

PS 300 DOCUMENT SET

VOLUME 2a

GRAPHICS PROGRAMMING

The contents of this volume are not to be reproduced or copied in whole or in part without the prior written permission of Evans & Sutherland.

Many concepts in this volume are proprietary to Evans & Sutherland, and are protected as trade secrets or covered by U.S. and foreign patents or patents pending.

Evans & Sutherland assumes no responsibility for errors or inaccuracies in this document. It contains the most complete and accurate information available at the time of publication, and is subject to change without notice.

PS1, PS2, MPS, and PS 300 are trademarks of the Evans & Sutherland Computer Corporation.

Copyright © 1984
EVANS & SUTHERLAND COMPUTER CORPORATION
P.O. Box 8700, 580 Arapeen Drive
Salt Lake City, Utah 84121



VOLUME 2

GRAPHICS PROGRAMMING

Volumes 2A and 2B consist primarily of a series of tutorials which teach PS 300 programming. Both volumes are designed to instruct programmers of various levels of expertise. Those with little computer graphics experience will want to read carefully through each module and do each exercise.

Each tutorial module begins with an introduction and ends with a summary. Those with some computer graphics experience may find it sufficient to read these and supplement them with the reference material in Volume 3. Though sophisticated users may want to rely primarily on this reference material, they are encouraged to read this volume as well to become familiar with the approach to graphics programming taken in the document set.

Every PS 300 programmer should read the following sections, which begin Volume 2A:

HANDS-ON EXPERIENCE

This brief section steps you through a first encounter with the PS 300. Even with no prior graphics experience, you can quickly learn to take advantage of the PS 300's capabilities.

GRAPHICS PRINCIPLES -- HIGH PERFORMANCE PS 300 DISTRIBUTED GRAPHICS

This is the foundation of Volume 2. It presents the concepts of interactive graphics--how to construct models in a coordinate system--and illustrates how PS 300 programming puts these concepts into effect.

TUTORIAL DEMONSTRATION PACKAGE

The tutorial demonstration package consists of programs which illustrate many of the graphics principles detailed in the tutorial modules. This set of software is distributed on magnetic tape. In addition to these programs, the tape contains a group of primitives which are required for many of the exercises in the tutorial modules. Before reading the tutorials, be sure to load the demonstration package.

THE TUTORIAL MODULES

The tutorial modules in Volumes 2A and 2B contain an in-depth discussion of PS 300 programming. They provide you with experience in creating and manipulating an object on the screen using PS 300 commands. After reading these modules initially, you may want to use Volumes 2A and 2B in conjunction with Volume 3 as a reference source to program your own applications.

Specifically, the tutorial modules in Volume 2A are prerequisite to reading Volume 2B. They should be read in the order in which they are arranged. Volume 2B contains more specialized material based on the fundamental principles taught in 2A. The modules in 2B can be read in any order desired.

Each tutorial consists of:

- A list of the demonstratable skills you should be able to perform on the PS 300 after reading the module.
- Any prerequisite reading you should do before reading that module.
- An introduction which is a functional overview of the contents.
- A detailed explanation of the programming concepts outlined in each objective, including examples and practice on the PS 300. (A horizontal line separates each practice from a feedback section.)
- A detailed summary of the module, including essential details about fundamental concepts.

MISCELLANEOUS

Sample programs have been provided at the end of Volume 2B. Also included is a glossary of terms, and an appendix of general user information.

NOTE

Volume 2 contains no user information that is specific to the PS 320. Information specific to the PS 320 is contained in Appendix A of Volume 5.

HANDS-ON EXPERIENCE

INTRODUCTION TO PS 300 GRAPHICS

CONTENTS

STRATEGY	1
For Systems With a Non-IBM Host	1
For Systems With an IBM Host	2
For Systems With a CSM Display	2
DISPLAYING A SQUARE	2
The Display List	4
Coordinate Values	4
Blanking the Screen	5
DISPLAYING A DIAMOND	5
DISPLAYING A STAR	6
TWO MORE VERSIONS OF STAR	7
UPDATING VALUES, CONNECTING AN INPUT DEVICE	9
ANOTHER WAY TO CLEAR THE SCREEN	10
CONNECTING A DIAL TO SPINSTAR	10
CONCLUSION	12



In this module you will begin programming the PS 300 to display a few simple objects. Unlike the demonstration programs you have already worked with, where a pre-programmed object was displayed and you were able to manipulate it, here you will actually create the object before you interact with it. Everything you will be doing in this module will be done locally on the PS 300, without any help from your host computer.

STRATEGY

First you will build and display a square on the PS 300 screen. Next, you will make a rotated version of that same square to display as a diamond shape. Then, you will link these two shapes together for display as one object, an eight-pointed star. Last, you will make two slightly modified versions of the star and manipulate them.

First, boot the PS 300. This is described in detail in the *User Operation and Communication Guide* in Volume 1. Briefly, here is what you need to do.

- Put the PS 300 Graphics Firmware Diskette in the disk drive.
- Boot the system by turning on the power.

For Systems With a Non-IBM Host

Once the system is booted, hold down the CONTROL key and press the LINE LOCAL key. Then press the RETURN key to enter a carriage return (<CR>). You will see this prompt

@@

which indicates the PS 300 is now in *command mode*. It will accept any instructions you give it and execute them locally. (Command mode and other modes of operation are described in the *User Operation and Communication Guide* in Volume 1.)

For Systems With an IBM Host

Once the system is booted, hold down the ALT key and press the LOCAL key. This prompt will appear:

@@

This indicates the PS 300 is in *command mode* and will accept any command you give it. When your host is an IBM, remember to enter a carriage return <CR> on the PS 300 keyboard (instead of the ENTER key) when you are working in command mode. The ENTER key does not work in command mode. (Command mode and other modes of operation are described in the *User Operation and Communication Guide* in Volume 1.)

For Systems With a CSM Display

Check with your system manager to be sure the CSM's line-drawing speed is set correctly. If it is not, displayed objects will be distorted. To set the line-drawing speed enter

```
SEND TRUE TO <1>CSM1;
```

DISPLAYING A SQUARE

Before the PS 300 can display anything, it needs the coordinate points of the object you want to build--the square. Any wire-frame object you define must be specified as a collection of *vectors*, coordinate points and lines. The VECTOR LIST command does this. Enter

```
@@Square := VECTOR_LIST .5,.5 .5,-.5 -.5,-.5 -.5,.5 .5,.5; <CR>
```

Enter this command exactly as you see it here. Pay special attention to all punctuation and the carriage return, but do not worry about capitalization (the PS 300 accepts either uppercase or lowercase letters). The "@@" prompt is shown here only because it appears on the screen when you enter commands. It is not something you have to enter.

If the command is accepted, another @@ prompt will appear on the next line, so this is what you should see on your screen.

```
@@Square := VECTOR_LIST .5,.5 .5,-.5 -.5,-.5 -.5,.5 .5,.5; <CR>
@@
```

If you get an error message instead, be sure you ended the line with a semicolon. The @@ prompt will not appear after an error message until you enter another carriage return.

After an error message, enter the command again, exactly as shown above. Try this two or three times. If the command still is not accepted, the problem lies elsewhere. Talk to your system manager.

After the PS 300 accepts this command, it knows about an object called Square that it will draw by going to the first point in the vector list (.5, .5) and then drawing to the next four points in the sequence listed (you need to end up back at .5, .5 to close the Square). The PS 300 will not display Square until you tell it to using the DISPLAY command. Enter

```
@@DISPLAY Square; <CR>
```

Square will appear centered on the screen. That is because Square is centered on the world coordinate system's origin, which currently corresponds to the center of the screen. By default, the part of the coordinate system viewed is from -1 to +1 in X and Y.

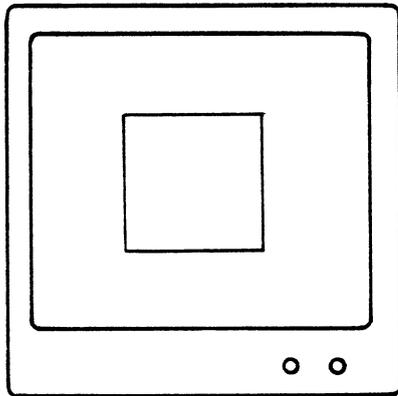
The Display List

As you have just seen, an object can be defined in the PS 300 and not be visible on the screen. When you use the DISPLAY command to display an object, the object's name is placed on a *display list*. The PS 300 continually checks this list to see if any names have been added or removed and then displays or "undisplays" the corresponding objects.

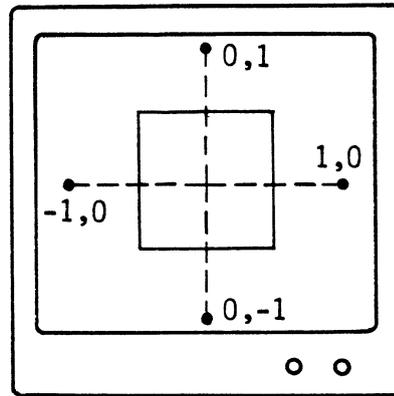
Coordinate Values

Right now, the screen shows a view of only part of the coordinate system, from plus 1 to minus 1 on both the X and Y axes. Anything to be drawn outside those coordinates will not show up on the screen. To see an object, you have to choose coordinates for it that are within these bounds. So, the coordinates for Square's corners are one-half unit in X and one-half unit in Y, and they appear about halfway from the center to the edge of the screen (Figure 1).

The Z axis, which accounts for the third dimension of "depth", will not be used in this module. Everything will be two dimensional and take place in the plane defined by the X and Y axes, with Z equal to zero.



This square as it appears on the screen.



If you could see the coordinate system axes, they would look like this.

IAS0369

Figure 1. The Part of the Coordinate System that Appears on the Screen

Blanking the Screen

Two very useful keys, TERM and GRAPH, are located to the left of the typewriter section of the keyboard.

- Press the TERM key when you want to clear the screen of text. Labels or titles that are part of the displayed object are unaffected. This key toggles, so you can press it again to re-display, or "unblank," the text.
- Press the GRAPH key to blank any graphics being displayed on the screen. This will allow you an uncluttered view of the text. Press GRAPH again to re-display the graphics.

DISPLAYING A DIAMOND

After displaying a square, the next thing to do is to superimpose a diamond on it to make a star shape. Create the diamond as a rotated version of Square. Enter

```
@@Diamond := ROTATE IN Z 45 APPLIED TO Square; <CR>
```

which means essentially "create a new object by applying a 45-degree rotation to the object Square."

To get a star figure to display on the screen, enter

```
@@DISPLAY Diamond; <CR>
```

Diamond is displayed superimposed on Square, resulting in a star-shaped object.

At this point, you have done what you set out to do, which was to display a star shape on the screen. But if you want to do anything to the star now on the screen, you must issue two commands, one for each of the two objects that make it up. There is no single object named Star that you can manipulate.

You can create such an object using an `INSTANCE` command. This command defines some new single object as a collection of other objects. You can define `Star` to be an *instance* of the two objects you already created. Enter

```
@@Star := INSTANCE OF Square, Diamond; <CR>
```

Now any operation you apply to `Star` will apply simultaneously to its two components, `Square` and `Diamond`.

DISPLAYING STAR

Before you display `Star`, remove the `Square` and `Diamond` from the screen. Enter

```
@@REMOVE Square; <CR>
```

and then

```
@@REMOVE Diamond; <CR>
```

The screen should now have nothing on it but text. There is a difference between removing objects from the screen this way and toggling the `GRAPH` key. When you press the `GRAPH` key, every object on the display list is blanked out from the screen (or unblanked so it will show up), but the contents of the display list stay the same.

When you `REMOVE` something, it is removed from the display list and will not display no matter how many times you press the `GRAPH` key.

Now enter

```
@@DISPLAY Star;<CR>
```

`Star` will appear. It looks like the two objects you just removed, but now it is defined in the PS 300 as only one object.

TWO MORE VERSIONS OF STAR

With the SCALE command, you can *scale* an object on the screen to shrink it or enlarge it. For example, to make a new star one-fourth the size of Star, enter

```
@@Smallstar := SCALE BY .25 APPLIED TO Star; <CR>
```

and

```
@@DISPLAY Smallstar; <CR>
```

Smallstar will appear inside Star, centered on the screen origin.

You can use the TRANSLATE command to define an object that is a "moved" version of some other object. Enter

```
@@Movestar := TRANSLATE BY .75,0 APPLIED TO Smallstar; <CR>
```

Movestar is a new version of Smallstar moved three-fourths of a unit to the right. The two values, .75 and 0, indicate how to move the object in X and Y. When the Y value is 0, the object translates horizontally only. Enter

```
@@DISPLAY Movestar; <CR>
```

and the new object will appear on the screen.

Even though the two newer stars, Smallstar and Movestar, are based on Star, they are separate objects with names of their own. You can do anything you want to Movestar or Smallstar and not affect Star. If you rotate or scale Smallstar, nothing will happen to Star. It will still be displayed at the center of the screen until you remove it.

The reverse is not true. If you redefine Star in some way, that will affect Smallstar and Movestar because they are defined in terms of Star. Redefine Star as a triangle and watch what happens to Smallstar and Movestar. Enter

```
@@Star := VECTOR_LIST 0,.43 .5,-.43 -.5,-.43 0,.43; <CR>
```

These coordinates define an approximately equilateral triangle.

As soon as you enter this command, what happens? Not only Star, but everything defined in terms of Star, changes.

As a further illustration of how Smallstar and Movestar depend on Star, redefine Star once more, as the word "STAR". You need a couple of commands to accomplish this. You could do the same thing with one BEGIN_STRUCTURE... END_STRUCTURE. BEGIN_STRUCTURE... END_STRUCTURE is a convenient way to group related commands together.

```
@@Star := BEGIN_STRUCTURE <CR>  
@@CHARACTER SCALE .1; <CR>  
@@CHARACTER -.2, 0 'STAR'; <CR>  
@@END_STRUCTURE; <CR>
```

All three objects will change from triangles to the word "STAR". Smallstar is still a quarter-size version of Star. And Movestar appears to the right of both of them.

Briefly, here is what the two commands in the BEGIN_STRUCTURE... END_STRUCTURE did:

CHARACTERS. The CHAR instruction specifies the word you want to display and the location of the lower left corner of the first character in the word. In this case, the S of STAR will be placed one-fifth unit (.2) out on the negative X axis. The characters in single quotation marks comprise the character string to be displayed.

CHARACTER SCALE. Without scaling, each character would appear on the screen one unit in size. The first letter would cover the entire upper right quarter of the screen, and any letters following it would be out of view to the right. So this instruction scales the characters to one-tenth their normal size so they can all appear on the screen.

The intricacies of BEGIN_STRUCTURE...END_STRUCTURE and the two CHARACTER commands (CHARACTER and CHARACTER SCALE) are explained in detail in other modules.

You can redefine Star to be the eight-pointed figure it was before. Square and Diamond still exist in memory, so all you need to do is re-enter the command that defines Star as an instance of those two objects. For an exercise, do that now.

UPDATING VALUES, CONNECTING AN INPUT DEVICE

If you wanted to re-position Movestar, you could do so by redefining it with new translation values like this:

```
Movestar := TRANSLATE BY -.5,0 APPLIED TO Smallstar;
```

This would redefine Movestar at a new position to the left of the origin.

There is a way to re-position Movestar without redefining it, and that is to SEND a new value to it. To do that, enter

```
@@SEND V3D(.75,.75,0) to <1>MOVESTAR; <CR>
```

Remember that Movestar is a *translation* applied to another object, Smallstar. Whenever you update a translation, you must send it a three-dimensional value; "V3D" stands for a three-valued vector. To supply Movestar with the right kind of data, you had to deal with all three dimensions, even though you are not making use of Z here.

This SEND command immediately updates the translation values in Movestar (they were .75,0 with an assumed Z value of 0). Movestar shifts to the upper right corner of the screen.

None of the definitions for any objects changed--the values changed. This is what input devices and *function networks* do. Without changing the basic definitions of objects, they alter and update values; how big to scale the objects, how much to rotate or translate them, and so on.

To illustrate this, hook a simple function network from a control dial to an object to make it rotate.

First, create the object. The PS 300 already knows about Smallstar, the scaled-down version of Star. Define Spinstar to be a version of Smallstar that can rotate. Enter

```
@@Spinstar := ROTATE IN Z 0 APPLIED TO Smallstar; <CR>
```

You must put 0 as the initial rotation value. Later, values coming from a dial will update Spinstar and make it rotate.

