color := set color 0,0;
   tran := trans by 1,0,.01;
   rot in y -90;
           rational polynomial .2,0,4 -.2,-.2,-4 0,.1,2 chords=15;
           rational polynomial .12,0,4 -.12,-.2,-4 0,.1,2 chords=15;
end_s;

Moon_Light_Marker := Begin_S
   set color 0,1 then ccircle;
   Set color 120,1;
   Vec n=2 .6,0 1,0;
   tran .6,0;
   rot 90;
   scale .025,.2,.025 then PYRAMID;
End_S;

{ light source guide circle }
```

```
CCircle := vec n=72
-1.0000, 0.0000
-0.9962, 0.0872
-0.9848, 0.1736
-0.9659, 0.2588
-0.9397, 0.3420
-0.9063, 0.4226
-0.8660, 0.5000
-0.8192, 0.5736
-0.7660, 0.6428
-0.7071, 0.7071
-0.6428, 0.7660
-0.5736, 0.8192
-0.5000, 0.8660
-0.4226, 0.9063
-0.3420, 0.9397
-0.2588, 0.9659
-0.1736, 0.9848
-0.0872, 0.9962
 0.0000, 1.0000
 0.0872, 0.9962
 0.1736, 0.9848
 0.2588, 0.9659
 0.3420, 0.9397
 0.4226, 0.9063
 0.5000, 0.8660
 0.5736, 0.8192
 0.6428, 0.7660
 0.7071, 0.7071
 0.7660, 0.6428
 0.8192, 0.5736
 0.8660, 0.5000
 0.9063, 0.4226
 0.9397, 0.3420
 0.9659, 0.2588
 0.9848, 0.1736
 0.9962, 0.0872
 1.0000, 0.0000
 0.9962,-0.0872
 0.9848,-0.1736
 0.9659,-0.2588
 0.9397,-0.3420
 0.9063,-0.4226
 0.8660,-0.5000
 0.8192,-0.5736
 0.7660,-0.6428
```

```
 0.7071,-0.7071
 0.6428,-0.7660
 0.5736,-0.8192
 0.5000,-0.8660
 0.4226,-0.9063
 0.3420,-0.9397
 0.2588,-0.9659
 0.1736,-0.9848
 0.0872,-0.9962
 0.0000,-1.0000
-0.0872,-0.9962
-0.1736,-0.9848
-0.2588,-0.9659
-0.3420,-0.9397
-0.4226,-0.9063
-0.5000,-0.8660
-0.5736,-0.8192
-0.6428,-0.7660
-0.7071,-0.7071
-0.7660,-0.6428
-0.8192,-0.5736
-0.8660,-0.5000
-0.9063,-0.4226
-0.9397,-0.3420
-0.9659,-0.2588
-0.9848,-0.1736
-0.9962,-0.0872
-1,0;

{ light source pyramid for arrow }
Pyramid := Vec block item n=10
   P  1,0, 1 L -1,0,1 L -1,0,-1 L 1,0,-1 L 1,0,1 L 0,1,0 L 1,0,-1
   P -1,0,-1 L  0,1,0 L -1,0,1;
```

6.3. RENDER.FUN

Programmed by: Scott Goodyear
Evans & Sutherland
P.O. Box 8700
Salt Lake City, Utah 84108

Created: May 31, 1987
Last update:

The function network created in RENDER.FUN renders dynamic wireframe and static shaded images of the model created in RENDER.300, and illuminated by light sources created in LIGHT.300.

```
{ network to mux the dials }
DIALMUX1 := F:CROUTE(4);
DIALMUX2 := F:CROUTE(4);
DIALMUX3 := F:CROUTE(4);
DIALMUX4 := F:CROUTE(4);
DIALMUX5 := F:CROUTE(4);
DIALMUX6 := F:CROUTE(4);
DIALMUX7 := F:CROUTE(4);
DIALMUX8 := F:CROUTE(4);

CONN Dials <1>:<2> DialMux1;
CONN Dials <2>:<2> DialMux2;
CONN Dials <3>:<2> DialMux3;
CONN Dials <4>:<2> DialMux4;
CONN Dials <5>:<2> DialMux5;
CONN Dials <6>:<2> DialMux6;
CONN Dials <7>:<2> DialMux7;
CONN Dials <8>:<2> DialMux8;

{ connections to bitbucket }
CONN DialMux4 <2>:<1> BitBucket;
CONN DialMux4 <3>:<1> BitBucket;
CONN DialMux4 <4>:<1> BitBucket;
CONN DialMux5 <4>:<1> BitBucket;
CONN DialMux6 <4>:<1> BitBucket;
CONN DialMux7 <2>:<1> BitBucket;
CONN DialMux7 <3>:<1> BitBucket;
CONN DialMux7 <4>:<1> BitBucket;
CONN DialMux8 <4>:<1> BitBucket;
```

```
{ dial control network }
{ world space rotation, X,Y,Z translation and back-clipping for the
object}
WSXMUL := F:MULC;
WSYMUL := F:MULC;
WSZMUL := F:MULC;
WSXROT := F:XROTATE;
WSYROT := F:YROTATE;
WSZROT := F:ZROTATE;
PRSCALE := F:DSCALE;
WSROTATE := F:CMUL;
WSRESET := F:XROTATE;

CONN DIALMUX5 <1>:<1> WSXMUL;
CONN DIALMUX6 <1>:<1> WSYMUL;
CONN DIALMUX7 <1>:<1> WSZMUL;
CONN DIALMUX8 <1>:<1> PRSCALE;

CONN WSXMUL <1>:<1> WSXROT;
CONN WSYMUL <1>:<1> WSYROT;
CONN WSZMUL <1>:<1> WSZROT;

CONN WSXROT <1>:<2> WSROTATE;
CONN WSYROT <1>:<2> WSROTATE;
CONN WSZROT <1>:<2> WSROTATE;

CONN WSRESET  <1>:<1> WSROTATE;
CONN WSROTATE <1>:<1> WORLD.ROT;
CONN WSROTATE <1>:<1> WSROTATE;
CONN PRSCALE  <1>:<1> WORLD.SCALE;
CONN PRSCALE  <2>:<3> PRSCALE;

SEND 180 TO <2> WSXMUL;
SEND 180 TO <2> WSYMUL;
SEND 180 TO <2> WSZMUL;
SEND 0 TO <1> WSRESET;
SEND 1   to <2> PRSCALE;
SEND 1   to <3> PRSCALE;
SEND 100 to <4> PRSCALE;
SEND .1  to <5> PRSCALE;

{ scale value }
ObjScalepr := f:print;
conn PRSCALE <2>:<1> ObjScalePr;
```

```
conn ObjScalepr <1>:<1> dlabel8;

{ translation of object }
TRAN_ACC := f:accumulate;
TRAN_XVEC := f:xvector;
TRAN_ZVEC := f:zvector;
TRAN_YVEC := f:yvector;

connect DIALMUX1 <1>:<1> tran_xvec;
connect DIALMUX2 <1>:<1> tran_yvec;
connect DIALMUX3 <1>:<1> tran_zvec;
connect tran_xvec <1>:<1> TRAN_ACC;
connect tran_yvec <1>:<1> TRAN_ACC;
connect tran_zvec <1>:<1> TRAN_ACC;
connect TRAN_ACC <1>:<1> WORLD.tran;

send v3D(0,0,0) to <2> TRAN_ACC;

{ print values }
Tran_Vals := f:parts;
TranXPr := f:print;
TranYPr := f:print;
TranZPr := f:print;

conn Tran_Acc <1>:<1> Tran_Vals;
conn Tran_Vals <1>:<1> TranXPr;
conn Tran_Vals <2>:<1> TranYPr;
conn Tran_Vals <3>:<1> TranZPr;
conn TranXPr <1>:<1> dlabel1;
conn TranYPr <1>:<1> dlabel2;
conn TranZPr <1>:<1> dlabel3;

{ PS 390 reset network - resets rotation, translation, }
{ and scaling matrices, and display node }
WSRESET := F:SYNC(5);
SETUP CNESS TRUE <2> wsreset;
SETUP CNESS TRUE <3> wsreset;
SETUP CNESS TRUE <4> wsreset;
SETUP CNESS TRUE <5> wsreset;

{ connections into function network and display tree }
{ translation reset }
CONN wsreset <2>:<1> Tran_Acc;
CONN wsreset <2>:<2> Tran_Acc;
```

```
{ rotation reset }
CONN wsreset <3>:<1> world.rot;
CONN wsreset <3>:<1> WSROTATE;
CONN wsreset <3>:<2> WSROTATE;

{ scale reset }
CONN wsreset <4>:<2> PRSCALE;
CONN wsreset <4>:<3> PRSCALE;
CONN wsreset <5>:<1> PRSCALE;

{ initial values to function }
SEND V3D(0,0,0)              TO <2> wsreset;
SEND M3D(1,0,0 0,1,0 0,0,1) TO <3> wsreset;
SEND 1 TO <4> wsreset;
SEND 0 TO <5> wsreset;

{ network to adjust back clipping plane }
backclip := f:fov;
BCKCLP_ACC := f:accumulate;
  CONN dialmux4 <1>:<1> BCKCLP_ACC;
  CONN BCKCLP_ACC <1>:<4>backclip;
  CONN backclip <1>:<1>split.persp;
  CONN backclip <1>:<1>full.persp;
  setup cness false <4>backclip;
  setup cness true <1>backclip;
  send true to <1>backclip;
  send 45   to <2>backclip;
  send 2.2  to <3>backclip;
  send 3.6 to <2> BCKCLP_ACC;
  send 0   to <3> BCKCLP_ACC;
  send 1   to <4> BCKCLP_ACC;
  send 30  to <5> BCKCLP_ACC;
  send 2.2 to <6> BCKCLP_ACC;

{ reset }
RESET_BCKCLP := F:SYNC(3);
SETUP CNESS TRUE <2> RESET_BCKCLP;
SETUP CNESS TRUE <3> RESET_BCKCLP;
CONN wsreset <1>:<1> RESET_BCKCLP;
CONN RESET_BCKCLP <1>:<1> bitbucket;
CONN RESET_BCKCLP <2>:<1> BCKCLP_ACC;
CONN RESET_BCKCLP <3>:<2> BCKCLP_ACC;
SEND 0   TO <2> RESET_BCKCLP;
SEND 3.6 TO <3> RESET_BCKCLP;
```

```
{ print backclip value }
BCKCLP_PRINT := F:PRINT;
CONN BCKCLP_ACC   <1>:<1> BCKCLP_PRINT;
CONN BCKCLP_PRINT <1>:<1> DLABEL4;