ANOTHER WAY TO CLEAR THE SCREEN

You have defined Spinstar. The next step is to display it. But first, clear the screen of the other objects. When you did this before with Square and Diamond, you used REMOVE for each of them. It is much more convenient to use the INITIALIZE DISPLAY command. Enter

```
@@INITIALIZE DISPLAY; <CR>
```

This clears everything from the display list. Now display Spinstar by entering

```
@@DISPLAY Spinstar;
```

CONNECTING A DIAL TO SPINSTAR

Now build a simple function network to take values from the dial and turn them into values that can be used to update Spinstar. The PS 300 contains a "master" function called F:DZROTATE that does that. To use it, make a copy of it and assign it a name, "Spinner" for example. Enter

```
@@Spinner := F:DZROTATE; <CR>
```

It is convenient to think of individual functions as "black boxes" with values coming in and other values going out. The functions are usually drawn as shown in Figure 2, as squares with input and output lines:

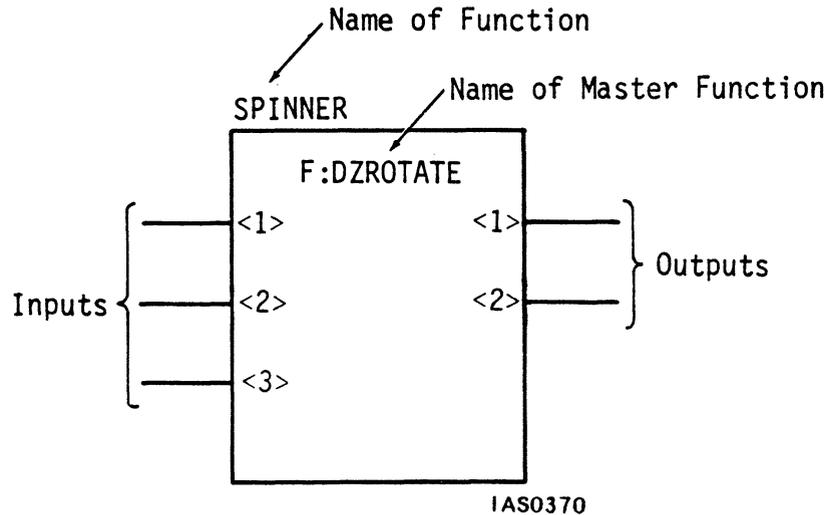


Figure 2. "Spinner" Function Diagram

Connect a dial to the first input of this function and the first output to Spinstar. Use the CONNECT command twice to do this.

```
@@CONNECT Dials<1>:<1>Spinner; <CR>
@@CONNECT Spinner <1>:<1>Spinstar; <CR>
```

These commands say to connect output <1> of the control dials (corresponding to the top left dial) to input <1> of the function Spinner. And connect output <1> of Spinner to input <1> of Spinstar. The numbers for the inputs of a function are to the left of the name, the output numbers are to the right.

You are not quite finished setting up your function network, because Spinner needs to be "primed." It needs two initial values for its second and third inputs. You have already used the SEND command to update Movestar. You can use this command again to send 0 and 200 to Spinner's second and third inputs, respectively.

```
@@SEND 0 TO <2>Spinner; <CR>
@@SEND 200 TO <3>Spinner; <CR>
```

The function network is now ready. If you turn dial 1, Spinstar will start turning. The values the dial generates update Spinstar so quickly that it appears to move in *real time*. When you turn the dial, Movestar responds instantaneously.

CONCLUSION

If any of what you have done is not completely clear to you, do not worry about it right now. The purpose of this section was to give you an opportunity to create and manipulate a few simple objects.

In the remaining modules, you will discover in more detail how you can use these commands to create *display trees* and more complex function networks for models. You will also learn how to save the commands in a host file so they are more convenient to use.

GRAPHICS PRINCIPLES

HIGH PERFORMANCE PS 300 DISTRIBUTED GRAPHICS



PREFACE

This guide introduces the concepts and terminology which you must understand to program the PS 300. It begins by explaining concepts which are common to most interactive graphics systems, but it soon becomes specific to the PS 300. The concepts introduced here are explained in much greater detail in the tutorial modules. In most cases, cross references are given to appropriate modules. New terms are introduced in *italics* and are explained in context. Italicized words also appear in the Glossary of Terms in Volume 2B.

Examples of the PS 300 Command Language and of some PS 300 Functions and Function Networks are given to show how specific computer graphics operations are performed by the PS 300. Little attempt is made to explain the syntax of commands or to explore all of the options of a particular command or function. Consult the *PS 300 Command Summary* and *PS 300 Function Summary* in Volume 3A for complete information on the commands and functions and their options.

Programmers with little or no experience of computer graphics systems should read this guide before embarking on the tutorial modules. Experienced programmers who do not plan to use the tutorials can read this guide as an introduction to the reference documentation in Volumes 3A and 3B.



GRAPHICS PRINCIPLES

CONTENTS

1. CREATING PRIMITIVE OBJECTS	1
COORDINATE SYSTEMS	1
Right-Hand Coordinate System	1
Left-Hand Coordinate System	2
The World Coordinate System	3
DATA BASE FOR AN OBJECT	4
Geometry	5
Coordinate Notation	6
Topology	7
Vector List	7
Polygon List	8
GRAPHICAL PRIMITIVES	8
Curve Primitives	9
Text Primitives	9
Same Geometry but Different Topologies	9
Same Topology but Different Geometries	10
SUMMARY	12
2. TRANSFORMING PRIMITIVES	13
CREATING NEW OBJECTS FROM PRIMITIVES	13
Applying Transformations	15

GRAPHICS PRINCIPLES

MODELING TRANSFORMATIONS	16
Rotation	16
Rotations Around an Axis	17
Translation	18
Translations in All Three Axes	19
Scaling	20
THE ORDERING OF TRANSFORMATIONS	22
Transformation Matrices	27
SUMMARY	30
3. CREATING COMPOUND OBJECTS	31
BUILDING WITH PRIMITIVES AND TRANSFORMATIONS	31
Creating a Star Primitive	31
Grouping Primitives and Transformations	35
SUMMARY	37
4. DESIGNING A MODEL FOR INTERACTION	39
DESIGNING A COMPLEX MODEL	39
Analyzing a Model as a Hierarchy	41
DISPLAY TREES	42
Display Tree for the Mechanical Arm	42
Display Tree Terminology	44
Nodes	44
Updating Nodes	44
Data Nodes	45
Operation Nodes	46
Instance Nodes	47
Grouping	47
Sphere of Influence	49
SUMMARY	52

GRAPHICS PRINCIPLES

5.	LOOKING AT OBJECTS	53
	VIEWING OPERATIONS	54
	Displaying an Object	54
	ESTABLISHING A LINE OF SIGHT	55
	INCLUDING PART OF THE WORLD COORDINATE SYSTEM	58
	Viewing Areas in the World Coordinate System	59
	Orthographic Views	59
	Perspective Views	62
	DISPLAYING AN IMAGE IN SOME AREA OF THE SCREEN	67
	Specifying a Viewport	67
	VIEWING TRANSFORMATIONS AND DISPLAY TREES	70
	SUMMARY	76
6.	USING ATTRIBUTES	77
	ATTRIBUTES	77
	APPEARANCE ATTRIBUTES	78
	Displaying Objects in Color	78
	Displaying All Vectors in the Same Color	79
	Color Blending Vectors	82
	Setting and Changing Intensity Levels	83
	Enabling and Disabling Depth-Clipping	84
	Displaying Images on Different Screens	87
	Choosing a Character Font for Text	88
	STRUCTURE ATTRIBUTES	90
	Conditional Referencing	91
	Level-Of-Detail	93
	Blinking or Alternating Displays	96
	PICKING ATTRIBUTES	98
	SUMMARY	102

GRAPHICS PRINCIPLES

7. INTERACTING WITH THE PICTURE	103
EVANS & SUTHERLAND AND INTERACTIVE GRAPHICS	103
PROGRAMMING THE INTERACTIVE DEVICES	105
Planning for Interaction	105
Updating a Node	107
Supplying the Correct Type of Data	108
PS 300 FUNCTIONS	108
Intrinsic Functions	110
Initial Function Instances	111
User-Written Functions	111
Creating Networks	111
Active and Constant Inputs	115
Data-Driven Networks	116
Why Function Networks?	116
Creating Function Networks	117
SUMMARY	118

GRAPHICS PRINCIPLES

ILLUSTRATIONS

Figure 1.	Right-Hand Coordinate System	2
Figure 2.	Left-Hand Coordinate System	3
Figure 3.	The World Coordinate System	4
Figure 4.	Coordinates of a Square	5
Figure 5.	Location of the Square in the World Coordinate System	6
Figure 6.	Primitives With the Same Geometry and Different Topologies	9
Figure 7.	Primitives With the Same Topology and Different Geometries	10
Figure 8.	Location of the Diamond	14
Figure 9.	The Structure of the Diamond	15
Figure 10.	Rotation in the World Coordinate System	16
Figure 11.	Orientation of the Rotated Arrow	17
Figure 12.	Rotation of an Object Not Centered at the Origin	18
Figure 13.	Location of the Translated Square	19
Figure 14.	Square Translated in X and Negative Y	20
Figure 15.	Scaling the Square	21
Figure 16.	Non-Uniform Scaling to Create a Rectangle	22
Figure 17.	A Two-Dimensional Arrow	23
Figure 18.	Rotated Arrow	23
Figure 19.	Arrow Rotated, Then Translated	24
Figure 20.	The Structure of Arrow_2	24
Figure 21.	Translated Arrow	25
Figure 22.	Arrow Translated, Then Rotated	26
Figure 23.	The Structure of Arrow_4	26
Figure 24.	An Identity Matrix	28
Figure 25.	Concatenating Matrices	29
Figure 26.	Location of the Star Primitive	32
Figure 27.	The Star Primitive Displayed on the Screen	32
Figure 28.	The Location of Trans_Star on the Screen	33
Figure 29.	The Structure of Trans_Star	34
Figure 30.	The Structures of Trans_Square and Trans_Diamond	34
Figure 31.	The Structure of Trans_Star1	36
Figure 32.	An Articulated Mechanical Arm	40
Figure 33.	Hierarchy of Parts for the Mechanical Arm	41
Figure 34.	Display Tree for the Mechanical Arm	43

GRAPHICS PRINCIPLES

Figure 35.	Inputs to a Vector List Node	45
Figure 36.	Inputs to a Rotation Node	46
Figure 37.	The Structure of Trans_Star1	47
Figure 38.	Display Tree for Trans_Star1	48
Figure 39.	Structure of the Upper Arm	49
Figure 40.	A Simple Display Tree	50
Figure 41.	Shape Represented by Display Tree in Figure 40	50
Figure 42.	The Location of the Square on the Screen	54
Figure 43.	A Cube With Labeled Faces	55
Figure 44.	Displaying the Cube	56
Figure 45.	"Looking Down" the Y Axis at the Cube	56
Figure 46.	Looking Down at the Cube: the View on the Screen	57
Figure 47.	How the LOOK Command Rearranges the Coordinate System	58
Figure 48.	An Orthographic Viewing Area	59
Figure 49.	"Visible" and "Non-Visible" Objects	60
Figure 50.	Clipping Parts of an Object	60
Figure 51.	Depth-Clipping of Objects	61
Figure 52.	Orthographic View of a Rotated Cube	61
Figure 53.	The Default Viewing Space	62
Figure 54.	Perspective View of a Rotated Cube	63
Figure 55.	A Viewing Area for Perspective Views	63
Figure 56.	The FIELD_OF_VIEW Viewing Pyramid	65
Figure 57.	The Viewing Pyramid Created by the EYE Command	66
Figure 58.	Displaying an Object With the Default Window	67
Figure 59.	Distorted Views of the Arrow	69
Figure 60.	A Group of Objects in the Coordinate System	70
Figure 61.	Display Tree for Shapes	70
Figure 62.	DISPLAYing Shapes	71
Figure 63.	Adding the LOOK Node	71
Figure 64.	The LOOK Transformation	72
Figure 65.	Calculating the Front and Back Boundaries	73
Figure 66.	Adding the FOV Node	74
Figure 67.	Adding the VIEWPORT Node	75
Figure 68.	The Final Display	75
Figure 69.	The Color Wheel	79
Figure 70.	A Simplified Display Tree for the Mechanical Arm	80
Figure 71.	Display Tree With Color Nodes	81
Figure 72.	An Interactive Intensity Node	84
Figure 73.	Depth-Clipping Enabled for a Viewing Area	85

GRAPHICS PRINCIPLES

Figure 74.	Objects Outside the Front and Back Boundaries	86
Figure 75.	Display Tree With Depth-Clipping Node	87
Figure 76.	Display Tree for a Group of Labeled Objects	89
Figure 77.	Display Tree With Character Font Nodes	90
Figure 78.	Simplified Display Tree for a Car	91
Figure 79.	Display Tree With Conditional Referencing Nodes	92
Figure 80.	Display Tree for a Contour Map	93
Figure 81.	Display Tree With Level-Of-Detail Nodes	95
Figure 82.	Conditional Nodes for Blinking	96
Figure 83.	Display Tree for Alternate Display of Two Objects	97
Figure 84.	The SET PICKING ON/OFF Node	99
Figure 85.	Making the Components Pickable	100
Figure 86.	Display Tree for Simple Interaction	106
Figure 87.	The SET DEPTH_CLIPPING Node	107
Figure 88.	Representation of a Function	108
Figure 89.	The F:DZROTATE Function	112
Figure 90.	The Initial Function Instance DIALS	113
Figure 91.	Inputs to a Rotate Node	114
Figure 92.	Simple Z-Rotation Network	114



1. CREATING PRIMITIVE OBJECTS

A graphics programmer using the PS 300 for designing, viewing, and manipulating objects begins by creating a *data base* of the mathematical information that defines the objects. Objects are defined as two-dimensional or three-dimensional shapes consisting of points and lines or planes. Objects defined as points and the lines that connect them are *wire-frame* models. Objects defined as planes differ from wire-frame models because they contain surface or solid information.

The data space in which the programmer models objects is known as the *world coordinate system*. This system provides a way of expressing the location of all the points which define the object.

The simplest object in a graphical data base is a *primitive*. This consists entirely of points and lines or planes. The points specify the *geometry* of the object; the lines or planes specify the *topology*.

COORDINATE SYSTEMS

The PS 300 displays convincing three-dimensional images of mathematically-defined objects. All mathematical information that the designer enters to create an object (the *data base*) must be given in terms of a three-dimensional *coordinate system*. A coordinate system is a way of specifying a three-dimensional space in which objects can be modeled.

Right-Hand Coordinate System

One conventional method for representing three-dimensional space uses three lines (*axes*) originating at a common point in space (the *origin*) and drawn at right-angles to each other in the dimensions of height, width, and depth. These axis are labelled X (width), Y (height), and Z (depth). Figure 1 represents a commonly-used coordinate system known as the right-hand coordinate system.

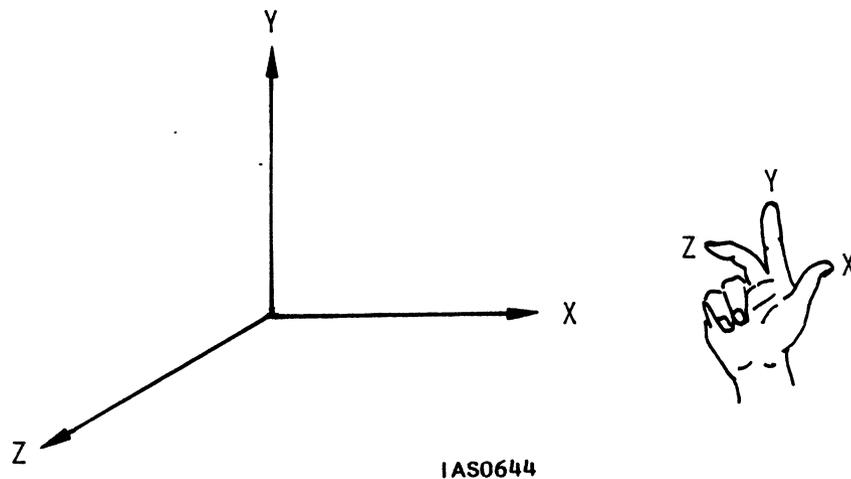


Figure 1. Right-Hand Coordinate System

As Figure 1 shows, the thumb and first two fingers of the right-hand can be used as a mnemonic for the names and positive directions of the axes in this system.

There is a disadvantage to this coordinate system for modeling with a computer graphics system. If you consider the computer screen to be parallel to the XY plane of this three-dimensional space, then positive values in the Z axis (depth) increase towards the eye of the viewer. The depth of an object displayed on the screen should be perceived as a dimension into the screen. So a coordinate system is needed with a Z axis that has positive values which increase into the screen away from the viewer.