{ network to move the sun }
Sun_Azimuth := f:dyrotate;
Sun_RA := f:dzrotate;

conn dialmux5  <2>:<1> Sun_Azimuth;
conn dialmux6  <2>:<1> Sun_RA;
conn Sun_Azimuth <1>:<1> Sun.Azimuth;
conn Sun_RA      <1>:<1> Sun.RA;

{ print values to dlabels }
Sun_Azimuth_Pr := f:print;
Sun_RA_Pr := f:print;
conn Sun_Azimuth <2>:<1> Sun_Azimuth_Pr;
conn Sun_RA      <2>:<1> Sun_RA_Pr;
conn Sun_Azimuth_Pr <1>:<1> Dlabel5;
conn Sun_RA_Pr      <1>:<1> dlabel6;

send 50 to <3> Sun_Azimuth;
send 50 to <3> Sun_RA;

RSSun         := f:sync(3);
setup cness true <2> RSSun;
setup cness true <3> RSSun;
conn RSSun <1>:<1> bitbucket;
conn RSSun <2>:<1> Sun_Azimuth;
conn RSSun <2>:<2> Sun_Azimuth;
conn RSSun <3>:<1> Sun_RA;
conn RSSun <3>:<2> Sun_RA;

send 0 to <2> RSSun;
send 0 to <3> RSSun;

{ network to move the moon }
Moon_Azimuth := f:dyrotate;
Moon_RA := f:dzrotate;

conn dialmux5 <3>:<1> Moon_Azimuth;
conn dialmux6 <3>:<1> Moon_RA;
conn Moon_Azimuth <1>:<1> Moon.Azimuth;
conn Moon_RA      <1>:<1> Moon.RA;
```

```
{ print values to dlabels }
Moon_Azimuth_Pr := f:print;
Moon_RA_Pr := f:print;
conn Moon_Azimuth <2>:<1> Moon_Azimuth_Pr;
conn Moon_RA      <2>:<1> Moon_RA_Pr;
conn Moon_Azimuth_Pr <1>:<1> dlabel5;
conn Moon_RA_Pr      <1>:<1> dlabel6;

send 90 to <2> Moon_Azimuth;
send 50 to <3> Moon_Azimuth;
send 50 to <3> Moon_RA;

RSMoon          := f:sync(3);
setup cness true <2> RSMoon;
setup cness true <3> RSMoon;
conn RSMoon <1>:<1> bitbucket;
conn RSMoon <2>:<2> Moon_Azimuth;
conn RSMoon <3>:<1> Moon_Azimuth;
conn RSMoon <3>:<1> Moon_RA;
conn RSMoon <3>:<2> Moon_RA;

send 90 to <2> RSMoon;
send 0  to <3> RSMoon;

{ scale for light sources }
LSCALE := F:DSCALE;

CONN DIALMUX8 <2>:<1> LSCALE;
CONN DIALMUX8 <3>:<1> LSCALE;
CONN LSCALE <1>:<1> Sun.scale;
CONN LSCALE <1>:<1> Moon.scale;
CONN LSCALE <2>:<3> LSCALE;

send 1  to <2> lscale;
send 1  to <3> lscale;
send 5  to <4> lscale;
send .25 to <5> lscale;

{ print values to dlabels }
Lite_Scale_Pr := f:print;
conn LSCALE <2>:<1> Lite_Scale_Pr;
conn Lite_Scale_Pr <1>:<1> dlabel8;
```

```
{ resets scale of lites }
LRESETER    := F:SYNC(3);
setup cness true <2> lreseter;
setup cness true <3> lreseter;

CONN LRESETER <2>:<1> LSCALE;
CONN LRESETER <3>:<2> LSCALE;
CONN LRESETER <3>:<3> LSCALE;

{ initial values to functions }
SEND 0 TO <2> LRESETER;
SEND 1 TO <3> LRESETER;

{ color network }
tripcolor := f:sync(5);
  setup cness true <2>tripcolor;
  setup cness true <3>tripcolor;
  setup cness true <4>tripcolor;
  setup cness true <5>tripcolor;
  CONN tripcolor<2>:<3>shadingenvironment;  { static viewport }
  CONN tripcolor<3>:<7>shadingenvironment;  { screen wash }
  CONN tripcolor<4>:<3>shadingenvironment;  { static viewport }
  CONN tripcolor<5>:<2>shadingenvironment;  { background color }
  send v3d(975,815,48) to <2>tripcolor;     { location of color box }
  send false to <3>tripcolor;
  send v3d(0,0,0) to <5>tripcolor;

blackbox := f:constant;
  CONN blackbox<1>:<2>shadingenvironment;
  CONN blackbox<1>:<1>tripcolor;
  send v3d(0,0,0) to <2>blackbox;

{ network to change sun color }
sunhue := f:xvector;
sunsat := f:yvector;
sunint := f:zvector;
suncolor := f:accumulate;
  CONN dialmux1 <2>:<1> sunhue;
  CONN dialmux2 <2>:<1> sunsat;
  CONN dialmux3 <2>:<1> sunint;
  CONN sunhue <1>:<1> suncolor;
  CONN sunsat <1>:<1> suncolor;
  CONN sunint <1>:<1> suncolor;
  CONN suncolor<1>:<2>sunlight;
  CONN suncolor<1>:<2>shadingenvironment;
  CONN suncolor<1>:<1>tripcolor;
```

```
  send v3d(0,0,1)      to <2>suncolor;
  send 0               to <3>suncolor;
  send v3d(20,.25,.25) to <4>suncolor;
  send v3d(360,1,1)    to <5>suncolor;
  send v3d(0,0,0)      to <6>suncolor;

{ print sun color values }
sunparts := f:parts;
sunhuepr := f:print;
sunsatpr := f:print;
sunintpr := f:print;
sunhuest := f:take_string;
  CONN suncolor <1>:<1> sunparts;
  CONN sunparts <1>:<1> sunhuepr;
  CONN sunparts <2>:<1> sunsatpr;
  CONN sunparts <3>:<1> sunintpr;
  CONN sunhuepr <1>:<1> sunhuest;
  CONN sunhuest <1>:<1> dlabel1;
  CONN sunsatpr <1>:<1> dlabel2;
  CONN sunintpr <1>:<1> dlabel3;

  setup cness true <2> sunhuest;
  setup cness true <3> sunhuest;
  send fix(1) to <2> sunhuest;
  send fix(3) to <3> sunhuest;

{ network to change moon color }
moonhue := f:xvector;
moonsat := f:yvector;
moonint := f:zvector;
mooncolor := f:accumulate;
  CONN dialmux1 <3>:<1> moonhue;
  CONN dialmux2 <3>:<1> moonsat;
  CONN dialmux3 <3>:<1> moonint;
  CONN moonhue <1>:<1> mooncolor;
  CONN moonsat <1>:<1> mooncolor;
  CONN moonint <1>:<1> mooncolor;
  CONN mooncolor <1>:<2> moonlight;
  CONN mooncolor <1>:<2> shadingenvironment;
  CONN mooncolor <1>:<1> tripcolor;
  send v3d(0,0,1)      to <2>mooncolor;
  send 0               to <3>mooncolor;
  send v3d(20,.25,.25) to <4>mooncolor;
  send v3d(360,1,1)    to <5>mooncolor;
  send v3d(0,0,0)      to <6>mooncolor;
```

```
{ print moon color values }
moonparts := f:parts;
moonhuepr := f:print;
moonsatpr := f:print;
moonintpr := f:print;
moonhuest := f:take_string;
  CONN mooncolor <1>:<1> moonparts;
  CONN moonparts <1>:<1> moonhuepr;
  CONN moonparts <2>:<1> moonsatpr;
  CONN moonparts <3>:<1> moonintpr;
  CONN moonhuepr <1>:<1> moonhuest;
  CONN moonhuest <1>:<1> dlabel1;
  CONN moonsatpr <1>:<1> dlabel2;
  CONN moonintpr <1>:<1> dlabel3;
  setup cness true <2> moonhuest;
  setup cness true <3> moonhuest;
  send fix(1) to <2> moonhuest;
  send fix(3) to <3> moonhuest;

{ network to change back color }
backhue := f:xvector;
backsat := f:yvector;
backint := f:zvector;
backcolor := f:accumulate;
  CONN dialmux1 <4>:<1> backhue;
  CONN dialmux2 <4>:<1> backsat;
  CONN dialmux3 <4>:<1> backint;
  CONN backhue <1>:<1> backcolor;
  CONN backsat <1>:<1> backcolor;
  CONN backint <1>:<1> backcolor;
  CONN backcolor <1>:<2> shadingenvironment;
  CONN backcolor <1>:<1> tripcolor;
  CONN backcolor <1>:<5> tripcolor;
  send v3d(0,0,0)      to <2> backcolor;
  send 0               to <3>backcolor;
  send v3d(20,.25,.25) to <4>backcolor;
  send v3d(360,1,1)    to <5>backcolor;
  send v3d(0,0,0)      to <6>backcolor;

{ print back color values }
backparts := f:parts;
backhuepr := f:print;
backsatpr := f:print;
backintpr := f:print;
backhuest := f:take_string;
  CONN backcolor <1>:<1> backparts;
```

```
  CONN backparts <1>:<1> backhuepr;
  CONN backparts <2>:<1> backsatpr;
  CONN backparts <3>:<1> backintpr;
  CONN backhuepr <1>:<1> backhuest;
  CONN backhuest <1>:<1> dlabel1;
  CONN backsatpr <1>:<1> dlabel2;
  CONN backintpr <1>:<1> dlabel3;
  setup cness true <2> backhuest;
  setup cness true <3> backhuest;
  send fix(1) to <2> backhuest;
  send fix(3) to <3> backhuest;

{ reset of background color }
rsback := f:constant;
  CONN rsback <1>:<1> backcolor;
  CONN rsback <1>:<2> backcolor;
  send v3d(0,0,0) to <2> rsback;