Left-Hand Coordinate System

A left-hand coordinate system, employed by many computer graphics systems including the PS 300, has a Z axis in which positive values increase away from the viewer. Figure 2 shows a representation of the left-hand coordinate system.

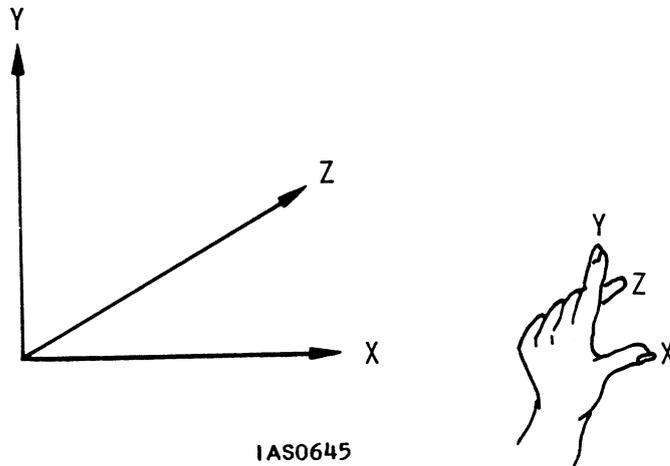


Figure 2. Left-Hand Coordinate System

Note that the thumb and first two fingers of the left hand indicate the positive direction of the axes in this coordinate system.

The World Coordinate System

The left-hand coordinate system with which the PS 300 graphics programmer works is known as the *world coordinate system*. The world coordinate system provides a way of expressing the mathematical data which the computer needs to create, display, and manipulate models in three dimensions. Figure 3 is a representation of the world coordinate system used in programming the PS 300.

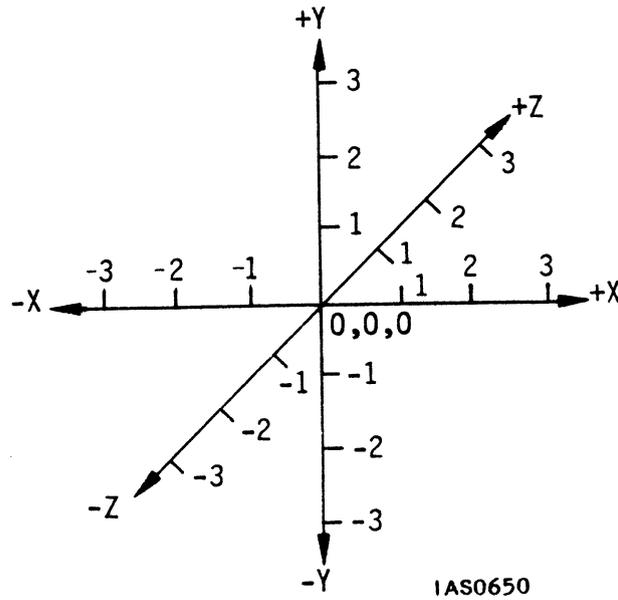


Figure 3. The World Coordinate System

All axes have a positive direction and a negative direction, and values are assigned for every point along an axis. The point at which the three axes meet is the *origin*.

DATA BASE FOR AN OBJECT

A data base for an object consists of points and lines (if the object is a *wire-frame* model) or planes (if the object is a *polygonal* model) expressed in world coordinate values. The points, lines, and planes define the *geometry* and *topology* of the object.

Geometry

The geometry of an object is the location in the world coordinate system of the points which define it. If, for example, you want to create a square centered at the origin of the world coordinate system with sides five units long, then the coordinates of the four points A, B, C, and D that define the square are as shown in Figure 4.

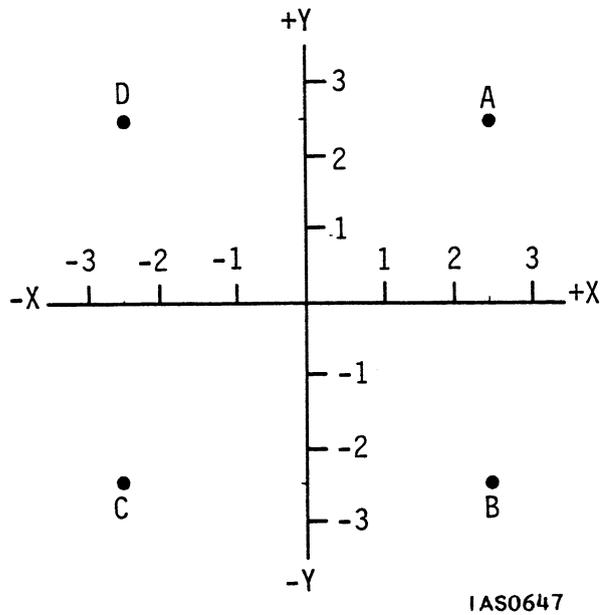


Figure 4. Coordinates of a Square

When these coordinates are connected with lines, the result is the square shown in Figure 5.

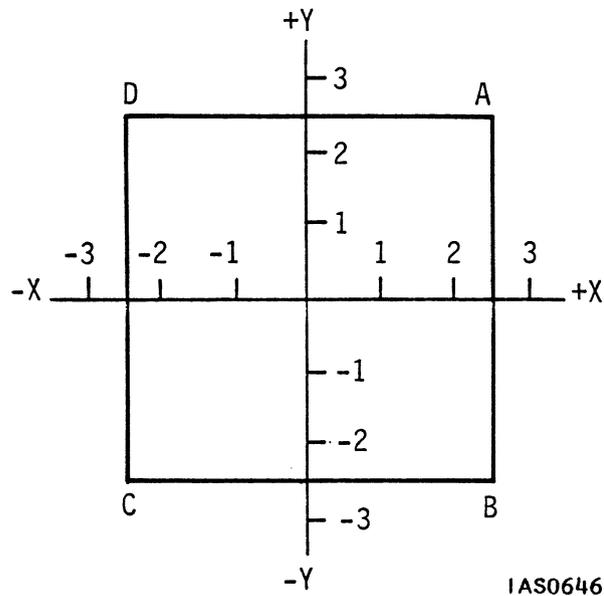


Figure 5. Location of the Square in the World Coordinate System

Coordinate Notation

The convention for defining coordinates in three-dimensional space is to give the X component first, then the Y component, and finally the Z component.

For example, point A is 2.5 units in the positive X axis, 2.5 units in the positive Y axis, and zero units in the positive Z axis, since the square is a two-dimensional figure. The notation for this coordinate is (2.5,2.5,0) or just (2.5,2.5) with the value for Z defaulting to zero. Point B is also 2.5 units in the positive X axis, but 2.5 units in the negative Y axis, and zero units in the positive Z axis. The notation for this coordinate is (2.5,-2.5,0) or just (2.5,-2.5).

The coordinates of the four corners of the square are as follows:

- Point A: (2.5,2.5,0) or (2.5,2.5)
- Point B: (2.5,-2.5,0) or (2.5,-2.5)
- Point C: (-2.5,-2.5,0) or (-2.5,-2.5)
- Point D: (-2.5,2.5,0) or (-2.5,2.5)

Position and line indicators are essential in vector lists for shapes that are not closed figures. For example, to draw just the left and right sides of the square, a vector list such as the following is needed.

```
Sides := VECTOR_LIST ITEMIZED N = 4 P -2.5,2.5 L -2.5,-2.5  
P 2.5,2.5 L 2.5,-2.5;
```

Polygon List

A *polygon* is a set of points that enclose and define a plane or surface. Just like a vector list, a polygon list contains the coordinates of the endpoints of the lines that make up the polygon. Unlike a vector list, a polygon list does not have to repeat the first point to close the figure, since by definition a polygon is a closed figure.

The following command creates a square as a polygon list.

```
Square := POLYGON 2.5,2.5 2.5,-2.5  
-2.5,-2.5 -2.5,2.5;
```

Only four items are needed in the polygon list to specify the topology of the square when it is defined as a polygon. Polygons can be only created on the PS 340 system. They are discussed fully in the tutorial module "Using the PS 340."

GRAPHICAL PRIMITIVES

Vector lists and polygon lists contain all the information needed to specify the geometry and topology of an object. Objects specified as vector lists or polygon lists are known as *graphical primitives*. The VECTOR_LIST and POLYGON commands are the two most commonly used to create primitives locally in the PS 300. Complex primitives are often created by a host application program and transferred to the PS 300 to be manipulated and viewed.

Curve Primitives

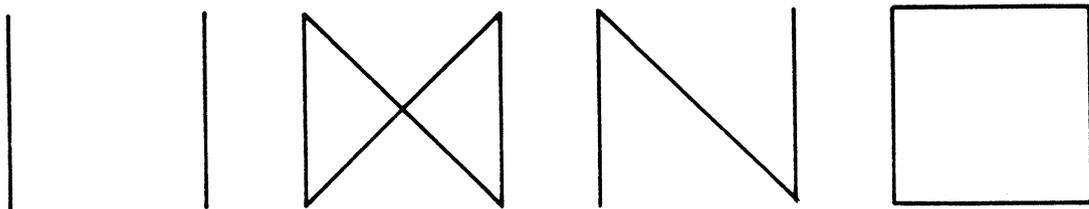
The examples used so far have been for primitives consisting of straight lines only. Other commands, such as the BSPLINE and POLYNOMIAL commands create curve primitives locally in the PS 300. For more information on these commands refer to the *Command Summary* in Volume 3A.

Text Primitives

Text is also treated as a graphical primitive in the PS 300. A standard 128-character ASCII set is provided with the system. The characters which compose this standard font are created as vector lists, so you do not have to create your own. However, if you want to create different fonts that can be used as a supplement to the standard font, there is a command which allows you to do this. The BEGIN_FONT ... END_FONT command lets you create 128 separate vector lists defining the characters which compose the font and assign them a single name. This font can be substituted for the standard font using the CHARACTER_FONT command. For more details, refer to the "Text Modeling" module and the *Command Summary* in Volume 3A.

Same Geometry but Different Topologies

Primitive objects can have the same geometry, but different topologies. That is, the same set of world coordinates can be connected by lines to create open figures or polygons of different sorts, as shown in Figure 6.



IAS0648

Figure 6. Primitives With the Same Geometry and Different Topologies

For example, the capital letter "N" can be created by the following vector list.

```
Capital_N := VECTOR_LIST N = 4 -2.5,-2.5 -2.5,2.5  
2.5,-2.5 2.5,2.5;
```

The bow tie shape can be created by the following vector list.

```
Bow_Tie := VECTOR_LIST N = 5 -2.5,-2.5 -2.5,2.5 2.5,-2.5  
2.5,2.5 -2.5,-2.5;
```

For open figures, such as the two parallel lines, position and line identifiers must be included in the vector list. The following command creates the two parallel horizontal lines as a single primitive.

```
Lines := VECTOR_LIST ITEMIZED N = 5 P-2.5,2.5 L2.5,2.5  
P-2.5,-2.5 L2.5,-2.5;
```

Although the geometry is the same for all of these objects, their topologies are different, and so their vector lists are different. Each object must be defined as a separate primitive with its own vector list.

Same Topology but Different Geometries

Primitives can also share the same topology and have different geometries. All of the four-sided shapes in Figure 7, for instance, consist of four points connected in the same manner.

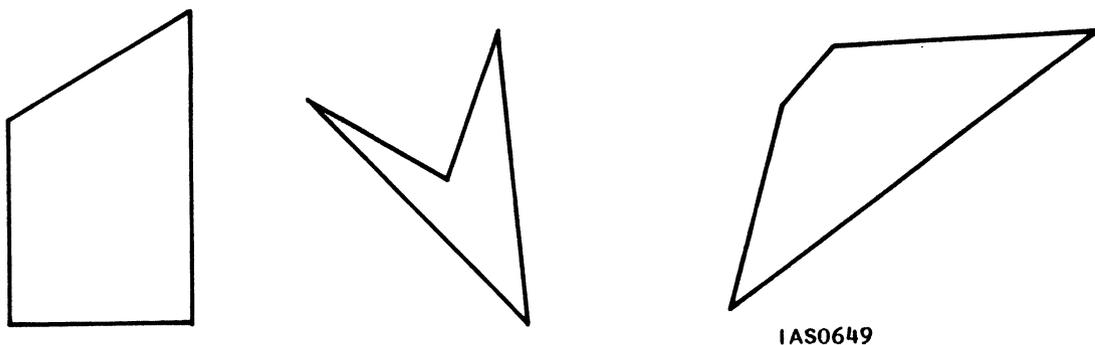


Figure 7. Primitives With the Same Topology and Different Geometries

Each of these objects must be defined as a separate primitive. However, as the next section (Transforming Primitives) shows, there are ways of changing the geometry of a primitive to create a new object without creating a new primitive.

SUMMARY

New Information Presented

1. To express the mathematical data which defines an object for graphical display, a programmer uses a coordinate system.
2. The coordinate system most useful for computer graphics purposes is a left-hand coordinate system. This coordinate system has a Z axis that has positive values which increase away from the eye of the viewer.
3. The coordinate system used in creating a data base for graphical objects is called the world coordinate system.
4. To create a model of an object with a graphics computer, you need to specify two things:
 - The positions of the endpoints of each line, expressed as three-dimensional (X, Y, Z) coordinates. This is known as the geometry of the object.
 - The way in which those points are connected by lines. This is known as the topology of the object.
5. The geometry and topology together form a vector list or polygon list for a graphical primitive. A primitive defined by a vector list is composed of lines. A primitive defined by a polygon list is composed of planes or surfaces.
6. Other primitives composed of points and lines are curves and text. Primitives of all sorts can be created locally using PS 300 commands. They can also be generated by a host application program and sent to the PS 300.

What Next?

At this point, you can create a graphical data base for a primitive. Vector lists define wire-frame objects made of lines, and polygon lists define objects made of planes. In the next section you will see how to apply mathematical transformations to primitives to create new objects. These new objects will have the same topology as the primitives, but their geometries will be different.

2. TRANSFORMING PRIMITIVES

Mathematical operations called *transformations* can be 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. Transformations create a new object, based on the definition of the old one, which has the same topology as the primitive, but a different geometry.

Using transformations, you can, in effect, move primitives around in the coordinate system or add numerous different objects to the data base using a small number of primitive shapes.

CREATING NEW OBJECTS FROM PRIMITIVES

The data base of shapes so far consists of a square with sides five units long. If you want to add to the data base a two-dimensional diamond shape with sides that are five units long centered at the origin of the world coordinate system, you could create it as a primitive by entering a vector list like this.

```
Diamond := VECTOR_LIST N = 5 0,3.54 3.54,0 0,-3.54  
-3.54,0 0,3.54;
```

The diamond will be located in the world coordinate system as shown in Figure 8.

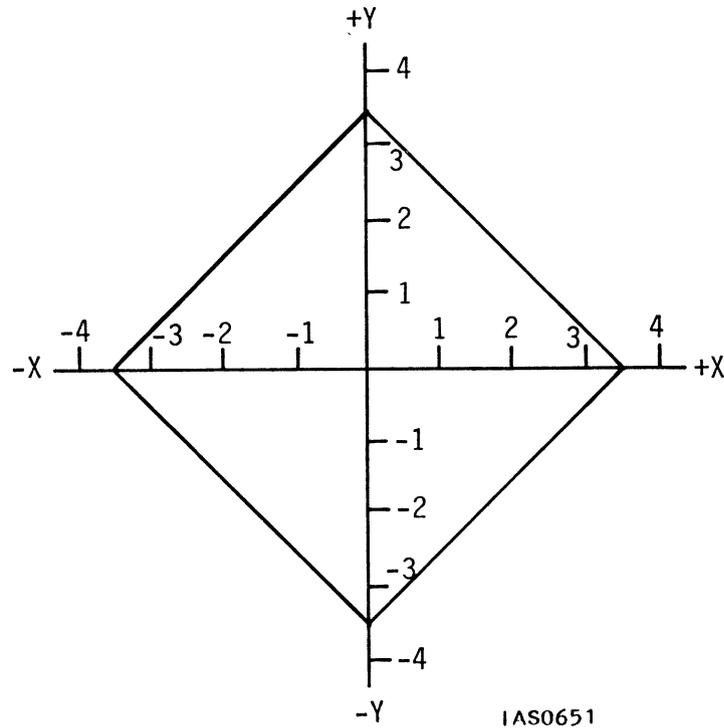


Figure 8. Location of the Diamond

Notice that the Diamond and the Square primitive that already exists share several features. They are both two-dimensional figures, they are the same size (5 units per side), and they have the same topology. In fact, the only difference between the two figures is their geometry. The points that define the four corners of the Square and the Diamond are in different locations within the world coordinate system. The diamond shape could be described as the square shape rotated 45 degrees around the Z axis.

Since the two objects share these relationships, there would be no need to create a separate primitive if there were some way to change the geometry of the square while maintaining its topology. PS 300 commands exist which do exactly that.

Applying Transformations

With the PS 300, you can apply mathematical operations to primitives that already exist to move them around in the coordinate system or create new shapes from them. The resulting objects are not defined as primitives themselves. Instead, they are structures which consist of *matrix transformations* applied to the coordinates which define a primitive.

Transformations are operations of matrix algebra which change the geometry of a graphical object, but do not affect the topology. When you create a vector list for an object, you have to calculate the coordinates of the points yourself. When you apply transformations to existing primitives, the PS 300 calculates the new coordinates for you. It is easier, for example, to create the diamond by rotating the square than to calculate yourself the coordinates of the diamond primitive. The following PS 300 command creates a diamond by rotating the square.

```
Diamond := ROTATE IN Z 45 APPLIED TO Square;
```

The structure of the diamond can be diagrammed as shown in Figure 9.

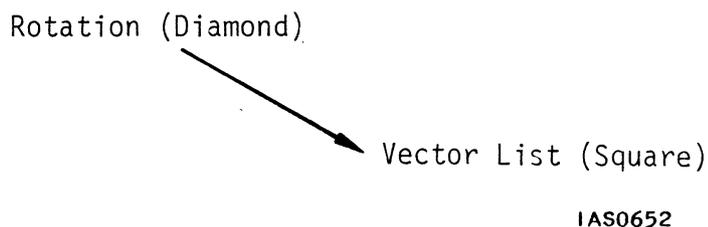


Figure 9. The Structure of the Diamond

The Diamond is shown as a rotation transformation applied to the vector list defining the Square.

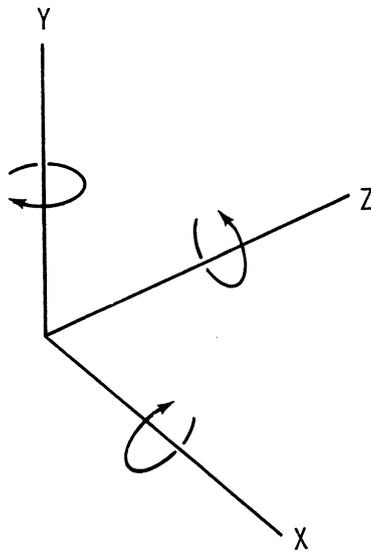
MODELING TRANSFORMATIONS

Transformations which are used to create new objects by changing the geometry of already defined primitives are often referred to as *modeling transformations*. There are three modeling transformations: *rotation*, *translation*, and *scaling*. The module called "Modeling" gives examples of the use of modeling transformations to create and position the parts of a complex object.

Rotation

A new object can be created by rotating a primitive through any number of degrees in any of the three dimensions. To perform a rotation on a primitive, the computer uses the sine and cosine of the angle specified in the rotate command to create a rotation matrix, which is applied to the points in the vector list.

When an object is rotated in the world coordinate system, it rotates around one of the X, Y and Z axes in the directions in Figure 10.



IAS0653

Figure 10. Rotation in the World Coordinate System

Rotations Around an Axis

Note the terms used to express rotations. A rotation "in X" means rotation around the X axis or in a plane parallel to the X axis, and so on. To determine the direction of rotation around an axis, use the left-hand coordinate mnemonic. 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.

Rotations always occur around one of the world coordinate axes. Consider a new object called Rot_Arrow created by rotating an existing 2D arrow which is centered at the origin through 120 degrees in positive Z.

```
Rot_Arrow := ROTATE IN Z 120 APPLIED TO Arrow;
```

The orientation of the rotated arrow will be as shown in Figure 11.

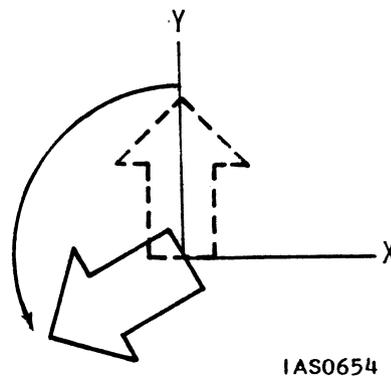


Figure 11. Orientation of the Rotated Arrow

The primitive arrow is drawn with dashed lines; the rotated arrow is drawn with solid lines. Since the primitive arrow was created with its base at the origin, the rotated arrow is based at the origin also.

If an object is not centered at the origin, however, and a rotation is applied, the rotation about the world axis will have the effect of "swinging" the object around the axis, as illustrated in Figure 12.

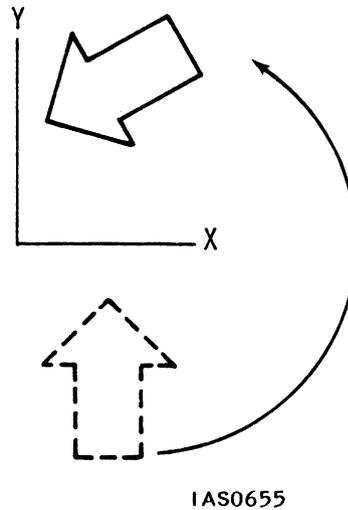


Figure 12. Rotation of an Object Not Centered at the Origin

Rotating an object while it is centered at the origin, then, effectively rotates it about its own center. Rotating an object which is not at the origin swings that object around one of the world axes to a new location in the world coordinate system.

Translation

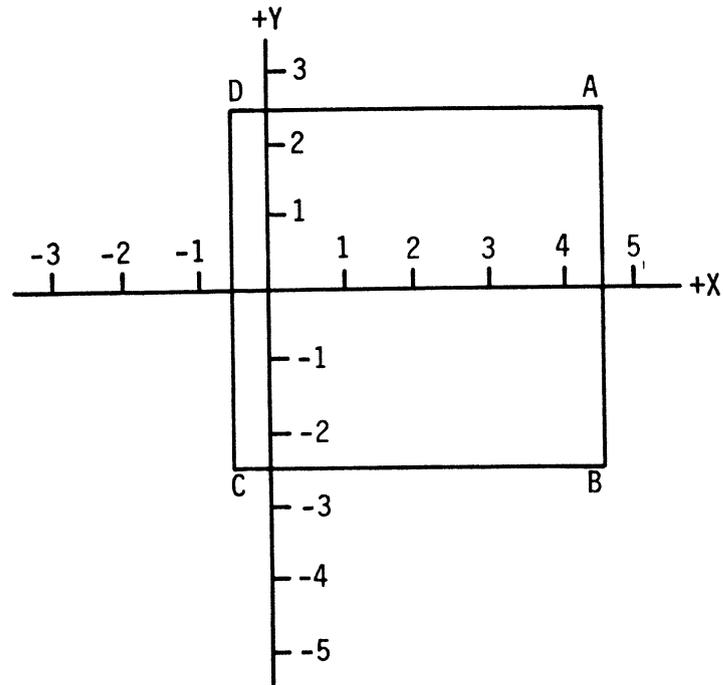
Translating an object means moving it to a new location in the world coordinate system. An object which is translated in X is moved in the X direction. An object translated in X and Y is moved some distance in the X direction and some distance in the Y direction.

The PS 300 performs translations on a primitive by adding the X, Y, and Z values specified in the translation command to the coordinates of each vector.

Consider a new square, created by translating the Square defined earlier by 2 units in the positive X axis.

```
Trans_Square := TRANSLATE 2,0 APPLIED TO Square;
```

The location of Trans_Square will be as shown in Figure 13.



IAS0656

Figure 13. Location of the Translated Square

Notice that in a translation in X, the X component of each coordinate is changed (in this case, increased by 2) but the Y and Z components are not.

Translations in All Three Axes

The PS 300 performs translations in any direction (X, Y, or Z) and in any combination of directions. For example, a translation of 2 units in positive X and 2 units in negative Y can be applied to Square.

```
New_Trans_Square := TRANSLATE 2,-2 APPLIED TO Square;
```

The new translated square will be located as shown in Figure 14.

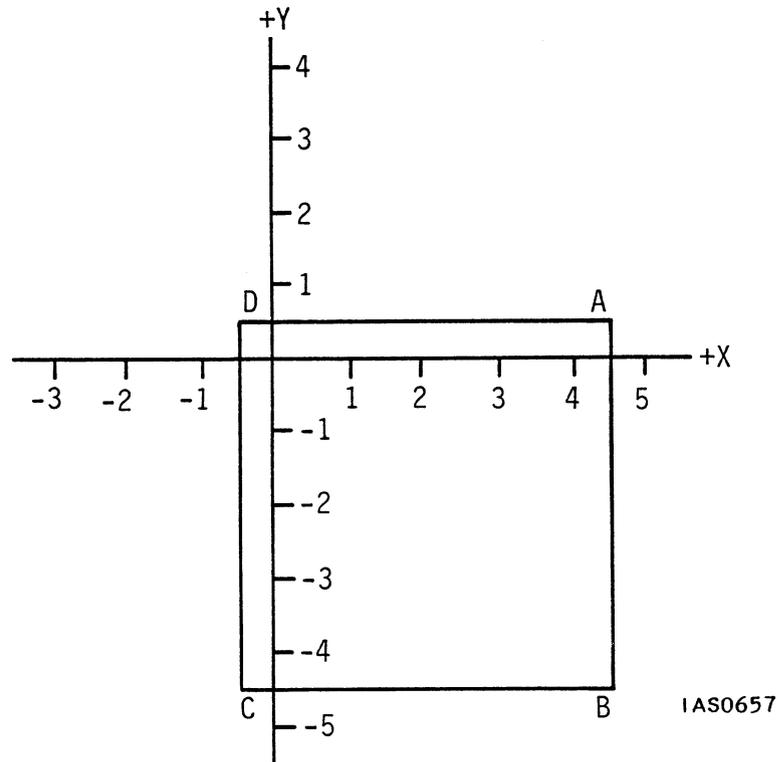


Figure 14. Square Translated in X and Negative Y

Naturally, translations may be specified in three dimensions.

The notation used for representing translations is to give the X component, the Y component, then the Z component, separated by commas. So, for example, a translation of 3,-2,4 is 3 units in X, 2 units in negative Y, and 4 units in Z.

Scaling

Scaling an object makes it smaller or larger, depending on the scale factor that is specified. The PS 300 creates a scaling matrix which multiplies the points in the vector list by the scale factor in the scaling command to determine the new coordinates of the scaled object.

For example, a small square can be created by scaling the square defined at the origin of the world coordinate system by 0.5.

Small_Square := SCALE BY .5 APPLIED TO Square;

The small square will have the coordinates shown in Figure 15.

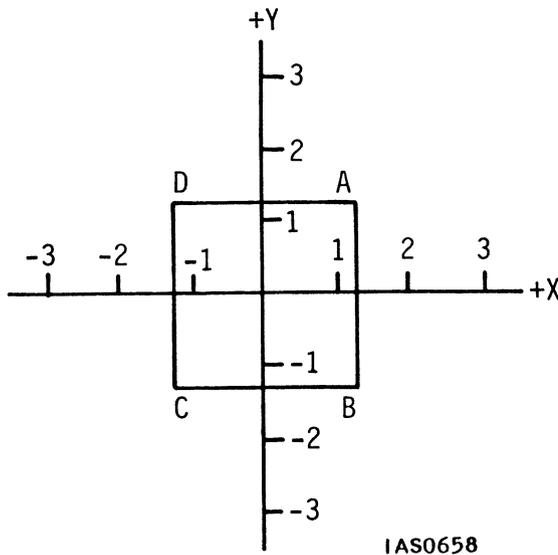


Figure 15. Scaling the Square

This type of scaling is called *uniform scaling*. The new object is created by scaling the primitive by the same amount in all dimensions.

Another type of scaling, *non-uniform scaling*, consists of scaling an object by different amounts in different dimensions. For example, a rectangle can be created by scaling the Small_Square by 2 units in X only.

Rectangle := SCALE 2,1,1 APPLIED TO Small_Square;

The rectangle will have the following coordinates (Figure 16).

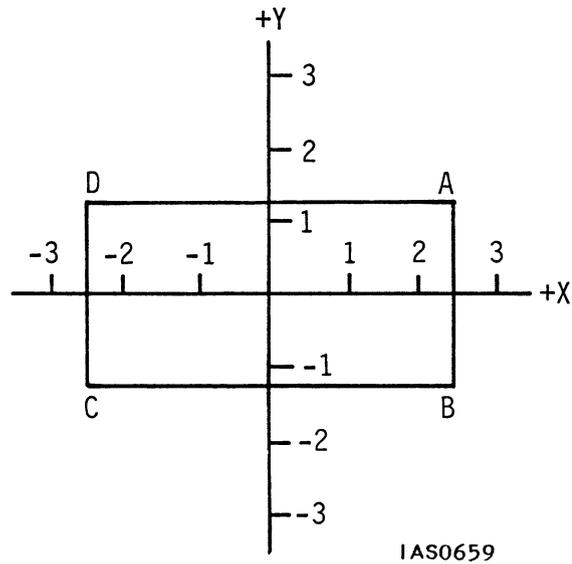


Figure 16. Non-Uniform Scaling to Create a Rectangle

Non-uniform scaling is a commonly-used modeling transformation; it distorts the shape of a primitive to produce a new object. For example, a non-uniform scale in Y and Z applied to a cube at the origin can create an object with the relative dimensions of a building brick. Circles can be scaled non-uniformly to create ellipses, and spheres to create ellipsoids, and so on.

THE ORDERING OF TRANSFORMATIONS

When a series of transformations is applied to a primitive, the order in which the transformations are applied always determines the final location and orientation of the object in the world coordinate system.

For example, consider a 2D arrow which has been created within the world coordinate system as shown in Figure 17.

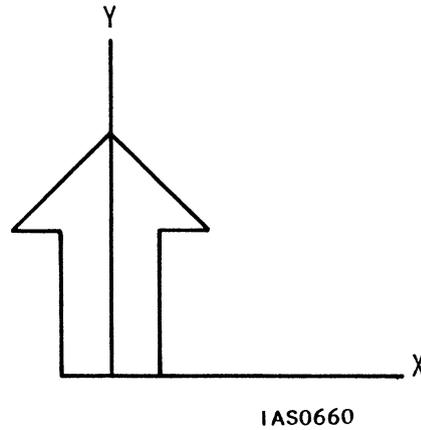


Figure 17. A Two-Dimensional Arrow

If the arrow is rotated 45 degrees in Z, rotation occurs around the Z axis. The rotated arrow (Arrow_1) is oriented as shown in Figure 18.

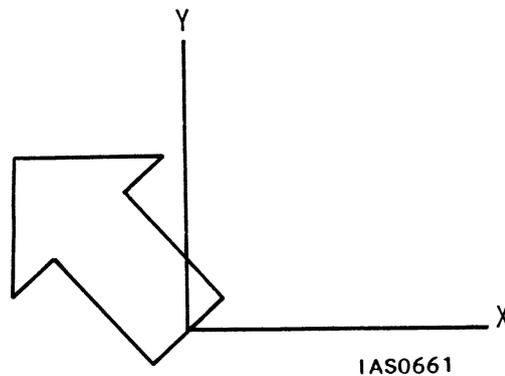


Figure 18. Rotated Arrow

A new object called Arrow_2 is now created by applying a translation in positive X and negative Y to the rotated arrow. The orientation of the translated arrow is still a rotation of 45 degrees in the plane of the Z axis, but its location would be something like this (Figure 19).