{ network to initialize dlabels }
InitDlabs := f:routec(4);
ObjDials  := f:sync(4);
SunDials  := f:sync(4);
MoonDials := f:sync(4);
BackDials := f:sync(3);
conn Initdlabs <1>:<4> ObjDials;
conn Initdlabs <2>:<4> SunDials;
conn Initdlabs <3>:<4> MoonDials;
conn Initdlabs <4>:<3> BackDials;
send 'init labels' to <2> Initdlabs;

conn ObjDials <1>:<1> Tran_Acc;
conn ObjDials <3>:<1> Dlabel5;
conn ObjDials <3>:<1> Dlabel6;
conn ObjDials <3>:<1> Dlabel7;
conn ObjDials <2>:<1> ObjScalePr;
conn ObjDials <2>:<1> bckclp_acc;
conn ObjDials <4>:<1> BitBucket;
setup cness true <1> ObjDials;
setup cness true <2> ObjDials;
setup cness true <3> ObjDials;
send v3d(0,0,0) to <1> ObjDials;
send 0          to <2> ObjDials;
send ' '        to <3> ObjDials;

conn SunDials <1>:<1> SunColor;
conn SunDials <2>:<1> Sun_Azimuth;
```

```
conn SunDials <2>:<1> Sun_RA;
conn SunDials <2>:<1> LScale;
conn SunDials <3>:<1> dlabel4;
conn SunDials <3>:<1> dlabel7;
conn SunDials <4>:<1> BitBucket;
send v3d(0,0,0)  to <1> SunDials;
send 0           to <2> SunDials;
send ' '         to <3> SunDials;
setup cness true <1> SunDials;
setup cness true <2> SunDials;
setup cness true <3> SunDials;

conn MoonDials <1>:<1> MoonColor;
conn MoonDials <2>:<1> Moon_Azimuth;
conn MoonDials <2>:<1> Moon_RA;
conn MoonDials <2>:<1> LScale;
conn MoonDials <3>:<1> dlabel4;
conn MoonDials <3>:<1> dlabel7;
conn MoonDials <4>:<1> BitBucket;
send v3d(0,0,0) to <1> MoonDials;
send 0   to <2> MoonDials;
send ' ' to <3> MoonDials;
setup cness true <1> MoonDials;
setup cness true <2> MoonDials;
setup cness true <3> MoonDials;

conn BackDials <1>:<1> BackColor;
conn BackDials <2>:<1> dlabel4;
conn BackDials <2>:<1> dlabel5;
conn BackDials <2>:<1> dlabel6;
conn BackDials <2>:<1> dlabel7;
conn BackDials <2>:<1> dlabel8;
conn BackDials <3>:<1> BitBucket;
send v3d(0,0,0) to <1> BackDials;
send ' ' to <2> BackDials;
setup cness true <1> BackDials;
setup cness true <2> BackDials;

{ the reseter }
reset := f:croute(4);
  CONN reset <1>:<1> wsreset;
  CONN reset <2>:<1> rssun;
  CONN reset <3>:<1> rsmoon;
  CONN reset <4>:<1> rsback;
```

```
{ network to turn bits on and off }
bits := f:constant;
  CONN bits<1>:<5>world.bits;

{ dial labels for following mux }
{ dial #1 labels }
diallabel1 := f:inputs_choose(5);
CONN diallabel1 <1>:<1> dlabel1h;
  send 'X-Trans'  to <1> diallabel1;
  send 'Hue'    to <2> diallabel1;
  send 'Hue'    to <3> diallabel1;
  send 'Hue'    to <4> diallabel1;

{ dial #2 labels }
diallabel2 := f:inputs_choose(5);
CONN diallabel2 <1>:<1> dlabel2h;
  send 'Y-Trans'  to <1> diallabel2;
  send 'Sat'    to <2> diallabel2;
  send 'Sat'    to <3> diallabel2;
  send 'Sat'    to <4> diallabel2;

{ dial #3 labels }
diallabel3 := f:inputs_choose(5);
CONN diallabel3 <1>:<1> dlabel3h;
  send 'Z-Trans'  to <1> diallabel3;
  send 'Intens'   to <2> diallabel3;
  send 'Intens'   to <3> diallabel3;
  send 'Intens'   to <4> diallabel3;

{ dial #4 labels }
diallabel4 := f:inputs_choose(5);
CONN diallabel4 <1>:<1> dlabel4h;
  send 'Backclip' to <1> diallabel4;
  send ' '    to <2> diallabel4;
  send ' '    to <3> diallabel4;
  send '      '   to <4> diallabel4;

{ dial #5 labels }
diallabel5 := f:inputs_choose(5);
CONN diallabel5 <1>:<1> dlabel5h;
  send '  X-Rot'  to <1> diallabel5;
  send 'Azimuth'  to <2> diallabel5;
  send 'Azimuth'  to <3> diallabel5;
  send '      '   to <4> diallabel5;
```

```
{ dial #6 labels }
diallabel6 := f:inputs_choose(5);
CONN diallabel6 <1>:<1> dlabel6h;
  send '  Y-Rot'  to <1> diallabel6;
  send 'RIGHT A.' to <2> diallabel6;
  send 'RIGHT A.' to <3> diallabel6;
  send '      '   to <4> diallabel6;

{ dial #7 labels }
diallabel7 := f:inputs_choose(5);
CONN diallabel7 <1>:<1> dlabel7h;
  send '  Z-Rot'  to <1> diallabel7;
  send '      '   to <2> diallabel7;
  send '      '   to <3> diallabel7;
  send '      '   to <4> diallabel7;

{ dial #8 labels }
diallabel8 := f:inputs_choose(5);
CONN diallabel8 <1>:<1> dlabel8h;
  send 'SCALE' to <1> diallabel8;
  send 'SCALE' to <2> diallabel8;
  send 'SCALE' to <3> diallabel8;
  send '      ' to <4> diallabel8;

{ way - network to control dials for }
{ objects, sun, moon, and background color }
way     := F:CONSTANT;
waytrip := F:ADDC;
waymod  := F:MODC;
wayadd  := F:ADDC;
wayvals := F:INPUTS_CHOOSE(5);
waylabs := F:INPUTS_CHOOSE(5);

CONN WAY     <1>:<1> WAYTRIP;
CONN WAYTRIP <1>:<1> WAYMOD;
CONN WAYMOD  <1>:<1> WAYADD;
CONN WAYMOD  <1>:<2> WAYTRIP;
CONN WAYADD  <1>:<5> WAYVALS;
CONN WAYADD  <1>:<5> WAYLABS;
CONN WAYVALS <1>:<1> dialmux1;
CONN WAYVALS <1>:<1> dialmux2;
CONN WAYVALS <1>:<1> dialmux3;
CONN WAYVALS <1>:<1> dialmux4;
CONN WAYVALS <1>:<1> dialmux5;
CONN WAYVALS <1>:<1> dialmux6;
```

```
CONN WAYVALS <1>:<1> dialmux7;
CONN WAYVALS <1>:<1> dialmux8;
CONN WAYVALS <1>:<5> diallabel1;
CONN WAYVALS <1>:<5> diallabel2;
CONN WAYVALS <1>:<5> diallabel3;
CONN WAYVALS <1>:<5> diallabel4;
CONN WAYVALS <1>:<5> diallabel5;
CONN WAYVALS <1>:<5> diallabel6;
CONN WAYVALS <1>:<5> diallabel7;
CONN WAYVALS <1>:<5> diallabel8;
CONN WAYVALS <1>:<1> reset;
CONN WAYVALS <1>:<2> bits;
CONN WAYVALS <1>:<1> Initdlabs;
CONN WAYLABS <1>:<1> FLABEL9;
CONN WAYLABS <1>:<1> FLABEL10;
CONN WAYLABS <1>:<1> FLABEL11;

  send fix(1) to <1>wayvals;
  send fix(2) to <2>wayvals;
  send fix(3) to <3>wayvals;
  send fix(4) to <4>wayvals;
  send '  OBJECT' to <1>waylabs;
  send '  SUN' to <2>waylabs;
  send '  MOON' to <3>waylabs;
  send '  BACK' to <4>waylabs;

send FIX(1)  TO <2> way;
send FIX(0)  TO <2> waytrip;
send FIX(4)  TO <2> waymod;
send FIX(1)  TO <2> wayadd;

{ network to control rendering style }
renstyle := F:CONSTANT;
stytrip := F:ADDC;
stymod  := F:MODC;
styadd  := F:ADDC;
styvals := F:INPUTS_CHOOSE(6);
stylabs := F:INPUTS_CHOOSE(6);
style   := F:CONSTANT;

CONN renstyle <1>:<1> stytrip;
CONN stytrip  <1>:<1> stymod;
CONN stymod   <1>:<1> styadd;
CONN stymod   <1>:<2> stytrip;
CONN styadd   <1>:<6> styvals;
```

```
CONN styadd   <1>:<6> stylabs;
CONN styvals  <1>:<2> style;
CONN style    <1>:<1> world.rendering;
CONN stylabs  <1>:<1> FLABEL2;

  send '  FLAT'       to <1> stylabs;
  send '  GOURAUD'    to <2> stylabs;
  send '  PHONG '     to <3> stylabs;
  send '  HIDDENLINE' to <4> stylabs;
  send '  WASH'       to <5> stylabs;
  send fix(6) to <1> styvals;      { flat }
  send fix(8) to <2> styvals;      { gouraud }
  send fix(7) to <3> styvals;      { phong }
  send fix(4) to <4> styvals;      { static hiddenline }
  send fix(5) to <5> styvals;      { wash }

SEND FIX(1)  TO <2> renstyle;
SEND FIX(0)  TO <2> stytrip;
SEND FIX(5)  TO <2> stymod;
SEND FIX(1)  TO <2> styadd;

{ type - network to choose type of rendering }
{ 1 - no anti-aliasing z-buffer }
{ 2 - edge anti-aliasing z-buffer }
{ 3 - full anti-aliasing z-buffer }
{ 4 - no anti-aliasing painters }
Type   := F:CONSTANT;
TypeTRIP := F:ADDC;
TypeMOD  := F:MODC;
TypeADD  := F:ADDC;
TypeVals := F:INPUTS_CHOOSE(5);
TypeNums := F:INPUTS_CHOOSE(5);
TypeLabs := F:INPUTS_CHOOSE(5);
TypeHead := F:INPUTS_CHOOSE(5);

CONN type     <1>:<1> typeTRIP;
CONN typeTRIP <1>:<1> typeMOD;
CONN typeMOD  <1>:<1> typeADD;
CONN typeMOD  <1>:<2> typeTRIP;
CONN typeADD  <1>:<5> typeVALS;
CONN typeADD  <1>:<5> typeNums;
CONN typeADD  <1>:<5> typeLABS;
CONN typeADD  <1>:<5> typeHead;
CONN typeVALS <1>:<5> shadingenvironment;
CONN typeNums <1>:<16> shadingenvironment;
```

```
CONN TypeLABS <1>:<1> FLABEL3;
CONN TypeHead <1>:<1> FLABEL3H;

  send fix(0) to <1> TypeVals;
  send fix(1) to <2> TypeVals;
  send fix(2) to <3> TypeVals;
  send fix(0) to <4> TypeVals;
  send fix(0) to <1> TypeNums;
  send fix(0) to <2> TypeNums;
  send fix(0) to <3> TypeNums;
  send fix(1) to <4> TypeNums;
  send '  NO AA  '  to <1> TypeLabs;
  send '  EDGE AA'  to <2> TypeLabs;
  send '  FULL AA'  to <3> TypeLabs;
  send '  NO AA'    to <4> TypeLabs;
  send '  ZBUFFER'  to <1> TypeHead;
  send '  ZBUFFER'  to <2> TypeHead;
  send '  ZBUFFER'  to <3> TypeHead;
  send '  PAINTERS' to <4> TypeHead;

SEND FIX(1)  TO <2> type;
SEND FIX(0)  TO <2> typeTRIP;
SEND FIX(4)  TO <2> typeMOD;
SEND FIX(1)  TO <2> typeADD;

{ pic - raster viewports }
pic     := F:CONSTANT;
pictrip := F:ADDC;
picmod  := F:MODC;
picadd  := F:ADDC;
picvals := F:INPUTS_CHOOSE(6);
piclabs := F:INPUTS_CHOOSE(6);
splitview := F:INPUTS_CHOOSE(6);

CONN pic     <1>:<1> pictrip;
CONN pictrip <1>:<1> picmod;
CONN picmod  <1>:<1> picadd;
CONN picmod  <1>:<2> pictrip;
CONN picadd  <1>:<6> picvals;
CONN picadd  <1>:<6> piclabs;
CONN picadd  <1>:<6> splitview;
CONN picvals <1>:<3> shadingenvironment; {setup static viewport}
CONN picvals <1>:<4> tripcolor;          {setup tripcolor for
                                           background}
CONN piclabs <1>:<1> FLABEL4;
```

```
CONN splitview <1>:<1> universe.split;

  send '  SQUARE '   to <1> piclabs;
  send 'UPPER RIGHT' to <2> piclabs;
  send 'UPPER LEFT'  to <3> piclabs;
  send 'LOWER LEFT'  to <4> piclabs;
  send '  BIG-PIC '  to <5> piclabs;

  send v3d (170,10,853)  to <1> picvals;  { larger viewport sizes }
  send v3d (592,432, 431) to <2> picvals; { for upper right corner }
  send v3d (160,432,431)  to <3> picvals; { for upper left corner }
  send v3d (160,0,431)    to <4> picvals; { for lower left corner }
  send v3d (0,-80,1023)   to <5> picvals; { big pic }

  send fix(0) to <1> splitview;
  send fix(1) to <2> splitview;
  send fix(1) to <3> splitview;
  send fix(1) to <4> splitview;
  send fix(0) to <5> splitview;

SEND FIX(1)  TO <2> pic;
SEND FIX(0)  TO <2> pictrip;
SEND FIX(5)  TO <2> picmod;
SEND FIX(1)  TO <2> picadd;

{ polygon edge enhancement }
{ network to toggle polyedges (on/off) }
PolygonEdge := F:CONSTANT;
petrip := F:ADDC;
pemod  := F:MODC;
peadd  := F:ADDC;
pevals := F:INPUTS_CHOOSE(3);
pelabs := F:INPUTS_CHOOSE(3);

CONN PolygonEdge <1>:<1> petrip;
CONN petrip <1>:<1> pemod;
CONN pemod  <1>:<1> peadd;
CONN pemod  <1>:<2> petrip;
CONN peadd  <1>:<3> pevals;
CONN peadd  <1>:<3> pelabs;
CONN pevals <1>:<15> shadingenvironment;
CONN pelabs <1>:<1> FLABEL6;

SEND FALSE TO <1> pevals;
SEND TRUE  TO <2> pevals;
```

```
{ label selector }
SEND ´  OFF´ TO     <1> pelabs;
SEND ´  ON ´ TO     <2> pelabs;

SEND FIX(1)  TO <2> PolygonEdge;
SEND FIX(0)  TO <2> petrip;
SEND FIX(2)  TO <2> pemod;
SEND FIX(1)  TO <2> peadd;

{ network to toggle color interpolation
{ across polygon verticies }
VertexColor := F:CONSTANT;
vctrip := F:ADDC;
vcmod  := F:MODC;
vcadd  := F:ADDC;
vcvals := F:INPUTS_CHOOSE(3);
vclabs := F:INPUTS_CHOOSE(3);

CONN VertexColor <1>:<1> vctrip;
CONN vctrip <1>:<1> vcmod;
CONN vcmod  <1>:<1> vcadd;
CONN vcmod  <1>:<2> vctrip;
CONN vcadd  <1>:<3> vcvals;
CONN vcadd  <1>:<3> vclabs;
CONN vcvals <1>:<10> shadingenvironment;
CONN vclabs <1>:<1> FLABEL7;

SEND FALSE TO <1> vcvals;
SEND TRUE  TO <2> vcvals;

{ label selector }
SEND ´  OFF´ TO     <1> vclabs;
SEND ´  ON ´ TO     <2> vclabs;

SEND FIX(1)  TO <2> VertexColor;
SEND FIX(0)  TO <2> vctrip;
SEND FIX(2)  TO <2> vcmod;
SEND FIX(1)  TO <2> vcadd;

{ transparency switch }
Transparency := F:CONSTANT;
TPtrip := F:ADDC;
TPmod  := F:MODC;
TPadd  := F:ADDC;
```

```
TPvals := F:INPUTS_CHOOSE(3);
TPlabs := F:INPUTS_CHOOSE(3);

CONN Transparency <1>:<1> TPtrip;
CONN TPtrip <1>:<1> TPmod;
CONN TPmod  <1>:<1> TPadd;
CONN TPmod  <1>:<2> TPtrip;
CONN TPadd  <1>:<3> TPvals;
CONN TPadd  <1>:<3> TPlabs;
CONN TPvals <1>:<11> Shadingenvironment;
CONN TPlabs <1>:<1> FLABEL8;

SEND FALSE TO <1> TPvals;
SEND TRUE  TO <2> TPvals;

{ label selector }
SEND ´  OFF´ TO     <1> TPlabs;
SEND ´  ON ´ TO     <2> TPlabs;

SEND FIX(1)  TO <2> Transparency;
SEND FIX(0)  TO <2> TPtrip;
SEND FIX(2)  TO <2> TPmod;
SEND FIX(1)  TO <2> TPadd;

{ turn on wireframe object and clear screen }
wireframe := F:SYNC(4);
SETUP CNESS TRUE <2> wireframe;
SETUP CNESS TRUE <3> wireframe;
SETUP CNESS TRUE <4> wireframe;
CONN wireframe <1>:<1> bitbucket;
CONN wireframe <2>:<2> world.bits;
CONN wireframe <3>:<7> shadingenvironment;
CONN wireframe <4>:<1> FLABEL5;

SEND FIX(1) TO <2> wireframe;
SEND FIX(2) TO <3> wireframe;
SEND ´  ON´ TO <4> wireframe;

{ turn on ACP after rendering }
{ use the output from the rendering node }
TurnOnACP := F:CONSTANT;
TurnOffObj := F:CONSTANT;
CONN world.rendering <1>:<1> TurnOnACP;
CONN world.rendering <1>:<1> TurnOffObj;
CONN TurnOnACP <1>:<1> TurnOnDisplay;
```

```
CONN TurnOffObj <1>:<3> world.bits;
SEND FIX(0) TO <2> TurnOnACP;
SEND FIX(1) TO <2> TurnOffObj;

{ toggle display of object and label }
togobj   := F:CONSTANT;
togtrip  := F:ADDC;
togmod   := F:MODC;
togadd   := F:ADDC;
togvals  := F:INPUTS_CHOOSE(6);
toglabs  := F:INPUTS_CHOOSE(6);
routebits := F:BROUTEC;
togworld := F:CONSTANT;
togflab5 := F:CONSTANT;

CONN togobj  <1>:<1> togtrip;
CONN togtrip <1>:<1> togmod;
CONN togmod  <1>:<1> togadd;
CONN togmod  <1>:<2> togtrip;
CONN togadd  <1>:<6> togvals;
CONN togadd  <1>:<6> toglabs;
CONN world.rendering <1>:<1> togworld; { conn from rendering node }
CONN togvals <1>:<2> togworld;
CONN togworld <1>:<1> routebits;
CONN routebits <1>:<2> world.bits;
CONN routebits <2>:<3> world.bits;
CONN world.rendering <1>:<1> togflab5; { conn from rendering node }
CONN toglabs <1>:<2> togflab5;
CONN togflab5 <1>:<1> FLABEL5;

SEND FALSE TO <1> togvals;
SEND TRUE  TO <2> togvals;
SEND TRUE  TO <3> togvals;
SEND TRUE  TO <4> togvals;
SEND FALSE TO <5> togvals;
SEND FIX(1) TO <2> routebits;

{ label selector }
SEND '  OFF' TO <1> toglabs;
SEND '  ON ' TO <2> toglabs;
SEND '  ON ' TO <3> toglabs;
SEND '  ON ' TO <4> toglabs;
SEND '  OFF' TO <5> toglabs;

SEND FIX(1)  TO <2> togobj;
```

```
SEND FIX(0)  TO <2> togtrip;
SEND FIX(5)  TO <2> togmod;
SEND FIX(1)  TO <2> togadd;

{ 12 function key router }
KEY_CHOOSE := F:INPUTS_CHOOSE(13);
KEY_ROUTE  := F:ROUTE(12);

{ connections }
CONN FKEYS <1>:<13> KEY_CHOOSE;
CONN FKEYS <1>:<1>  KEY_ROUTE;
CONN KEY_CHOOSE <1>:<2> KEY_ROUTE;

{ fkey # 1 - render key }
  CONN key_route <1>:<1> style;

{ fkey # 2 - rendering style }
  CONN key_route <2>:<1> renstyle;

{ fkey # 3 - anti-alias }
  CONN key_route <3>:<1> type;

{ fkey # 4 - raster viewport }
  CONN key_route <4>:<1> pic;
  CONN key_route <4>:<1> togobj;

{ fkey # 5 - wireframe }
  CONN key_route <5>:<1> wireframe;

{ fkey # 6 - edges of polygons }
  CONN key_route <6>:<1> PolygonEdge;

{ fkey # 7 - vertex color on }
  CONN key_route <7>:<1> VertexColor;

{ fkey # 8 - transparency }
  CONN key_route <8>:<1> Transparency;

{ fkey # 9 - objects for dial use }
  CONN key_route <9>:<1> way;

{ fkey # 10 - objects on/off }
  CONN key_route <10>:<1> bits;
```

```
{ fkey # 11 - reset object }
  CONN key_route <11>:<2> reset;