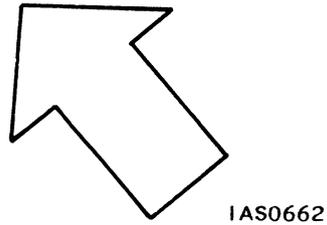
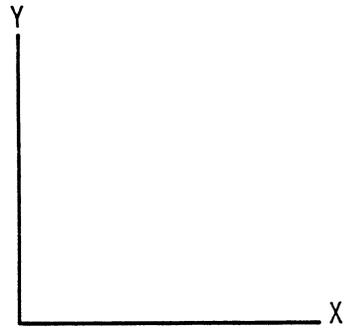
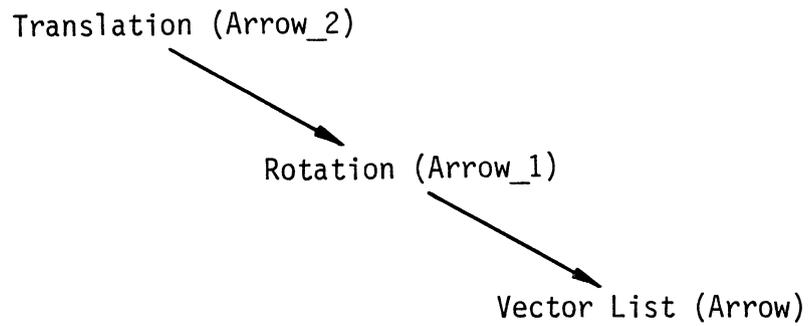


Figure 19. Arrow Rotated, Then Translated

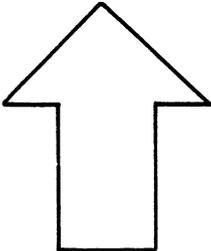
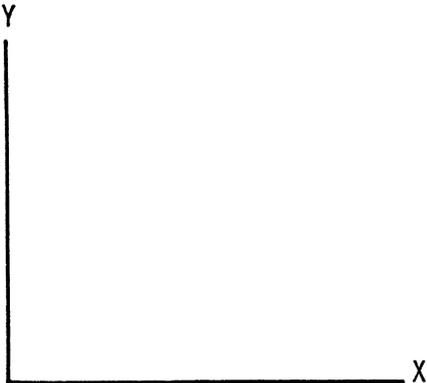
The structure of Arrow_2 is a translation pointing to a rotation, pointing to a vector list. It can be diagrammed as shown in Figure 20.



IAS0663

Figure 20. The Structure of Arrow_2

Now consider what happens if the original arrow is translated first, and then is rotated. Translating the arrow in positive X and negative Y creates an object (Arrow_3) located in the world coordinate system as shown in Figure 21.



IAS0664

Figure 21. Translated Arrow

If a rotation of 45 degrees in Z is now applied to the translated arrow, the new object Arrow_4 will "swing" around the Z axis to a new location in the world coordinate system (Figure 22).

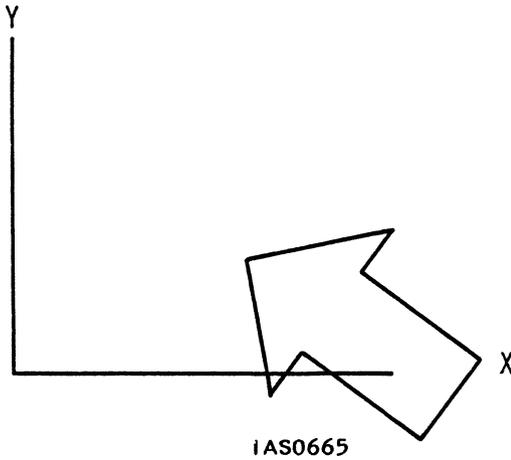


Figure 22. Arrow Translated, Then Rotated

The structure of Arrow_4 is a rotation pointing to a translation pointing to a vector list. It can be shown as follows (Figure 23).

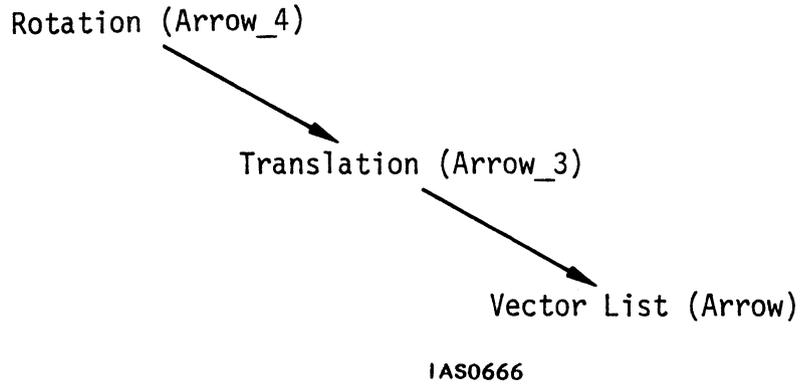


Figure 23. The Structure of Arrow_4

The order in which transformations are applied to objects determines the ultimate location and orientation of the new object in the world coordinate system. *The same transformations applied to the same primitive in a different order produce different results.* When you are applying a series of transformations to an object, you must take care to apply those transformations in the correct order to get the result you want.

Transformation Matrices

Translations, rotations, and scalings are the three basic transformations which are applied to data in a computer graphics system. We have called these three *modeling transformations*.

As you will see in Section 5, other transformations called *viewing transformations* can be applied to data to create different views of objects—for example, top views, side views, or perspective views. Although viewing transformations are more complex, they are still combinations of translations, rotations, and scales.

Later sections also describe how transformations can be applied *interactively* to data. Values from the keyboard, data tablet, dials, and buttons can be used to apply a series of transformations in rapid succession, giving the illusion of movement to displayed objects.

All transformations applied to graphical data are performed by matrix algebra. The most commonly used matrices in computer graphics are 2X2 (two-dimensional rotations and scales for characters and text strings); 3X3 (three-dimensional rotations and scales for objects); and 4X3 and 4X4 (most of the viewing transformations described in Section 5).

All matrices are governed by the laws of matrix algebra. Of particular interest to the graphics programmer is the law that matrix A times matrix B does not equal matrix B times matrix A. This property is known as the *non-commutativity* of matrices. The non-commutativity of matrices makes the careful ordering of transformations necessary in graphics programming.

When a transformation is applied to an object, the new coordinates of the vectors which compose the object are calculated by multiplying the old coordinates by the elements in the matrix.

When more than one transformation is applied to graphical data, the matrices are *concatenated*. This means that each matrix is pre-multiplied to a matrix called the *current transformation matrix*. The current transformation matrix 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. A 4X4 current transformation matrix is large enough to handle all of the transformations needed for computer graphics operations.

Matrix concatenation works like this. Suppose you want to scale a primitive to twice its size, rotate it 180 degrees in Z, and then translate it in X and Y. Instead of applying three separate matrices to the points that define the object, the PS 300 pre-multiplies the matrices that represent these transformations into the current transformation matrix. This single matrix is then applied to the vector list that defines the object.

The current transformation matrix (CTM) starts out as an *identity matrix*, as shown in Figure 24.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

IAS0742

Figure 24. An Identity Matrix

An identity matrix is composed of ones and zeros, with the ones running in a diagonal. Multiplying by an identity matrix is the equivalent of multiplying by one: nothing changes. Each transformation matrix in turn--scale, rotate, and translate--is pre-multiplied to the identity matrix. The result is a CTM which consists of the cumulative transformations and the order in which they are to be applied to the data. The vector list defining the object is run through the CTM as the last stage in the process, as shown in Figure 25.

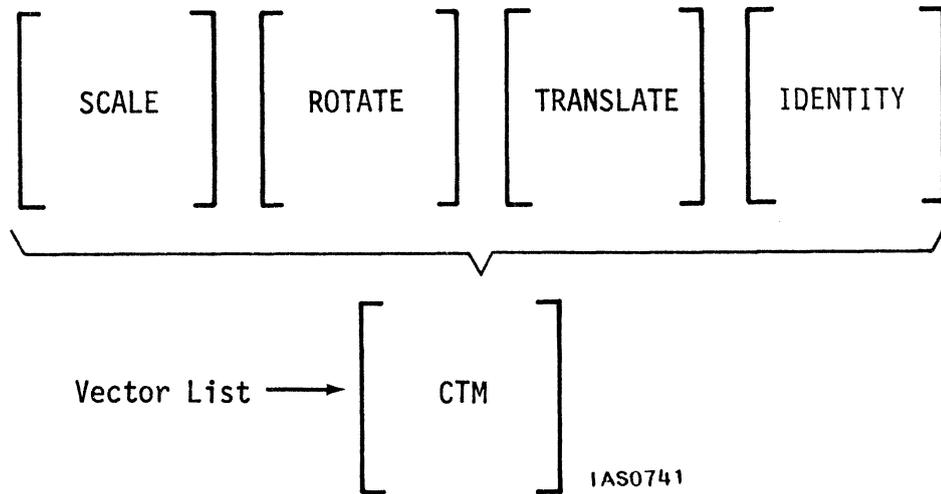


Figure 25. Concatenating Matrices

The transformed vectors which result form the points and lines of the newly-oriented object. If the order of the transformations were changed, then the final CTM would be different. If this matrix were applied to the data defining the object, the ultimate location and orientation in the world coordinate system of the transformed object would change.

For more information about matrix algebra, consult Newman W.M. and Sproull R.F., Principles of Interactive Computer Graphics, Second Edition, McGraw-Hill, 1979. This text contains an appendix which introduces vectors and matrices.

SUMMARY

New Information Presented

1. New objects can be created by applying transformations to primitives.
2. Transformations change the geometry of the primitives but leave their topology the same.
3. Three basic transformations are translations, rotations, and scales.
4. When more than one transformation is applied to an object, the order in which the transformations are applied affects the final location and orientation of the object in the world coordinate system.
5. All transformations are applied through matrix algebra. Transformations are concatenated into a single matrix known as the current transformation matrix.
6. Matrices are said to be non-commutative. That is, matrix A times matrix B does not equal matrix B times matrix A. The non-commutativity of matrix multiplication requires the careful ordering of transformations to be applied to graphical data.

What Next?

By applying matrix transformations to existing primitives you are now able to move objects around and create new objects of different sizes and shapes.

In the next section, you will see how to create compound objects. Commands exist to group collections of primitives and transformations under one name. The resulting compound object can be transformed as a single entity.

3. CREATING COMPOUND OBJECTS

Compound objects can be created with the PS 300 using primitives and transformations. Primitive objects and transformed primitives can be grouped into one named object which can be transformed as a single entity.

BUILDING WITH PRIMITIVES AND TRANSFORMATIONS

No matter how complicated an object is, you can create it as a primitive by figuring out the vector list or polygon list needed to specify the coordinates of all the line endpoints and the way in which those points are connected. An alternative, however, is to use primitives and transformations as building blocks to create new objects which are compound structures.

Creating a Star Primitive

If, for example, you want to create an eight-pointed star centered at the origin, making the object out of lines (not polygons) five units long, you could create it as a primitive by entering the following vector list:

```
Star := VECTOR_LIST ITEMIZED N = 10 P 0,3.54 L 3.54,0 L 0,-3.54
      L -3.54,0 L 0,3.54 P 2.5,2.5
      L 2.5,-2.5 L -2.5,-2.5
      L -2.5,2.5 L 2.5,2.5;
```

Notice that this form of the vector list has the word "itemized" and has "P" and "L" identifiers preceding each coordinate. This is necessary because the star shape cannot be drawn as a set of continuous lines.

The new primitive, Star, created by this command is located in the world coordinate system as shown in Figure 26.

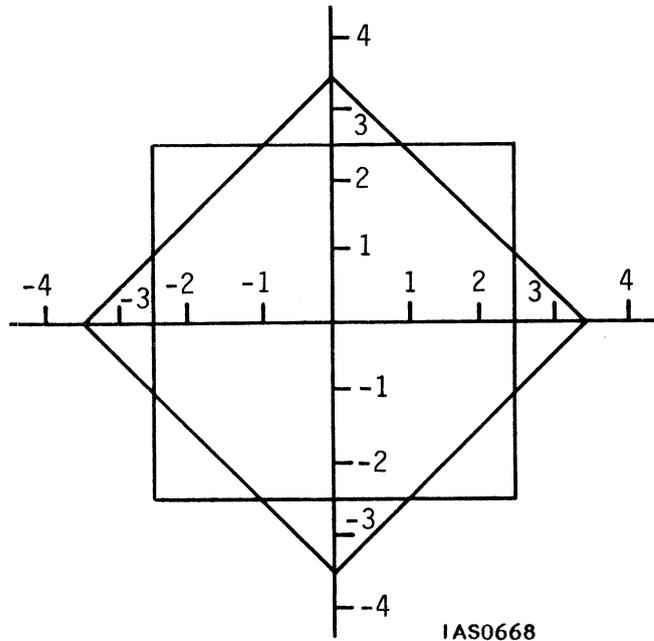


Figure 26. Location of the Star Primitive

When Star is displayed with the correct viewing transformations applied to it (these are discussed in Section 5), it will be located on the screen as shown in Figure 27.

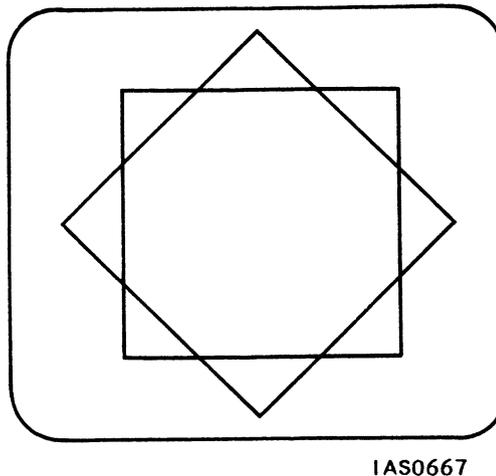


Figure 27. The Star Primitive Displayed on the Screen

The same shape can be displayed using existing primitives without adding a new primitive to the graphical data base. If you display at the same time the Square primitive and the Diamond primitive that already exist in the graphical data base, the picture on the screen will look the same as when you displayed the Star primitive.

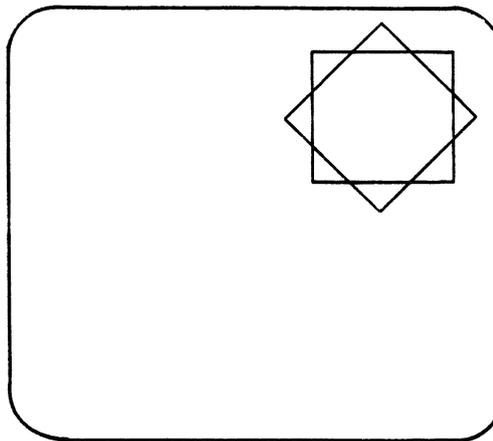
The advantage of using the Square and the Diamond is that you do not have to calculate the coordinates for the Star primitive vector list. Your task as a programmer is simplified by using existing objects.

If however, you want to do more than just display a picture of the star--if you want to apply transformations to the star to rotate or translate it, for example--the new primitive is easier to use than the Square and Diamond.

If you want to create a small star and move it to the upper-right corner of the screen, you can create the small star by scaling the primitive and then apply a translation in positive X and positive Y to the small star.

```
Scale_Star := SCALE BY .25 APPLIED TO Star;
Trans_Star := TRANSLATE .5,.5 APPLIED TO Scale_Star;
```

When displayed, Trans_Star will appear like this on the screen (Figure 28).



IAS0669

Figure 28. The Location of Trans_Star on the Screen

The structure of Trans_Star can be diagrammed as shown in Figure 29.

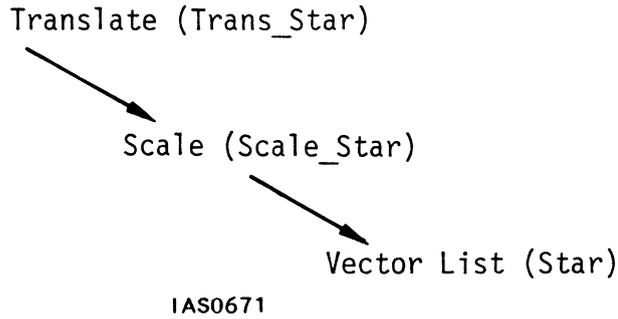


Figure 29. The Structure of Trans_Star

If you use the Square primitive and the Diamond structure (rotation applied to the Square) instead of the Star primitive, however, four new objects have to be created and displayed to get the same picture.

You must create a scaled square, a scaled diamond, a translated small square, and a translated small diamond, and display them together.

```
Scale_Square := SCALE BY .25 APPLIED TO Square;  
Scale_Diamond := SCALE BY .25 APPLIED TO Diamond;  
  
Trans_Square := TRANSLATE .5,.5 APPLIED TO Scale_Square;  
Trans_Diamond := TRANSLATE .5,.5 APPLIED TO Scale_Diamond;
```

The two structures look like this (Figure 30).