{ fkey # 12 - exit network }
var exit;
  CONN key_route <12>:<1> exit;

{ initial values for key choose }
  send 'RENDER'          to <1> key_choose;
  send 'STYLE'           to <2> key_choose;
  send 'type'            to <3> key_choose;
  send 'R VIEWPORT'      to <4> key_choose;
  send 'WIREFRAME'       to <5> key_choose;
  send 'POLYGON EDGES'   to <6> key_choose;
  send 'VETREX COLOR' to <7> key_choose;
  send 'TRANSPARENCY'    to <8> key_choose;
  send 'WAY'             to <9> key_choose;
  send 'BITS'     to <10> key_choose;
  send v3d(0,0,0) to <11> key_choose;1  send 'EXIT'    to <12>
key_choose;

{ trip headers }
TripHeaders := F:SYNC(15);
SETUP CNESS TRUE <1> TripHeaders;
SETUP CNESS TRUE <2> TripHeaders;
SETUP CNESS TRUE <3> TripHeaders;
SETUP CNESS TRUE <4> TripHeaders;
SETUP CNESS TRUE <5> TripHeaders;
SETUP CNESS TRUE <6> TripHeaders;
SETUP CNESS TRUE <7> TripHeaders;
SETUP CNESS TRUE <8> TripHeaders;
SETUP CNESS TRUE <9> TripHeaders;
SETUP CNESS TRUE <10> TripHeaders;
SETUP CNESS TRUE <11> TripHeaders;
SETUP CNESS TRUE <12> TripHeaders;
SETUP CNESS TRUE <13> TripHeaders;
SETUP CNESS TRUE <14> TripHeaders;
CONN TripHeaders <1>:<1> FLABEL1H;
CONN TripHeaders <2>:<1> FLABEL2H;
CONN TripHeaders <3>:<1> BitBucket;
CONN TripHeaders <4>:<1> FLABEL4H;
CONN TripHeaders <5>:<1> FLABEL5H;
CONN TripHeaders <6>:<1> FLABEL6H;
CONN TripHeaders <7>:<1> FLABEL7H;
CONN TripHeaders <8>:<1> FLABEL8H;
```

```
CONN TripHeaders <9>:<1> FLABEL9H;
CONN TripHeaders <10>:<1> FLABEL10H;
CONN TripHeaders <11>:<1> FLABEL11H;
CONN TripHeaders <12>:<1> FLABEL12H;
CONN TripHeaders <13>:<1> FLABEL1;
CONN TripHeaders <14>:<1> FLABEL12;
CONN TripHeaders <15>:<1> BitBucket;
SEND 'TRIGGER'      TO <1> TripHeaders;
SEND 'STYLE'        TO <2> TripHeaders;
SEND ' '    TO <3> TripHeaders;
SEND 'R. VIEWPORT'  TO <4> TripHeaders;
SEND 'WIREFRAME'    TO <5> TripHeaders;
SEND 'POLY EDGES'   TO <6> TripHeaders;
SEND 'VERTEX COLOR' TO <7> TripHeaders;
SEND 'TRANSPERANCY' TO <8> TripHeaders;
SEND 'DIAL MUX'     TO <9> TripHeaders;
SEND 'TOGGLE '      TO <10> TripHeaders;
SEND 'RESET'        TO <11> TripHeaders;
SEND 'EXIT'         TO <12> TripHeaders;
SEND 'RENDERING'    TO <13> TripHeaders;
SEND 'NETWORK'      TO <14> TripHeaders;

{ shadingenvironment considerations }
SEND V3D(0,0,.1) to <1> ShadingEnvironment;

{ tripping network to initialize rendering network }
TRIPPER := F:CONSTANT;
SEND FIX(0) TO <2> TRIPPER;

{ connections from tripper to initialize fkeys and labels }
CONN TRIPPER <1>:<1> wayTRIP;
CONN TRIPPER <1>:<1> stytrip;
CONN TRIPPER <1>:<1> typeTRIP;
CONN TRIPPER <1>:<1> picTRIP;
CONN TRIPPER <1>:<1> peTRIP;
CONN TRIPPER <1>:<1> vcTRIP;
CONN TRIPPER <1>:<1> TPTRIP;
CONN TRIPPER <1>:<1> togTRIP;
CONN TRIPPER <1>:<1> wireframe;
CONN TRIPPER <1>:<15> TripHeaders;

{ Trip the thing }
send 'surf' to <1> Tripper;
```

Section GT16
Glossary

ACP. *See* Arithmetic Control Processor.

Active input. One of two types of function inputs. A value on an active input disappears or is consumed when the function fires. If values arrive on an active input faster than they are consumed, they will queue in the order they arrive. *See also* Constant input.

Alternating display. A type of conditional referencing which is the alternate displaying of two different objects. *See also* Blinking.

Ambient light. Refers to light surrounding an object and coming from all directions. The lighting is the same everywhere on the object.

Antialiasing. A technique used to smooth the jagged appearance of lines on a raster display. *See also* Shading.

Application routine. One of two types of Graphics Support Routines. Application routines correspond almost one for one with the standard PS 390 commands. *See also* Utility routine.

Arithmetic Control Processor. *Also called* ACP. A subsystem in the Display Processor of the PS 390. The ACP is the master controller for Display Processor input. It includes a microprocessor that performs arithmetic and logical functions on data in Mass Memory. The ACP traverses display structures; performs matrix multiplications for rotations, scaling, and windowing; then applies the matrices to the data nodes; and outputs the transformed coordinates of the data to the Pipeline Subsystem. The ACP also performs clipping. The state of the ACP is considered to be those values that are context dependent, such as transformation matrix contents, viewport boundaries, and color.

Aspect ratio. The ratio of width to height (X:Y). The aspect ratios of viewing areas and viewports in which they are displayed must be the same, or objects will appear distorted on the display screen.

Asynchronous serial line. A data communication interface between the host and the PS 390. The arrival time of each character is random (asynchronous). The bits that represent a character are sent one after the other (serially). Each character is delineated by the use of a start bit and one or more stop bits (start/stop protocol).

Attribute. A characteristic or aspect of a displayed image. Attribute setting commands set and change these aspects. Unlike transformations, attributes are not matrix operations. There are three classes of attributes: appearance attributes, picking attributes, and structure attributes. **Appearance attributes** govern the following aspects of the displayed image: color and intensity of the lines that form the image, depth-clipping on the image, and character font for any text in the image. **Picking attributes** allow the user to mark objects or parts of objects as candidates for picking, to turn picking on or off, and to assign a name (pick ID) which will be reported as a text string when a pick occurs. **Structure attributes** create nodes in a display structure at which branching may occur. These attributes allow the user to reference objects or parts of objects by setting conditional bits to add or remove detail from an object, to control blinking, and to control the alternating display of more than one image.

Attribute table. Stores color (hue, intensity, saturation), radius, diffuse and specular attributes.

B-spline. A mathematical representation of a curve which approximates a specified set of points.

Backface removal. A rendering operation that removes all polygons facing away from the viewer, and which is an intermediate step in hidden-line removal. Backface removal is displayed in a dynamic viewport. *See also* Hidden-line removal.

Blinking. A type of conditional referencing which is the alternate displaying and blanking of an image at a selectable rate. The rate may be under the control of the PS 390 update rate or a clock. *See also* Alternating display; Conditional referencing.

Block. A group of either data structures or messages in Mass Memory.

Block-normalized vectors. When the components of all vectors that comprise a vector list have a common exponent, they are said to be block-normalized vectors. Only block-normalized vectors are displayed on the PS 390. *See also* Vector-normalized vectors.

Boolean value. A Boolean value is either true or false. Boolean values are types of data which may be sent to a node or function.

Boundaries, front and back. *See* Clipping planes.

Bounded plane. *See* Surface.

Branches. Connections between nodes in a PS 390 display structure. Branches determine the paths the Arithmetic Control Processor must take when it traverses the structure of the object in memory; i.e. when the object is displayed. *See also* Pointer.

Character. A letter or digit or other symbol. The CHARACTERS and LABELS commands allow the user to display text as character strings or blocks of labels. By using optional attribute operations in the structure, characters can be transformed in the same way as any other graphical data, or they may be oriented to the screen and fixed at a certain size and intensity. *See also* Character data; Character string.

Character data. Consists of an initial translation, spacing information, and the character string. This data may be drawn by the Arithmetic Control Processor if the data is part of a data node (also called a characters node). *See also* Character; Character string; Label.

Character font. A set of characters of a particular style (size and type face). A 256 ASCII character set is provided as the standard font for the PS 390. Alternate fonts may be created by using BEGIN...END FONT and CHARACTER FONT commands, or by using the Character Font Editor, MAKEFONT.

Character string. *Also called* Text string. A sequence of up to 240 characters. Strings can be created and manipulated with commands and manipulated interactively using function networks and interactive devices. If text is created using CHARACTERS command, the user can manipulate any character in the string. *See also* Character; Character data; Label.

CI. *See* Command Interpreter.

Clipping. A viewing operation that eliminates the lines or parts of lines that are outside the boundaries of the viewing area. *See also* Depth clipping; Viewing area.

Clipping planes. The six boundary planes that define a viewing area. The user can only interact with the front and back faces of the viewing area, which are located along the Z-axis. The front face is called the Hither plane or Front boundary, and the back face is called the Yon plane or Back boundary.

Color. Color is specified by hue, saturation and intensity. Hue is the shade of color which distinguishes a particular color from all other colors. Saturation is the ratio of the selected hue to white and ranges from no saturation (grays) to fully saturated hue. Intensity is a measure of the brightness of the color; no intensity is black. Any displayable color is a combination of the primary phosphor colors: red, green and blue. Varying the hue, saturation, and intensity of these three color components produces the wide variety of colors available to the raster display.

Color lookup table. *Also called* CLUT. An array of color values that defines the red, green and blue (RGB) components of pixels.

Command. Using PS 390 commands, the user can build display structures that represent the objects and models the user wants to display and manipulate. In addition to creating and modifying display structures, the PS 390 commands create and modify function networks, instruct the Display Processor to display items or remove them from the display list, and query or reset the Command Interpreter. PS 390 commands are not stored in memory; they are interpreted and either execute immediately or create display structure elements in Mass Memory.

Command Interpreter. *Also called* CI. The CI is a PS 390 intrinsic system function that checks the validity of commands and puts them into effect. The PS 390 expects messages (tokens) which consist of a size, a data type, and a value. Once given, the type of command is implicit in the type of the token. The CI accepts tokens until it has enough to carry out a command. At that point, the CI passes all required data to (and fires/triggers) the appropriate function. The CI in the command mode handles user commands. The CI in the configure mode can access intrinsic system functions.

Command mode. *Also called* CI mode; Local command mode. One of three types of communication modes available on any style PS 390 keyboard. Command mode implies that commands entered locally on the PS 390 keyboard (as opposed to data received from the host) are to be routed to the Command Interpreter. Command mode displays the "@@" prompt on the screen. *See also* Local mode; Terminal Emulator mode.

Communication mode. *See* Command mode; Local mode; Terminal Emulator mode.

Concatenate. *See* Transformation matrix.

Conditional bit. Any one of 15 (0-14) bits which may be set on or off in order to conditionally reference for display selected branches of a display structure.

Conditional referencing. The referencing of data only when certain conditions are met. Conditional referencing is used to selectively display or blank parts of a display structure. *See also* Alternating Display; Blinking; Level-of-detail.

Configure mode. A privileged mode of operation. In this mode the user may reconfigure intrinsic system functions and has access to functions previously created in configure mode. The Command Interpreter is in this mode while reading the CONFIG.DAT file from the graphics firmware diskette. *See also* Suffix.

Conjunctive/disjunctive. All PS 390 functions have conjunctive or disjunctive inputs and outputs. A function with conjunctive inputs must have a message on every input before it will fire. A function with conjunctive outputs will send a message on every output when the function is fired. A function with disjunctive inputs does not require a message on every input to fire. A function with disjunctive outputs might not send a message on every output every time the function is fired.

Constant input. One of two types of function inputs. Constant inputs hold only one value at a time; i.e., there is no queuing. A value on a constant input is not consumed when the function fires. It will remain until it is overwritten by another value. *See also* Active input.

Contour. *See* Inner contour; Outer contour.

Coordinate system. *Also called* World coordinate system. A way of specifying a three-dimensional space in which objects can be modeled. The designer creates an object by entering, in the conventional order of X, Y, Z, mathematical information that defines and locates an object in three-dimensional space. The coordinate system used in programming the PS 390 is a left-handed Cartesian coordinate system, usually referred to as the world coordinate system. *See also* Screen coordinate system.

Coplanar. Denotes that polygons have the same plane equation.

Count mode packet. One type of communication packet sent by the host interface to the PS 390 runtime environment. Count mode packets consist of a Start of Packet character, followed by two bytes of count data, followed by the data itself. This type of packet is generated automatically by the Graphics Support Routines. *See also* Escape mode packet.

Cross sectioning. A rendering operation that uses a sectioning plane to create a cross section of an object. When this operation is used, the object on either side of the plane is discarded and only the slice defined by the sectioning plane remains. Cross sectioning is displayed in a dynamic viewport. *See also* Sectioning.

CRT. *Abbreviation for* cathode-ray tube.

Current state-of-the-machine. *See* State-of-the-machine.

Current Transformation Matrix. *Also called* CTM. *See* Transformation Matrix.

Data driven. This means that a function fires only when data arrives at its inputs to be processed. *See also* Conjunctive/disjunctive.

Data node. A node in a display structure that contains data which defines a primitive or untransformed object or shape. Data nodes, the terminal nodes in a display structure, may contain vector lists, polygons, curves, or text. *See also* Primitive.

Data structure. Display structures and other data. *See also* Display structure.

Data tree. *See* Display structure.

Demultiplexing. The reverse process of multiplexing. *See* Multiplexing.

Depth clipping. *Also called* Z-clipping. A viewing operation that removes from displayed objects or parts of objects anything which extends beyond the front and back clipping planes in a viewing area.

Depth cueing. An operation that imparts an illusion of depth to the image of a three-dimensional object by decreasing the intensity of lines as they recede into the distance (i.e., along the positive Z-axis).

Diagnostic utility commands. Available on diagnostic diskettes and not part of the runtime environment. These commands format diskettes, check, copy, delete, modify, download, and send back files.

Diffuse. An attribute used in shading polygons that specifies the proportion of color contributed by diffuse reflection versus specular reflection. Increasing the diffuse attribute contribution makes the surface more matte. *See also* Specular.

Display list. Contains the names of all display structures that are currently being traversed for display. Whenever anything is displayed, its name goes on the display list.

Display Processor. Accesses display structures in the Mass Memory and in each update cycle traverses them under the control of the Arithmetic Control Processor, transforming the data to be displayed. Performs clipping, perspective projections, and viewport mapping, and draws the data on the raster display screen.

Display structure. *Also called* Data tree; Display tree; Hierarchical data structure. The structure of a model in Mass Memory. A display structure is an ordering of data nodes (primitives), operation nodes (attributes, transformations), and instance nodes connected by branches. Display structures are hierarchically ordered; each node or element is used as a reference to all elements below it. *See also* Data node; Instance node; Operation node.

Distributed graphics. Allows the graphical portion of an application to be performed locally by the graphics system without burdening the host computer.

Dynamic viewport. The viewport in which wireframe graphical models are displayed and manipulated in real time using the interactive devices. In addition, the following static renderings are displayed in a dynamic viewport: backface removal, cross-sectioning and sectioning. Dynamic viewports are defined by viewport operation nodes in the PS 390 display structure. *See also* Backface removal; Cross sectioning; Sectioning; Static viewport.

Escape mode packet. One type of communication packet sent by the host to the PS 390 runtime environment. The escape mode is selected by the user. The escape mode packet consists of a Start of Packet character, followed by the muxing byte, followed by data. *See also* Count mode packet.

Escape sequence. A sequence of characters in an escape mode packet that is used for control purposes, to perform a control function, and whose first character is the ESC control character. Used to set and reset modes, as well as tell the terminal how to respond to coded sequences.

Explicit naming. To code having one named command correspond to each node in the display structure. Explicit naming forces the user to name every node. Explicit naming is also the only way a user can access a single node for display structure manipulation, or node content updating.

Explicit referencing. In a command sequence that builds a display structure, all nodes except data nodes must be explicitly referenced, or point, to other nodes upon which they perform operations or which they group into instances.

Exposure. A shading parameter that controls the overall brightness of an object displayed on the PS 390 screen.

Field-of-view angle. *Also called* Viewing angle. The angle at the apex of the viewing pyramid used to define a perspective viewing area.

Flat shading. A shading process that produces an object with area-filled colored polygons. Flat shading uses color, one light source, and depth cueing to shade the polygons in a rendered image. Flat shading produces a faceted surface. *See also* Rendering operation; Shading.

Frame buffer. *Also called* Image buffer. The memory allocated to hold the image that is to be displayed. The frame buffer temporarily stores image data drawn by pixel processors so that the screen may be refreshed from the buffer and not directly from the pipeline. It is also used in displaying host-generated pixel images.