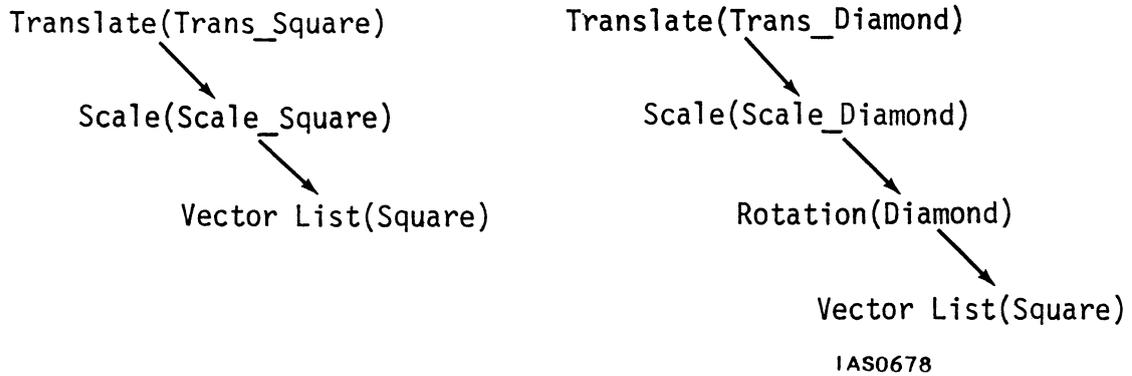


Figure 30. The Structures of Trans_Square and Trans_Diamond

When they are displayed together, Trans_Square and Trans_Diamond look just like Trans_Star. However, unless this shape can be manipulated as a single entity, some of the programming time and effort saved by not creating the star as a primitive will be lost.

Grouping Primitives and Transformations

The PS 300 allows you to construct a single named object from groupings of primitives and transformed primitives. The resulting compound structure represents an object which is composed of separate parts, but which can be treated as a single item, much like a primitive.

The INSTANCE command lets you create compound objects such as this:

```
Star1 := INSTANCE OF Diamond, Square;
```

The object called Star1 has the same dimensions and location in the coordinate system as Star, but it is not defined as a primitive vector list. It is a compound object which groups the two existing definitions Diamond and Square under a single name.

This compound object can be manipulated as easily as a primitive. A small star can be created by scaling Star1.

```
Scale_Star1 := SCALE BY .25 APPLIED TO Star1;
```

And the small star can be moved to the upper-right of the screen by translating Scale_Star1.

```
Trans_Star1 := TRANSLATE .5,.5 APPLIED TO Scale_Star1;
```

The structure for Trans_Star1 can be diagrammed as shown in Figure 31.

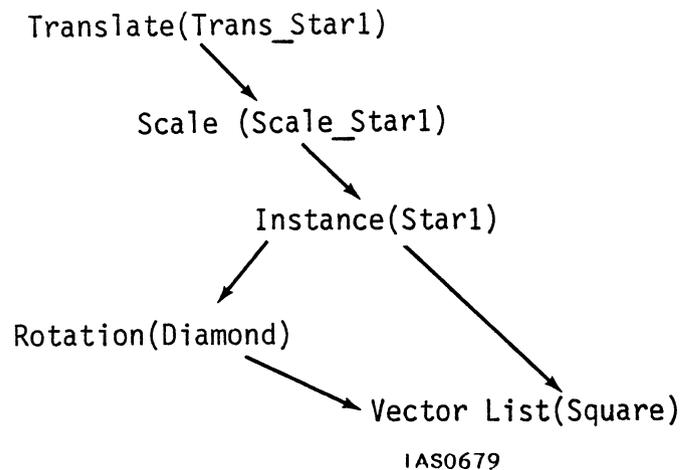


Figure 31. The Structure of Trans_Star1

The name Trans_Star1 identifies the translation which points to Scale_Star1. The scaling transformation points to the name Star1. Star1 groups the vector list defining the Square with the rotation that defines the Diamond. Both Diamond and Square share the same primitive definition. A complete set of commands which would create Trans_Star1 is as follows.

```

Trans_Star1 := TRANSLATE .5,.5 APPLIED TO Scale_Star1;
Scale_Star1 := SCALE BY .25 APPLIED TO Star1;
Star1 := INSTANCE OF Diamond, Square;
Diamond := ROTATE IN Z 45 APPLIED TO Square;
Square := VECTOR_LIST N = 5    2.5,2.5 2.5,-2.5 -2.5,-2.5
                               -2.5,2.5 2.5,2.5;
  
```

Unlike the separate parts it is composed of, the compound object named Star1 created by the INSTANCE command can now be treated as a single entity. The translation and scale transformations (Trans_Star1 and Scale_Star1) are applied directly to Star1. There is no longer any need to transform the Diamond and Square separately, now that they are grouped into a compound object.

There is also a structuring command BEGIN_STRUCTURE ... END_STRUCTURE which groups primitives and transformations into compound structures with a single name. Refer to the "PS 300 Command Language" tutorial module for details on using this command.

SUMMARY

New Information Presented

1. Compound objects can be created by grouping primitives and transformed primitives under a single name.
2. Groupings such as these can be treated as a single object. Transformations applied to the named compound object automatically apply to the parts it is composed of.

What Next?

The data base of graphical objects now consists of:

- Graphical primitives
- Transformed primitives
- Compound objects: structures consisting of primitives and transformed primitives grouped into one object.

In the next section, you will learn how compound objects are used to create complex models with parts that can be manipulated using the PS 300's interactive devices.



4. DESIGNING A MODEL FOR INTERACTION

The transformations discussed so far have been called modeling operations. They are equivalent in the real world to assembling the raw materials for a model and making the parts that the model is composed of. Complex 3D models consisting of separate parts are made by building each part as a compound object made of primitives and transformations. The parts are then grouped together to form the complete model.

Complex models are designed as a hierarchical structure called a *display tree*. The display tree shows the dependencies of parts within the model's structure and contains all the primitives and transformations needed to create the model in the PS 300's memory.

Models designed as hierarchical trees are a tremendously flexible design tool. Complicated models can be created in smaller parts and assembled as the designer requires. Changes can be made to any component of the model without affecting other parts. Interaction with the entire model or with any component is possible using the PS 300's dials, buttons, function keys, and data tablet.

The tutorial module "Modeling" gives an extended example of designing a model as a display tree.

DESIGNING A COMPLEX MODEL

The PS 300 can be used to model objects of any complexity. Consider the articulated mechanical arm shown in Figure 32.

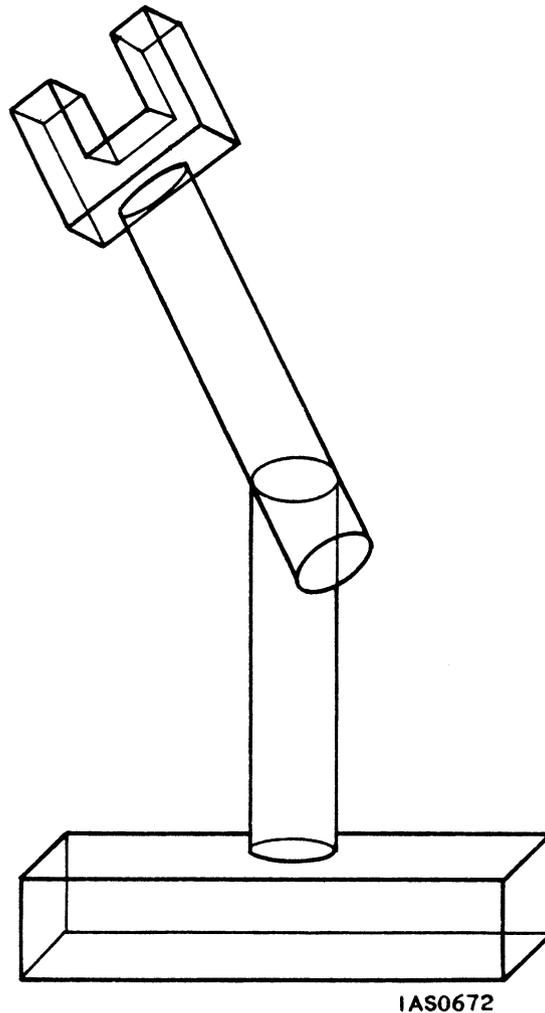


Figure 32. An Articulated Mechanical Arm

The arm consists of a base, two jointed sections, and a hand. The base is fixed and cannot move. The whole arm can rotate at the base. The two arm pieces and hand are affected by this movement. The movement at the "elbow" affects the upper arm and hand only. And movement at the "wrist" only affects the hand.

Clearly, a computer model which simulates this mechanical arm cannot be created as a primitive vector list or polygon list. Even if the object were created as a primitive by a host application program, it would not be a useful model of the mechanical arm. Transformations could be applied to translate, rotate, or scale the complete model, but there would be no way to interact with the individual parts. The arm could not be made to rotate at the base, the elbow joint would not bend, and the hand could not twist at the wrist.

Analyzing a Model as a Hierarchy

Complex models such as the mechanical arm which are to be designed on the PS 300 are analyzed to determine a *hierarchy* of the parts that compose the model, and to show their dependencies. A hierarchy is a principled organization of components. The organizing principle will vary depending on the relationship between components which the hierarchy is designed to show. The model for the Mechanical Arm, for example, can be represented by the hierarchy in Figure 33. This hierarchy shows the dependent and independent motion of the components.

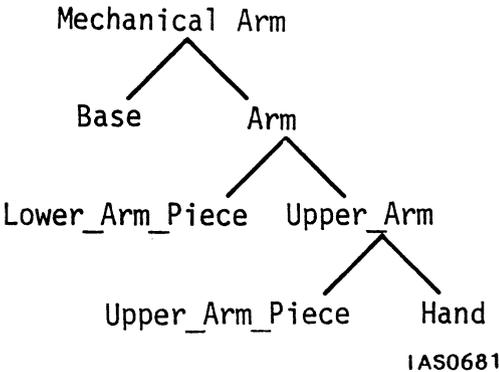


Figure 33. Hierarchy of Parts for the Mechanical Arm

This hierarchy shows that the model consists of a base and an arm. The arm consists of a lower arm and an upper arm. The upper arm is made up of the upper-arm piece and hand.

If the whole mechanical arm moves, then all the parts that compose it move too. If the arm moves, the lower-arm piece and upper arm move with it. If the upper arm moves, the upper-arm piece and hand move. The hand can also move on its own without affecting anything else.

DISPLAY TREES

For a complex model designed to be manipulated interactively with the PS 300, a hierarchy is drawn as a *display tree*. Much like a flowchart for a conventional computer program, a display tree represents the graphical primitives that must be created and the transformations that must be applied to create this model in the PS 300's memory. It also indicates the interaction points in the model's structure to which interactive devices will be connected to change the model dynamically.

Display Tree for the Mechanical Arm

The hierarchy that has been established for the Mechanical Arm can be used to create the display tree shown in Figure 34.

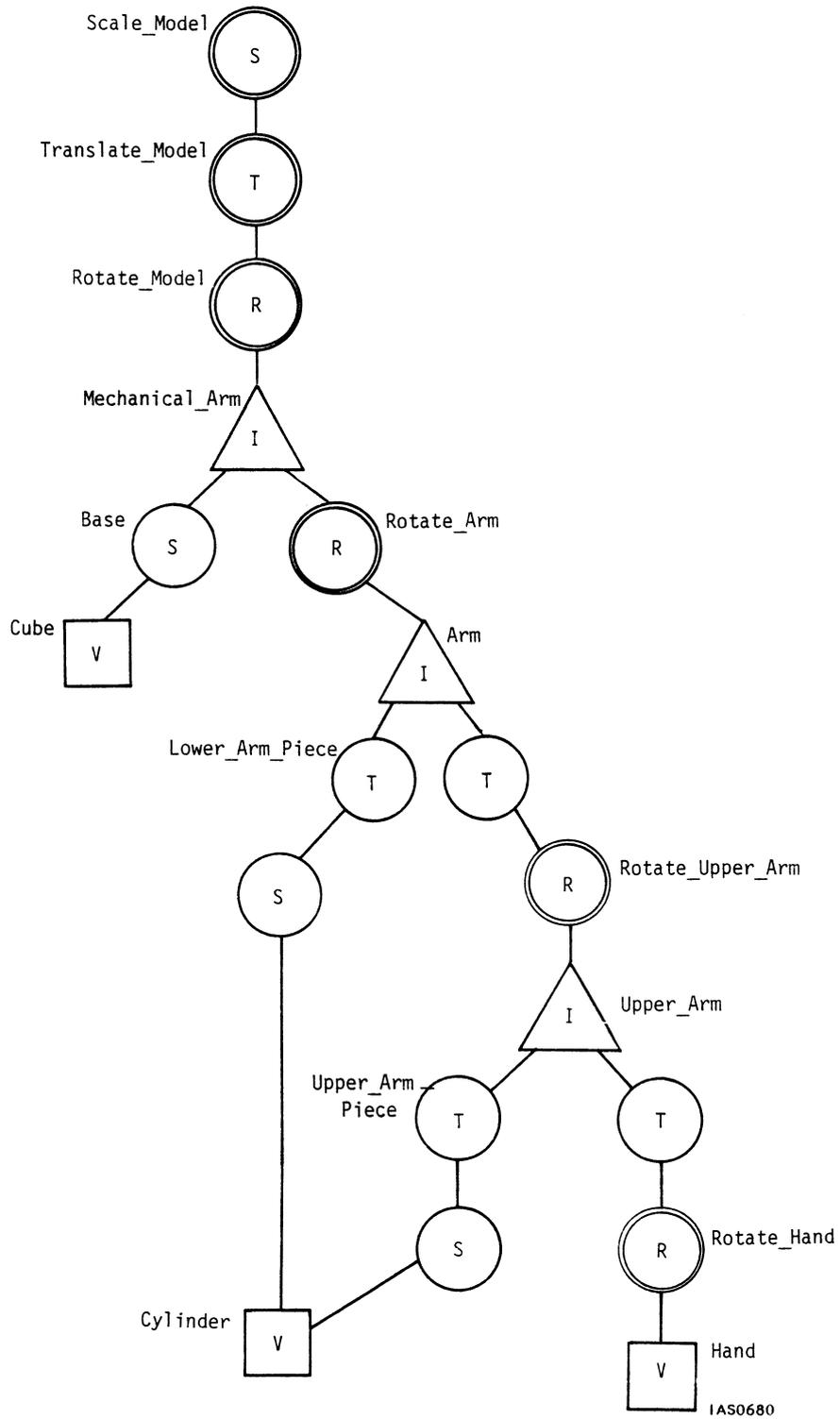


Figure 34. Display Tree for the Mechanical Arm

The display tree shows details of the structure of the model in the PS 300 which the hierarchy of parts in Figure 33 does not. In particular, it includes the primitives, the modeling transformations which create the parts of the model from the primitives, and the interaction points which will provide motion to the whole model and its parts.

Display Tree Terminology

Display trees consist of *nodes* and the *branches* that connect them. A node is an element in the hierarchy. The squares are *data nodes*. These are used to represent the primitives from which individual pieces of the model are built: the cube, the cylinder, and the hand. The triangles are *instance nodes*. These represent the grouping of primitives and modeling transformations into parts: the Upper arm, the Arm, and the complete Mechanical Arm. Circles represent transformations and are called *operation nodes*. Single circles represent the modeling transformations that are applied to primitives to create the pieces and move them into place. Double circles represent *interaction points*. These are the operation nodes in the model which will receive new values from interactive devices such as dials or the data tablet.

Nodes

Nodes are created by PS 300 commands. Commands such as VECTOR_LIST create data nodes. ROTATE, SCALE, and TRANSLATE commands are three of the many which create operation nodes. The INSTANCE command creates instance nodes.

Updating Nodes

Each data and operation node contains information. A rotation node contains a rotation matrix, a vector list node contains point and line information, and so on. An instance node does not contain data in the same way. It acts as a pointer to paths in the display tree and occurs at the head of a hierarchical branch. All operation nodes and most data nodes can have their contents changed in several ways. You can redefine the command that created the node and change its contents that way. You can send a new value to a node using the SEND command. Or you can program an interactive device to send a stream of constantly changing values to a node and so change the model dynamically.

Nodes have inputs to which data can be sent. The number of inputs depends on the type of node. An input will only accept data compatible with its contents. A rotation node, for instance, will only accept a 3X3 matrix; a translation node will only accept a 2D or 3D vector, and so on.

Data Nodes

Data nodes represent primitive objects. Vector lists, polygon lists, curves, and text are all defined as graphical primitives using commands which create data nodes. These nodes always appear at the bottom of a branch. Data nodes have inputs so that their contents can be updated. Figure 35 shows the inputs to a vector list data node.

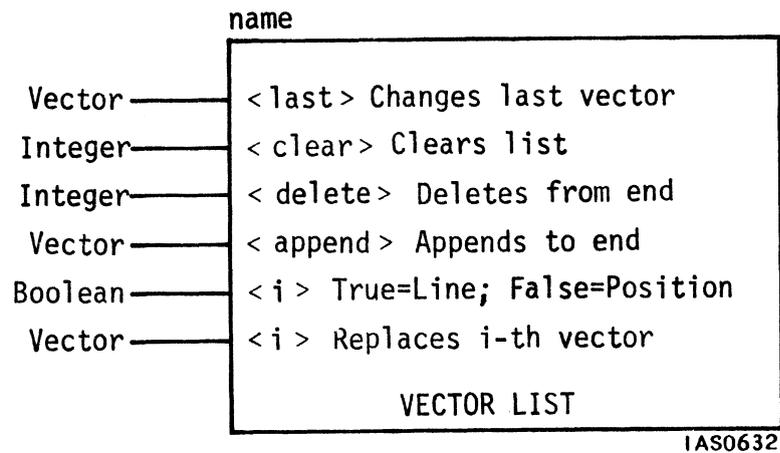


Figure 35. Inputs to a Vector List Node

Most of the inputs to a vector list node are named instead of being numbered. A new vector sent to input <last> is substituted for the last vector in the list. An integer sent to input <clear> removes the vector whose position in the list corresponds to the number sent; for example, sending 4 will remove the fourth vector. An integer sent to input <delete> will delete that many vectors from the end of the vector list. Any vector sent to input <append> is added to the end of the vector list. A Boolean true or false can be sent to a numbered input (shown as input <i>). This will change the identifier of that vector to an L for line or a P for position. A vector sent to any numbered input is substituted for the vector whose position in the list corresponds to the number of the input. By sending new values to this node, you can change the geometry and topology of an object.

Operation Nodes

Operation nodes represent transformations that are applied to objects: translations, rotations, and scales, viewing transformations (discussed in Section 5) and attribute operations (discussed in Section 6). Operation nodes have inputs which will accept data to update a node. Figure 36 shows the input to a rotation node.

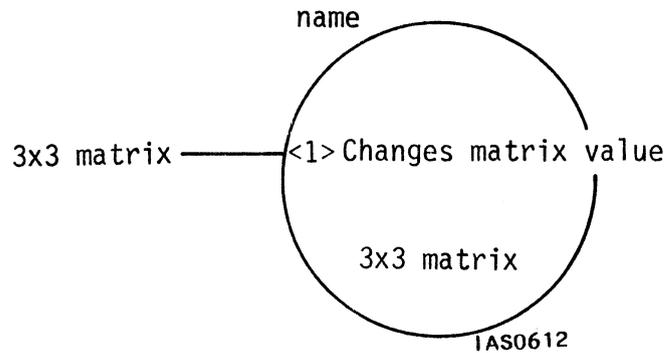


Figure 36. Inputs to a Rotation Node

The rotation node has a single input which accepts a 3X3 matrix which is substituted for the matrix currently contained in the node.

Operation nodes may be created for modeling purposes or for interaction.

Modeling nodes represent transformations used to create the original static model by sizing the pieces and moving or rotating them into place. These nodes are shown as single circles in the display trees. The value contained in a modeling node is not usually updated.

Interaction nodes represent places in the model which will be connected to interactive devices. These are operation nodes whose contents will be updated with data from the devices to which they are connected. Interaction nodes are shown as a double circles in a display tree. Naturally, any node that can be updated has the potential for being an interactive node. But certain nodes are specifically created as interaction points in a model's structure.

In Figure 34, for example, the scale node called Base is used for modeling purposes: it scales the vector list Cube by a fixed amount in X, Y, and Z to create the shape which forms the base of the arm.

The scale node called `Scale_Model`, however, serves a different purpose. It is drawn as a double circle to show that it is an interaction point in the structure. This node will be created with a value of one (scaling by one has no effect on the model at all). Then a dial will be programmed to supply a 3X3 scaling matrix to this node. Each time the dial is turned, a different scaling matrix will be sent to update the node and the model will grow smaller or larger on the screen.

A rotation node designed for interaction is usually created with a value of zero. When the object is displayed, the zero rotation will have no effect on the object's orientation. As rotation matrices are supplied to the node from a dial, the object will rotate. Translation nodes set up for interaction are created with a value of zero in X, Y, and Z. As new vectors are sent to the translation node, the object will move in any of the three directions.

Instance Nodes

Instance nodes group operation nodes and data nodes into larger named entities and set up and maintain *spheres of influence* in the display tree.

Grouping

Instance nodes form what were called *compound objects* in Section 3. They group transformations and primitives into a single named entity. In a display tree for a complex model, instance nodes are often at the "head" of branches which represent the individual parts of the model.

Recall the notation used in Section 3 to show the structure of the object called `Trans_Star1`.

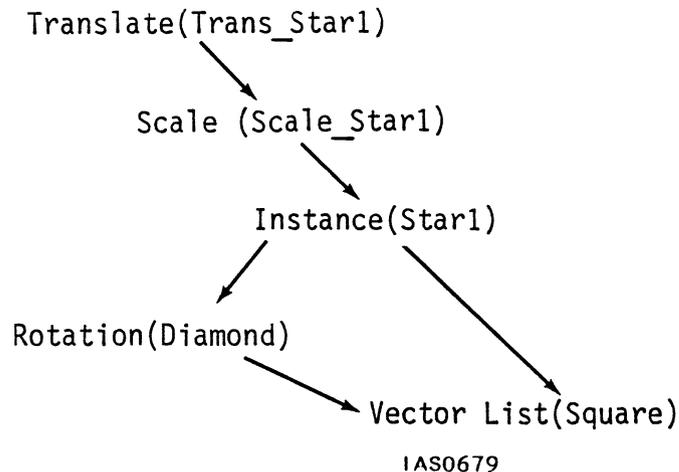


Figure 37. The Structure of `Trans_Star1`

The name `Trans_Star1` is a translation which points to `Scale_Star1`. `Scale_Star1` is a transformation that points to `Star1`. `Star1` groups the untransformed vector list defining the Square with the rotated square that defines the Diamond. Both Diamond and Square share the same primitive definition.

If the structure `Trans_Star1` is now drawn as a display tree, it appears as shown in Figure 38.

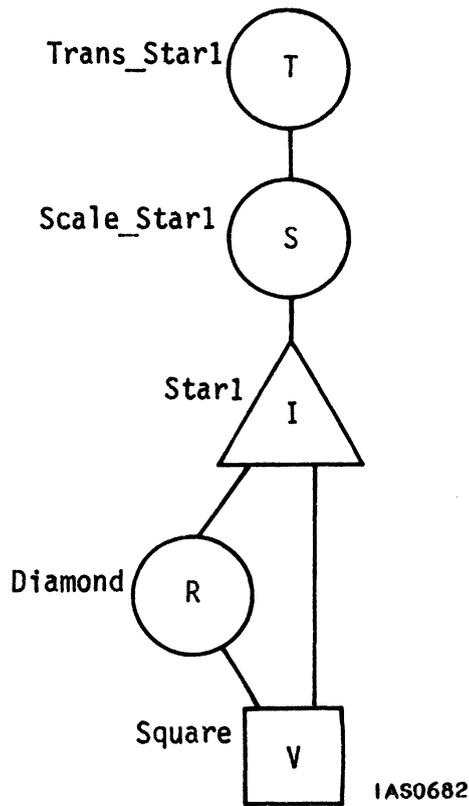


Figure 38. Display Tree for `Trans_Star1`

The single instance node, `Star1`, groups all of the transformations that are applied to the primitive Square under one name.

Because instance nodes perform this grouping function, they have more than one branch out of them. Instance nodes are the only nodes in a display tree which may have more than one branch coming out of them, though data nodes and operation nodes may have more than one branch into them.

Sphere of Influence

In a display tree, nodes higher up in the structure affect nodes lower down. For example, the nodes `Trans_Start1` and `Scale_Start1` at the head of the display tree in Figure 38 affect everything below them. If a new scaling matrix is sent to `Scale_Start1`, the complete model will get bigger or smaller on the screen.

However, a node can only affect its descendants, that is other nodes below it on the same hierarchical branch. Consider a simplified representation of the structure for the upper arm of the mechanical arm model (Figure 39).

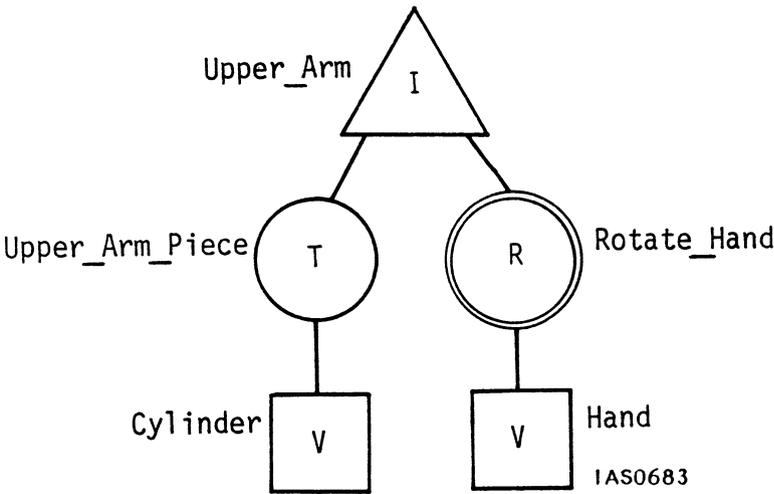


Figure 39. Structure of the Upper Arm

When a dial is connected to the interaction node `Rotate_Hand`, only the hand must move, not the upper-arm piece it is connected to. So the rotation node is placed on a different branch from the `Upper_Arm_Piece` data node to restrict the sphere of influence of the rotation. The rotation will only affect the data node `Hand`.

Instance nodes govern the spheres of influence in a hierarchy. Every branch out of an instance node is affected by operations above the instance node. Operations below the instance node in one branch affect only data in that branch. Instance nodes maintain the integrity of each branch they govern. Consider the following simple tree in Figure 40.

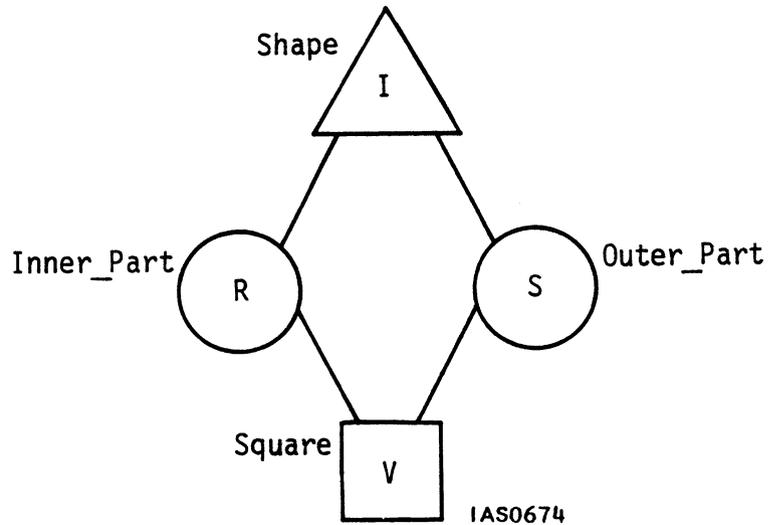


Figure 40. A Simple Display Tree

The tree in Figure 40 represents the structure of the shape in Figure 41.

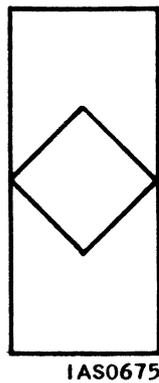


Figure 41. Shape Represented by Display Tree in Figure 40

The shape is created in two parts from a single square primitive. The inner part is the square rotated 45 degrees in Z. The outer part is made by scaling the square non-uniformly in X and Y. The instance node Shape groups the primitive and both transformations into a single compound object.

Both transformations are applied to the same primitive, but they apply independently. The instance node Shape ensures that this occurs. The rotation in the left-most hierarchical branch out of Shape does not affect the scale in the the right branch, and vice versa. Any transformations applied to Shape (that is, above Shape in the display tree) would then affect both branches grouped by the instance node.

SUMMARY

New Information Presented

1. Complex models consisting of separately maneuverable parts are designed as a hierarchy of the model's components.
2. A display tree is a hierarchy which shows the primitives, transformations, and groupings that are used to create the model in the PS 300. Display trees consist of data nodes, operation nodes, and instance nodes, and the branches that connect them.
3. Data nodes and operation nodes can have their contents modified. Certain operation nodes serve as interaction points in the model. They are designed to be updated by values from the interactive devices. In this way, a dial can be connected to a rotation node, for example, to allow the model to be dynamically rotated.

What Next?

The data base now contains all of the "building blocks" for complex models.

- Primitives
- Transformed primitives
- Compound objects
- Complex objects: hierarchical groupings of independent parts of a model, equipped with interaction points

In the next section, you will see how *viewing nodes* are added to the display tree to create different views of the model that has been created.

5. LOOKING AT OBJECTS

When you have created an object as a primitive, a compound object, or a complex model, you will want to get some view of that model on the screen.

In the real world, you can see a different view of an object by moving it. This is simulated in a graphics system by applying modeling transformations (translations and rotations) to the object. An alternative in the real world is for you to move. Leaving the object alone, you can walk around it, and change your viewpoint.

The PS 300, in effect, lets you do the same thing. Using *viewing operations*, you can obtain a number of "natural" views of a model on the screen.

These operations mimic the way you look at objects in real life. You decide your eye point in the coordinate system and the direction you are looking in. You can determine how much of the world coordinate system (and the model) will appear in your view. You can enhance your perception of three dimensions using *perspective* (to make objects further away appear smaller) and *depth-cueing* (to make them dimmer as they recede). In the real world, objects at a distance or outside your range of view disappear naturally. The PS 300 performs *clipping* to eliminate objects or parts of objects that lie outside the screen boundaries.

Once you have determined the particulars of the view (the viewpoint and "window" into the world coordinate system) you can determine where that view will appear on the screen. Areas of the PS 300 screen can be defined as *viewports* in which views of the models will appear.

Viewing operations are defined as part of a model's structure. They are represented as operation nodes in the display tree.

VIEWING OPERATIONS

The modeling transformations discussed earlier let you use primitives as building blocks for the components of a hierarchically structured model, changing their basic shape and moving them into position. Once the model is designed, you need to get a picture of it on the screen. The PS 300 offers a set of *viewing operations* that can be applied to a model to create various views of objects in the world coordinate system.

Displaying an Object

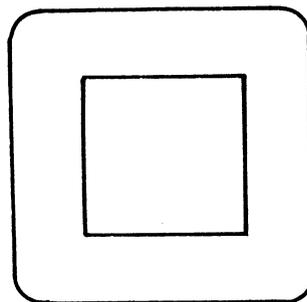
With the PS 300, you can get a picture of a model on the screen by entering a single command. Consider a square with sides one unit long. This shape can be created by entering the following vector list.

```
Square := VECTOR_LIST N = 5 .5,.5 .5,-.5 -.5,-.5 -.5,.5 .5,.5;
```

To display this shape on the screen, it is sufficient to enter

```
DISPLAY Square;
```

The Square shape will appear on the screen as shown in Figure 42.



IAS0686

Figure 42. The Location of the Square on the Screen

The apparent operation of a single command is, in fact, more complicated. The PS 300 does not simply display Square; it displays a view of Square. Before the PS 300 can display this view it needs information about

- A *line of sight*--your vantage point (as viewer) in the world coordinate system and the direction in which you are looking.
- A *viewing area*--what part of the world coordinate system to include in the view.
- A *viewport*--where on the PS 300 screen to display the view.

If you do not specify a line of sight, a viewing area, and a viewport, the PS 300 uses default values. It assumes you are looking from the origin along the positive Z axis. The viewing area extends from -1 to 1 in X and Y and from almost zero to 10^{15} in Z. And the viewport is the full PS 300 screen. These three default values are in effect when you simply display the Square.

ESTABLISHING A LINE OF SIGHT

In the real world, you establish a *line of sight* by standing in some spot, looking towards something, and possibly tilting your head. This gives you a specific view of the object you are looking at. The PS 300 simulates this same ability with a LOOK command. Suppose, for example, the world coordinate system contains a cube with its faces labeled Top, Bottom, Front, Back, Left and Right. The cube is centered around the origin, as shown in Figure 43.