Frustum. A section of a perspective viewing pyramid that is obtained by slicing through the pyramid parallel to the base. The frustum encloses a portion of the world coordinate system. Any objects within in the frustum will be displayed in perspective projection on the screen.

Function. The processing component of a function network, it performs one or more operations by accepting input, processing that input, and producing output. In the PS 390 there are two types of functions: intrinsic functions and user-written functions. *See also* Initial function instance; Intrinsic function; User-written function.

Function instance. A specific case of an intrinsic function or user-written function named by a user. A function instance must be created before the function can be used in a function network. A function instance includes all information necessary to identify the function type, its input source(s), and its output destination(s). *See* Initial function instance.

Function network. A collection of function instances. PS 390 commands are used to uniquely name instances of functions and connect outputs to inputs to form a function network. Functions are data driven. That is, they are active when appropriate data is received on function inputs and otherwise dormant.

General Purpose Interface Option card. *Also called* GPIO card. A card through which the PS 390 accepts data from a host. This interface allows more closely-coupled host control of the PS 390 in directing the dynamics of the displayed image.

Gouraud shading. *See* Smooth shading.

Graphics Control Program. The collection of graphics firmware that executes whenever the PS 390 is being used in the normal mode of operation. The Graphics Control Program is made up of data structuring definitions, the scheduler, and the functions. The Graphics Control Program executes following the confidence or self tests, initializes all data structures and communication handlers, and awaits input from host software or the keyboard. By loading and bootstrapping the graphics firmware, the Graphics Control Program creates the PS 390 runtime environment. *See* also Runtime environment.

Graphics firmware. Microcode and system software that, in conjunction with the hardware, manipulates the data structures traversed by the Display Processor. *See also* Host-resident software.

Graphics Support Routines. *Also called* GSRs. A set of host-resident software routines that are the standard vehicle for communication to the PS 390 from the host. These are a collection of FORTRAN, Pascal or UNIX/C routines that preparse and package data on the host computer. The GSRs provide a set of routines that perform all formatting and routing duties for the applications, which include attaching to the graphics device; creating and modifying display structures; creating, connecting, and modifying function networks; and receiving data from the graphics device. There is a GSR that corresponds to almost every PS 390 command. *See also* Application routine; Utility routine.

Hidden-line removal. A PS 390 rendering operation that generates a view in which only the unobstructed portions of an object are displayed. Hidden-line removal is displayed in a static viewport. *See also* Backface removal.

Hierarchical data structure. *See* Display structure.

Hierarchy. A ranked organization of components. The organizing principle will vary depending on the relationship between components which the hierarchy is designed to show.

Highlight. *See* Specular.

Hither plane. *See* Clipping planes.

Host-resident software. A software package distributed by Evans & Sutherland on magtape which is loaded onto the host system and which includes the Graphics Support Routines and other programs. This package should not be confused with the graphics firmware or with host application programs. *See also* Graphics firmware.

Hue. *See* Color.

Identity matrix. A matrix that is composed of ones and zeros, with the ones running in a diagonal (top left to bottom right). Multiplying a matrix by an identity matrix leaves the matrix unaltered.

Illumination. An attribute that is used to specify light sources applied to a shaded object. Illumination nodes may be placed anywhere in the display structure, allowing lights to be stationary or to rotate with the object or both.

Initial display structure. Display structure built and loaded into memory by the CONFIG.DAT file. This structure sets up the framework that allows the user to build display structures and contains everything needed to generate data displayed at bootup.

Initial function instance. An intrinsic system or user function that is, upon system initialization, automatically instanced for use. Such a function may be connected by the system into an initial function network or may be connected by the user to a user function network. *See also* Function; Function instance; Intrinsic function.

Inner contour. A polygon that represents a cavity, hole, or protrusion site in a polygonal object. Vertices of inner contours must be associated with a corresponding outer contour by running them in the opposite sense (e.g., counterclockwise versus clockwise). *See also* Vertex ordering rule; Outer contour.

Instance node. *Also called* Set node. An element of a display structure which groups other elements such as primitives, transformations, and attributes into a single-named entity.

Intensity. *See* Color; Depth cueing

Interactive device. *Also called* Peripheral. Provides programmable capabilities that allow an operator to interact with graphical data. These devices include, but are not restricted to, control dials, data tablet, keyboard, mouse, and function buttons.

Interactive mode. *See* Local mode.

Interaction node. *Also called* Interaction point. A point in the structure of a model where a function network from an interactive device can be connected to change the model dynamically.

Intrinsic function. One of the set of namable functions that are provided with the PS 390 for constructing function networks. These functions have the "F:" prefix and must be instanced (i.e., given a unique system- or user-defined name) before they can be used in a function network. There are two types of intrinsic functions. **Intrinsic system functions** (*Usually called* System function; *Also called* Internal function) are functions that should not be instanced by a user. **Intrinsic user functions**(*Usually called* Intrinsic function; *Also called* Standard function) may be instanced by a user to create a function network. *See also* Function; Initial function instance; User-written function.

Joint Control Processor. *Also called* JCP. It is the central processor for the PS 390. It provides the interfaces to devices external to the system and manages all internal system communications. The JCP controls the creation and update of display structures in Mass Memory as well as the traversal of those display structures by the Display Processor. The JCP also conducts self-test operations on all major PS 390 components, and loads and executes the PS 390 diagnostic programs.

Label. Character strings may be combined into a block, or node, of several strings, each of which is called a label. Labels are created as data nodes in a display structure by using the LABELS command. *See also* Character string.

Left-hand rule. A mnemonic for the direction of rotation around an axis in the world coordinate system of the PS 390. Point the thumb of your left hand in the positive direction of any axis, and your fingers will curl in the direction of positive rotation. *See also* Coordinate system.

Level-of-detail. An attribute that allows data to be conditionally referenced in a predetermined sequence.

Light source. *See* Illumination.

Line of sight. The orientation of the viewer in relation to the object being viewed. The PS 390 uses the line of sight to perform a matrix operation which transforms the coordinates of an object to produce the correct view on the screen. All points in the world coordinate system are translated and rotated to place the "from" point at the world coordinate system origin and the "at" point on the positive Z-axis.

Local mode. *Also called* Interactive mode. One of three communication modes available on a PS 390 keyboard. In local mode, function keys and any other programmed keys provide local input from selected devices for user-constructed PS 390 function networks. There is no cursor or screen prompt in local mode, and the keyboard does not send any information to the host. *See also* Command mode; Terminal Emulator mode.

Logical device coordinates. Ranges of X and Y values that define the dimensions and position in virtual address space of the area that contains the picture, which can be larger or smaller than the screen space.

Mass Memory. PS 390 memory in which display structures and other data are stored and managed by the Joint Control Processor. This data is accessed by the Display Processor through a dedicated port. *See also* Working storage.

Matrix. *See* Transformation matrix.

Memory alert. Whenever available memory drops below an acceptable level, a system function alerts the user by causing a message to be displayed. The message indicates the amount of remaining memory.

Message. *Also called* Token. The unit of information that is passed between two functions or a function and a data structure.

Model. As a verb, model or modeling is the process of defining graphical primitives and using transformations to shape model parts from primitives and to move parts into position relative to other parts that are grouped within the display structure. As a noun, a model is a visual representation of any real or imagined object. *See also* Display structure; Transformation.

Modeling transformations. Transformations that move primitives to a new location in the coordinate system or reform primitives to create new shapes. There are three modeling transformations: rotation (moving an object around an axis), scaling (altering the dimensions of an object), and translation (moving an object in coordinate space).

Multiplexing. The process of transmitting several messages simultaneously over the same transmission medium.

Mux box. *Abbreviation for* Peripheral Multiplexer. Located in the pedestal that supports the screen. All lines from the interactive devices are connected to the PS 390 control unit through the mux box. The mux box also supplies power to the screen and to the power supplies which drive the interactive devices.

Named entity. Data structure (data block) representing function instances, variables, character fonts and display structures. Data structure which can be (but not necessarily is) named and referenced.

Naming. Give a unique name to a single node or to a group of nodes in a display structure. Once a name has been given, all data specified by the commands are referenced by the assigned name. The naming convention of the PS 390 treats memory as a collection of objects, each created with a name and accessed (addressed) by that name using the system commands. *See also* Explicit naming; Instance node.

Node. *See* Data node; Instance node; Operation node.

Noncommutativity. A property of matrix algebra where the order of transformations must be preserved.

Normal. A surface normal is a vector that is perpendicular to a point on a surface. A vertex normal is a vector that is the average of the normals of the surfaces that are common to the vertex. Normals are used with shaded renderings and given with each direction vertex of the polygon. The shaded rendering operation interpolates between these normals when rendering the polygon to generate a smooth-shaded image. If any vertex of a polygon has a normal then all vertices for the polygon must.

Null object. Created when a name is referenced that has not previously been defined.

Object. Objects may be a single primitive created as lines, polygons, and characters, or a named grouping of primitives, attributes, and the mathematical operations which are applied to the data and attributes.

Object-space rotation. An object is said to rotate in object space when it rotates around its own set of axes, as opposed to the axes of the world coordinate system. *See also* World-space rotation.

Operation node. Element of a display structure that represent transformations and attributes. Operation nodes can be used as points of interaction with a model; they can receive new values from interactive devices such as dials or the data tablet. Operation nodes modify the state of the Arithmetic Control Processor. *See also* Attribute; Modeling; Transformation.

Orthographic projection. *Also called* Parallel projection. The two-dimensional projection of a three-dimensional object in which lines that are parallel in the object always appear parallel, without perspective. *See also* Perspective projection.

Outer contour. A polygon that represents the face of an object. Vertices of outer contours must be associated with a corresponding inner contour by running them in the opposite sense (e.g., counterclockwise versus clockwise). *See also* Inner contour; Vertex ordering rule.

Packet. Includes the data sent to the PS 390 and all necessary information to process it. Packets are built from bytes by the PS 390 system functions that interface between the system and the hardware. A QPacket is a block of character data (bytes) that can be sent from one PS 390 function to another.

Parallel projection. *See* Orthographic projection.

Perspective projection. A viewing projection that allows spatial relations (distance and position) of three-dimensional objects to be represented as they might appear to the eye. Parallel lines in the object appear to converge with respect to relative distance or depth from the eye position. Objects become smaller as they recede in the distance. *See also* Orthographic projection.

Phong shading. *See* Smooth shading.

Physical I/O operation. A method of data communication, understood by the General Purpose Interface Option microcode, which permits the host to directly access the internal contents (data, not structure, portions) of any node (or other PS 390 data structure) and modify, read, and/or write those Mass Memory locations.

Pick identifier. *Also called* Pick ID. A user-assigned identifier (tag) returned in the pick list.

Pick list. The information about an object which is returned when a pick occurs.

Picking. Selecting a displayed object or a portion of an object with a pointing device, such as a puck. When some part of the displayed object is picked, the PS 390 returns a pick list.

Pipeline Subsystem. *Also called* PLS. A subsystem in the Display Processor. The PLS accepts transformed data from the Arithmetic Control Processor and completes the transformations of an object to be displayed from world coordinates to screen coordinates. The PLS also performs two clipping tests on the data and tells the Arithmetic Control Processor what needs to be clipped. The PLS performs perspective division, intensity computations, and block normalizations, then outputs the fully transformed data to the raster backend bitslice card.

Pixel. A picture element. It is the smallest element which can be displayed on a raster display.

Pointer. A special kind of address (location in Mass Memory) that has a value and points to another address.

Polygon. A closed planer figure defined by the coordinates of its vertices. The edges of the polygon are defined by lines that connect those vertices. In the PS 390, a polygon must have at least three vertices and no more than 250, all of which must lie in the same plane. *See also* Solid; Sphere; Surface.

Polygon list. A polygon list contains the coordinates of the endpoints of the lines that make up the polygons. It specifies an object that is composed of surfaces joined together.

Primitive. *Also called* Graphic primitive. The simplest object in a display structure. It is specified as a vector list of points and lines or as a polygon list. *See also* Data node.

Priming. To initialize the constant input of a function. *See also* Constant input.

Q. A prefix added to words to indicate that the material defined by that word is stored on a queue, as in QData or QPacket.

Queue. An input queue holds items to be processed in sequence of arrival by the function. *See also* Active Input; Constant Input. There are also output queues for sending messages created by the functions, and private queues for storing information.

Radius. One of the six components of an attribute table, radius is used to define a sphere.

Raster backend bitslice card (RBE/BS). A subsystem in the Display Processor. Among the RBE/BS components are the master bit-slice processor, the delta depth cue calculator, eight of the pixel processors, and one-half of the frame buffer. The RBE/BS receives transformed data from the graphics pipeline, calculates line slopes and endpoints, converts this data into the format required by the pixel processors, and draws this image data into its frame buffer.

Raster backend video card (RBE/VC). A subsystem in the Display Processor. Among the components of the RBE/VC are eight of the pixel processors and one-half of the frame buffer. The RBE/VC takes pixel data from the frame buffer and uses that data to generate video values which it converts from digital to analog signals that are displayed on the screen.

Raster. A technique used for producing an image on a CRT screen. Raster images are generated with an intensity controlled, scanline-by-scanline sweep across the screen from top to bottom, in contrast to calligraphic images that trace only the displayed lines, dots, or characters.

Rate attribute setting. *See* Blinking.

Real time. Term which describes the response of the graphics system, in which there is no perceptible delay between the action of the user and the displayed result, nor in the refresh rate. In real time the picture appears to change immediately and smoothly without jerkiness.

Refresh rate. *Also called* Frame rate. The rate at which refresh frames are displayed; i.e. the number of times per second a picture is drawn on the display screen. The standard refresh rate for the PS 390 is 60 refresh frames each second. *See also* Update rate.

Rendering. *Also called* Rendered image. As used in the PS 390 system, a rendering is the new polygon list that is created from the rendering operations performed on the original polygon list.

Rendering operation. Definable option in the visualization of a model. As used in the PS 390 system, rendering operations can be performed only on a polygon list, and can be displayed in a dynamic viewport or a static viewport, depending on the rendering operation. Rendering operations that are displayed in a dynamic viewport are backface removal, cross-sectioning and sectioning. Rendering operations that are displayed in a static viewport are hidden line removal and shading. *See also* Backface removal; Cross sectioning; Hidden line removal; Sectioning; Shading.

RGB. *Abbreviation for* red, green and blue, the primary colors of light. *See also* Color.

Right-hand rule. *See* Vertex ordering rule.

Rotation matrix. A 3x3 matrix used to perform a rotation on an object. The PS 390 uses the sine and cosine of the angle of rotation to create the matrix, then applies the matrix to the coordinates of the points which define the object.

Routing. The transfer of data from the host to the Command Interpreter to the Terminal Emulator or other destinations. Routing bytes are characters used to select the appropriate channel for data in the PS 390.

Run-length encoding. A set of consecutive pixels of the same color specified in a single command containing the number of consecutive pixels and the color value of the pixels. Run-length encoding of raster data from the host allows more efficient picture transmission than pixel-by-pixel encoding.

Runtime code. Contains all definitions for system level commands and functions, as well as the definitions for user-accessible commands and functions. These definitions are loaded into the Joint Control Processor local memory and all other processors.

Runtime environment. Everything (including the PS 390 functions linked together to form the system function network, all user interfaces, and initialized hardware) to which the user has access after the system is booted.

Saturation. *See* Color.

Scaling matrix. The PS 390 creates a 3x3 scaling matrix which multiplies the coordinates of the points which define the object by the scale factor. This determines the new coordinates of the scaled object.

Scheduler. The driving force behind the Graphics Control Program. It executes activated functions (instanced functions which have received all inputs needed for execution).

Screen coordinate system. The coordinate system in which an object is viewed on the screen. *See also* Coordinate system; Viewport.

Screen-oriented characters. Screen-oriented characters are not affected by ROTATE and SCALE nodes that are applied to the object of which they are a part. Screen-oriented characters maintain their size and their front-facing orientation when other data is transformed.

Sectioning. *Also called* Sectioned rendering. A rendering operation which cuts away parts of polygons that extend beyond an arbitrarily positioned plane called the sectioning plane. This plane passes through the object to divide the object into two pieces. When sectioning is performed, the affected polygons are reconstructed so that they do not extend beyond the sectioning plane; one piece is removed while the other remains displayed. Sectioning is displayed in a dynamic viewport. *See also* Cross sectioning.

Set-operate-data structures. Data structures in Mass Memory containing a combination of set (instance), operation, and data nodes.

Shading. The process of drawing the surface of a rendered image displayed in a static viewport. The characteristics of that surface are defined with the ATTRIBUTES command. Wash shading fills the interior of the polygons with the color given in the attribute node corresponding to that polygon, without considering the light source. Flat shading uniformly fills the interior of the polygons with the color given in the attribute node corresponding to that polygon, and it takes into consideration the light source. Smooth shading fills the polygons in a non-uniform manner to give the appearance of curvature to the object surface. There are two types of smooth shading: Gouraud shading and Phong shading. The shading styles in increasing order of quality are: Wash, Flat, Gouraud and Phong. Shading is displayed in the static viewport. *See also* Flat shading; Smooth shading; Wash shading.

Smooth shading. A rendering operation applied to images displayed in a static viewport. The color of a polygon is varied across its surface, affected by the normals at the vertices of the polygon, the direction and color of various active light sources, the attributes of the polygon (both color and highlights), and depth cueing. Gouraud shading and Phong shading are the two styles of smooth shading. Gouraud shading is faster than Phong shading, but it does not produce the shading quality of Phong shading. Objects that simulate a curved surface can be produced with smooth shading.

Soft edges. Invisible in hidden-line images except when they make up part of the profile of an object or a silhouette. They can, therefore, be used to approximate curved surfaces. *See also* Hidden-line removal.

Software, software packages. *See* Graphics firmware; Host-resident software.

Solid. A polygonal object that encloses a volume of space. It is composed of both front and back facing polygons. In a solid, every edge of every polygon must coincide with an edge of a neighboring polygon.

Specular. Polygon attribute used in shading polygons to adjust the concentration of specular highlights/reflected light. Increasing specular makes the surface shine more. *See also* Diffuse.

Sphere. A rendered image primarily used in molecular modeling. Represented as a vector list instead of an explicit spherical data type. *See also* Polygon.

State-of-the-machine. A description that includes the current transformation matrix (CTM), the current level of detail, current conditional bit values, and status of pick identifiers. Instance nodes save and restore the state-of-the-machine between descendent branches of a display structure.

Static viewport. The viewport in which hidden-line and shaded rendering operations are displayed. Rendering styles displayed in the static viewport include flat shading, Gouraud shading, Phong shading, wash shading, and hidden-line removal. In a static viewport interaction with the displayed image is not possible. *See also* Dynamic viewport.

String. A sequence of characters and spaces enclosed in single quotation marks. Strings can be displayed as text or can be used as inputs to function instances. *See also* Character string.

Suffix. Characters added to the end of a name. Name suffixing distinguishes system-level names and function instances from user-defined names and function instances, and is automatically performed by the Command Interpreter, except in configure mode. *See also* Naming.

Surface. A bounded plane which defines a polygonal object, in which the backfacing polygons have been removed, that does not necessarily enclose a volume of space. Surfaces can have edges that belong to just one polygon. *See also* Solid.

System or system-level function. *See* Intrinsic function.

System function network. The network of the PS 390 system functions through which data flows internally. Part of the PS 390 Runtime environment.

Terminal Emulator. *Also called* TE. A feature available over standard interface lines that allows the PS 390 to be used as a host terminal.

Terminal Emulator mode. *Also called* TE mode. One of three types of communication modes available on a PS 390 keyboard. In Terminal Emulator mode the PS 390 terminal functions as a standard host terminal so that the user can log on to the host, access and edit host resident files, and use the available host system utility commands. *See also* Command mode; Local mode.

Terminal Emulator network. Function network responsible for directing the flow of data that will appear on the screen as text or graphics display, for determining certain attributes of the Terminal Emulator feature, for routing input from the PS 390 keyboard to the host or to local PS 390 system functions, and for receiving data from the host line.

Text string. *See* Character string.

Token. *See* Message.

Transformation. A mathematical operation ultimately applied to a primitive to change its geometry by moving some or all of its points to a new location in the world coordinate system. The basic transformations are: modeling (rotating, scaling, translating) and viewing (line of sight, viewing area).

Transformation matrix. All graphical transformations are applied to objects through transformation matrices. The display structure indicates the types of transformations and the order in which they are to be performed. The PS 390 performs the concatenation of matrices that this involves. This means that each matrix is premultiplied to a matrix called the current transformation matrix. The current transformation matrix (CTM) contains the accumulation of all transformations that are to be applied to graphical data and preserves the order in which they are to be applied.

Transformed data. Image data that has been fully processed by the PS 390's graphics pipeline. This type of data has had all of the pipeline's transformation operations applied to it, including modeling, viewing, and projection transformations. Transformed data may be either matrix (from which operation nodes may be created) or vector list (from which data nodes can be created) representations of the transformation operations in a display structure. Transformed data can be retrieved from a given data node and then established as a separate data or operation node in the display structure. Transformed data can also be transmitted to the host.

Traverse. A process in which the Arithmetic Control Processor steps through the display structure in Mass Memory to retrieve data and operation specifications necessary to generate an image.

Trigger. To start the operation of a function when all necessary inputs have data.

Update. The process of changing the contents of a data or operation node.

Update rate. The frequency at which data structures containing new values are traversed by the Arithmetic Control Processor to produce a new or changed image on the display system. *See also* Refresh rate.

User-written function. A function written by a PS 390 user for a specific application. That application may perform operations not provided by intrinsic functions or perform operations that would require a large network of intrinsic functions to accomplish; i.e. to collapse a large function network into a single user-written function. User-written functions are documented in the Advanced Programming Volume of the PS 390 Document Set.

Utility routine. One of two types of Graphics Support Routines. Utility routines are specific to the operation of the Graphics Support Routines. These routines are used to attach and detach the PS 390, set the string delimiting character, select multiplexing channels, and send and receive messages. *See also* Application routine.

Vector list. A set of coordinate pairs (X,Y) or triples (X,Y,Z) which specify the points within the world coordinate system at which lines start and end.

Vector-normalized vectors. When the components (X, Y or X, Y, Z) of a single coordinate (vector) location share a common exponent, they are said to be vector-normalized. Vector-normalized vectors are not displayed on the PS 390. All ASCII and Graphics Support Routine vector list commands which do not specify block-normalized vectors will create 32-bit block-normalized vectors internally in the PS 390. *See also* Block-normalized vectors.

Vertex ordering rule. In a polygon command creating a solid, vertices should be listed so that if one starts at any vertex and moves to the next vertex, one travels around the edges of the polygon in a clockwise direction. The right-hand rule states that if one points the thumb of the right hand towards the center of a polygonally defined object and curls the fingers towards the wrist, the direction in which the fingers move indicates the order in which the vertices of that polygon should be listed.

Viewing area. *Also called* Window. The portion of the world coordinate system in which objects can be viewed. The viewing area is mapped to the viewport. *See also* Coordinate system; Screen coordinate system; Viewport.

Viewing pyramid. A perspective viewing area. The actual viewing area is shaped like a frustum. The pyramid is completed by extending the converging sides of the pyramid until they meet. This point, the apex of the pyramid, is the eye point of the viewer and is coincident with the world coordinate system origin. *See also* Frustum.

Viewing transformations. Matrix operations which specify whether displayed objects are displayed as perspective or orthographic projections. Viewing transformations also specify a point to *look from* and a direction to *look at* in the world coordinate system.

Viewport. A portion of the screen coordinate system with horizontal (X) and vertical (Y) boundaries and an optional intensity range in which images are displayed. The viewport specification is a ratio and proportion calculation, unlike viewing transformations which are matrix operations. To obtain an accurate view of an object, the viewport in which it is displayed must have the same aspect ratio as the front boundary of the viewing area that enclosed the object. *See also* Dynamic viewport; Screen coordinate system; Static viewport.

Wash shading. A shading operation that produces an object with area-filled colored polygons. Wash shading ignores normals, light sources, all lighting parameters, and all depth cueing parameters. *See also* Shading.

Window. *See* Viewing area.

Wireframe model. Objects defined as points and the lines that connect them.

Working storage. Consists of a large contiguous block of PS 390 Mass Memory needed to create renderings. Working storage is explicitly reserved with the RESERVE_WORKING_STORAGE command. *See also* Mass Memory.

World coordinate system. *See* Coordinate system.

World-space rotation. An object is said to rotate in world space when it rotates around any of the world coordinate system axes, as opposed to one of the axes of the object itself. *See also* Object-space rotation.

World-oriented characters. Characters that are transformed along with an object of which they are a part. *See also* Screen-oriented characters.

Yon plane. *See* Clipping planes.

Z-clipping. *See* Depth clipping.