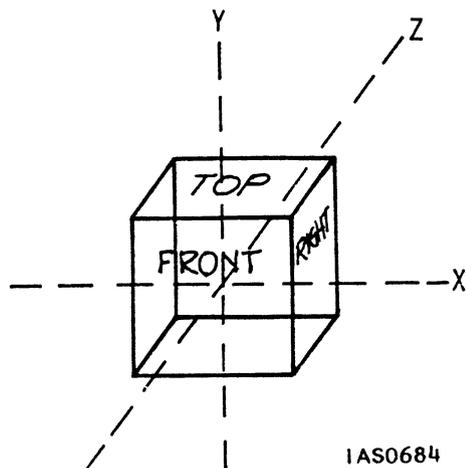
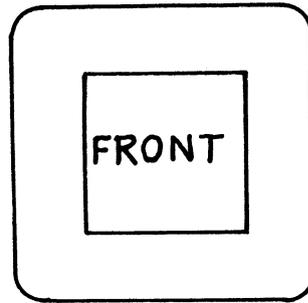


Figure 43. A Cube With Labeled Faces

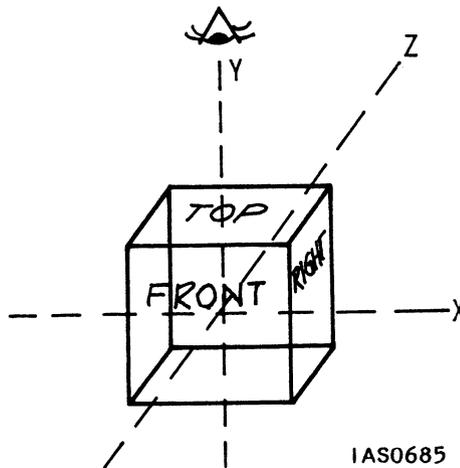
For clarity in the following illustrations, only three labels are shown at a time. If you display the cube without changing the default line of sight, viewing area, or viewport, the screen will show the picture in Figure 44.



IAS0686

Figure 44. Displaying the Cube

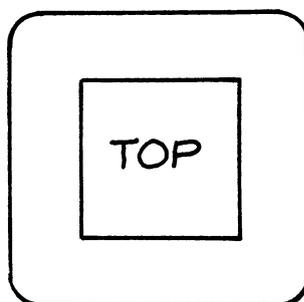
If you want a picture of the top of the cube, you can think of this as moving your eye above the cube and looking down the Y axis at it, as shown in Figure 45.



IAS0685

Figure 45. "Looking Down" the Y Axis at the Cube

The view of the cube which will be displayed is shown in Figure 46.



IAS0687

Figure 46. Looking Down at the Cube: the View on the Screen

A PS 300 command which will create this view of the object is as follows:

```
Top_View := LOOK AT 0,0,0 FROM 0,.5,0 APPLIED TO Cube;
```

An optional UP clause in the command lets you specify what direction is up. This is equivalent to tilting your head left or right.

The concept of "looking at an object" is a very natural way for humans to think. With a graphics system, of course, every visual effect is an illusion. When you look at an object from a location in the world coordinate system, the computer cannot actually move your eye to that location. Instead, it applies transformations to the points and lines that comprise the object and creates a picture of what you would see if you could move your eye.

To get this effect, the LOOK command actually performs the following transformations. First, it translates all points in the coordinate system so that the FROM point is at the origin. It then rotates all points so that you are looking along the positive Z axis towards the AT point. It also rotates points so that the "up" vector is in the positive YZ plane. The ultimate effect of all this is to place your "eye" at the origin and place the object you are looking at in front of your "face" in the positive Z axis.

After you create Top_View with the LOOK command, the world coordinate system and the points and lines defining the cube have been transformed as shown in Figure 47.

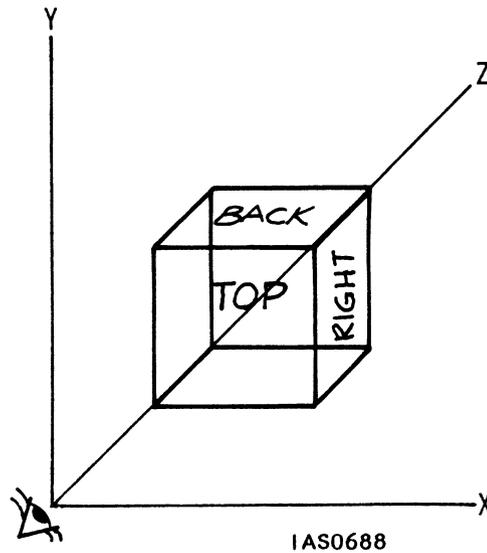


Figure 47. How the LOOK Command Rearranges the Coordinate System

This rearrangement of the world coordinate system is accomplished with a 4X3 transformation matrix, a compound matrix of rotations and translations. The PS 300 uses the information you supply in the LOOK command to create this matrix. It then multiplies all coordinates by this matrix to create the correct "view" of the object for the line of sight you specified.

The tutorial module "Viewing Operations" teaches how to use the LOOK command with all of its options. Mastering this command lets you locate your eye-point anywhere in the world coordinate system, look in any direction, and specify any direction as "up" to create a specific view of an object.

INCLUDING PART OF THE WORLD COORDINATE SYSTEM

In the real world, your view is limited by several factors. If you do not change your position, you cannot see things that are behind you or to either side beyond your field of view. Your view is further limited if you are looking out of a window, or through binoculars or the view finder of a camera. You can only see whatever part of the world is "framed" by the window or the lenses.

With a graphics system, looking at the world coordinate system is much like looking through a view finder. You must specify how much of the world will appear in the view which is displayed on the screen. An area of the world specified for viewing is called a *viewing area* or a *window*. To "see" an object in the world coordinate system, that object must lie in the direction of your line of sight and must be contained within the viewing area you specify.

Viewing Areas in the World Coordinate System

The PS 300 lets you create two types of viewing areas. The WINDOW command creates a viewing area for orthographic or parallel projection views. The FIELD_OF_VIEW and EYE commands create a viewing area for displaying objects as perspective views.

Orthographic Views

A viewing area for orthographic views can be thought of as a box which can be positioned anywhere in world coordinate space but oriented with its sides parallel to the three major coordinate system planes (XY, XZ, and YZ), as shown in Figure 48.

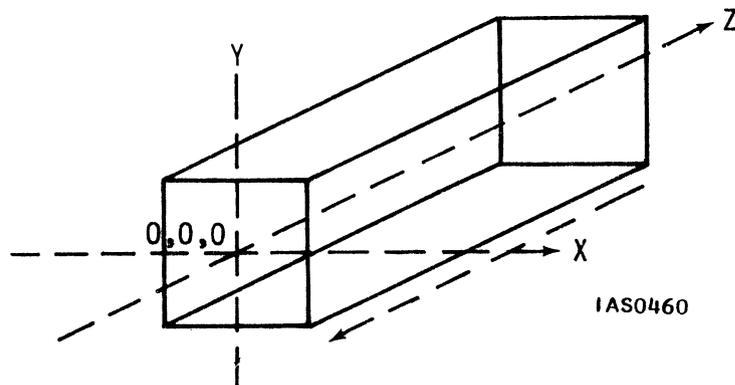


Figure 48. An Orthographic Viewing Area

The viewing area defined by the box has limited X (width), Y (height), and Z (depth) dimensions. In general, if an object lies within the area, it is visible; if it is outside the space, it is not visible. The X and Y boundaries of the viewing space are always in effect. Any object outside those boundaries is never visible. The XY planes at the front and back of the box, however can be enabled or disabled at will. If these planes are disabled, as long as an object lies within the X and Y boundaries, it will be visible no matter where it is located along the positive or negative Z axis. This is shown in Figure 49.

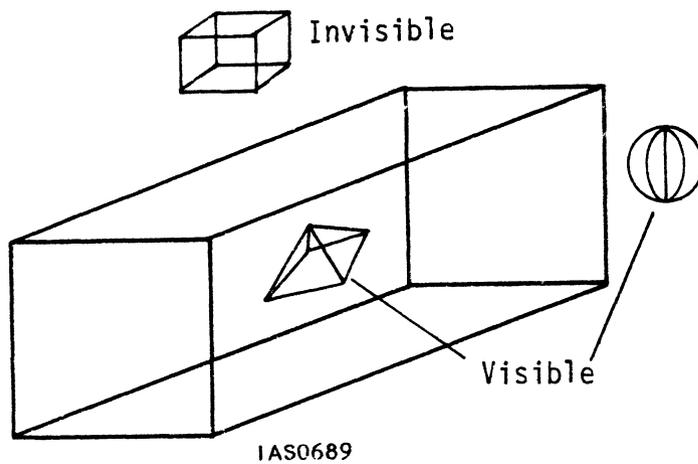


Figure 49. "Visible" and "Non-Visible" Objects

If an object is only partially within the XY bounds of the viewing area, only parts of it are visible. In this case, the computer calculates which lines are visible and *clips* those that are not visible from the view (Figure 50).

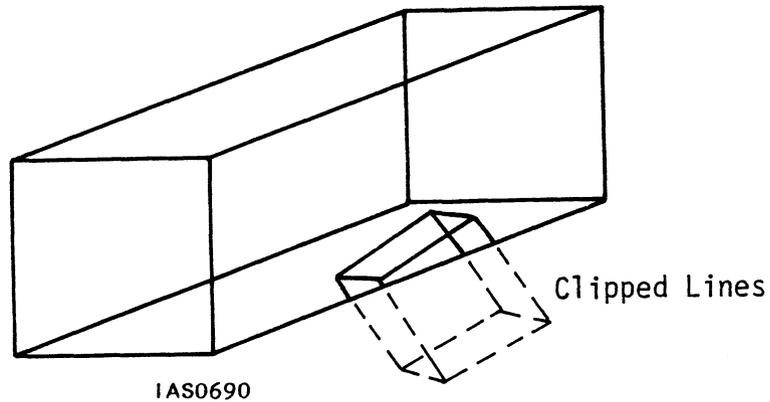


Figure 50. Clipping Parts of an Object

Clipping can also be specified in the Z dimension by enabling the front and back faces of the viewing space which are called *clipping planes*. The front boundary is sometimes called the *hither* plane; the back is called the *yon* plane. Objects or parts of objects that lie outside the front and back boundaries may be clipped from view. This is known as *depth-clipping*, and is illustrated in Figure 51.

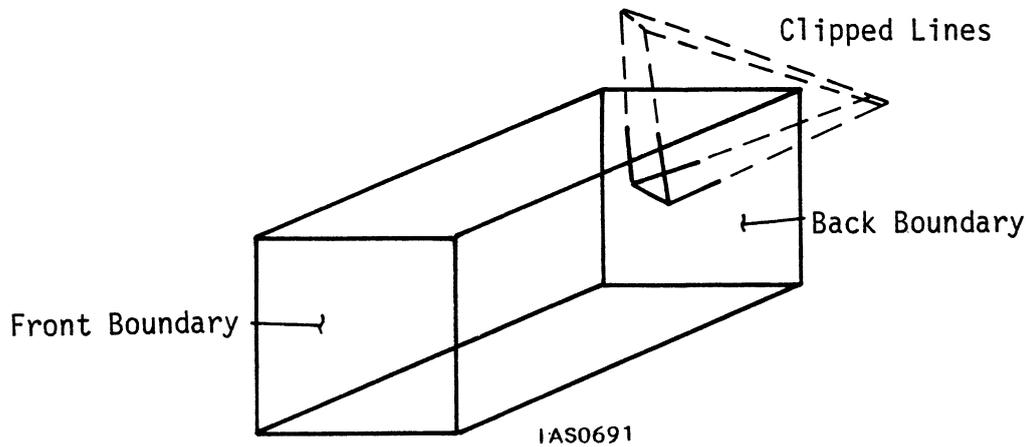


Figure 51. Depth-Clipping of Objects

Objects within an orthographic viewing area are displayed in orthographic or *parallel* projection. This produces a view in which lines that are parallel in the object remain parallel in the view. A rotated cube viewed in orthographic projection, for example, appears as shown in Figure 52.

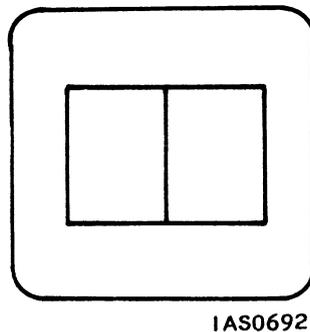


Figure 52. Orthographic View of a Rotated Cube

An object must be enclosed in a viewing space before it can be displayed. If you simply display an object (as with the Square at the beginning of this Section) without explicitly defining a viewing space, the PS 300 defines one for you. The default viewing space imposed by the system is shown in Figure 53.

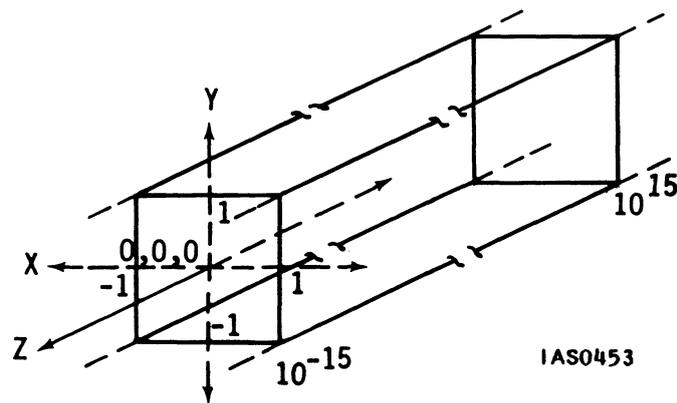


Figure 53. The Default Viewing Space

This is a viewing space for orthographic views only. It extends from -1 to 1 in the X and Y dimensions, and from 10^{-15} (almost zero) to 10^{15} in Z .

With the PS 300, a viewing space for orthographic views is created explicitly with the `WINDOW` command. For example, the command

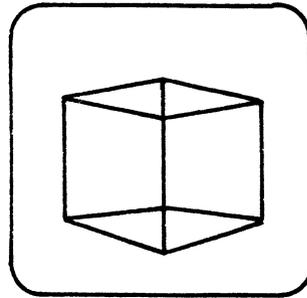
```
New_View := WINDOW X = -2:2, Y = -2:2 APPLIED TO Cube;
```

creates a viewing space twice as high and twice as wide as the default space, but with the same depth. Optional parameters of the command allow you to change Z values by specifying the location of front and back clipping planes. The section called Defining An Orthographic Window in the "Viewing Operations" module explains the `WINDOW` command and its options.

Perspective Views

One way in which the PS 300 creates the illusion of depth on a flat screen is to display objects in perspective.

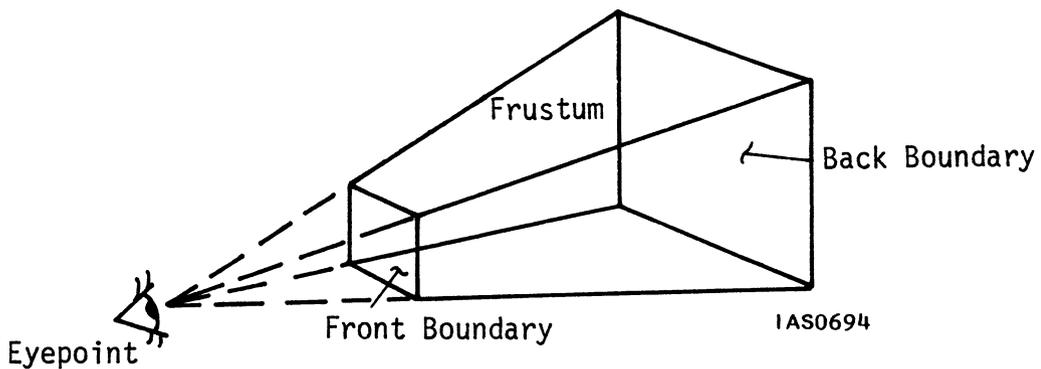
In perspective views, parallel lines that go back from your eye point appear to be converging. A rotated cube viewed in perspective might appear as shown in Figure 54.



IAS0693

Figure 54. Perspective View of a Rotated Cube

A perspective viewing space is a volume shaped like a *frustum*: a section of a pyramid bounded by the front and back clipping planes. If you extend the sides of the pyramid back, the apex of the pyramid is the eye point as defined in the LOOK command (Figure 55).



IAS0694

Figure 55. A Viewing Area for Perspective Views

Two PS 300 commands create perspective views: the FIELD_OF_VIEW command and the EYE command. Both commands are used in conjunction with the LOOK command.

The FIELD_OF_VIEW command (abbreviated to FOV) lets you specify an angle of view from the eye point, which is the FROM point specified in the LOOK command. Optional clauses let you specify the location of front and back boundaries. These determine the depth of the viewing area created with this command. A perspective view is fully defined in conjunction with a LOOK transformation. If no LOOK is specified, the default values are assumed.

The following commands set up a line of sight and a perspective view of an object called Cube using a viewing angle of 30 degrees.

```
Look_Cube := LOOK AT 0,0,0 FROM 0,0,-5 APPLIED TO Cube;  
View_Cube := FIELD_OF_VIEW 30 FRONT = 4.5 BACK = 5.5  
            APPLIED TO Look_Cube;
```

The LOOK transformation will place the center of the Cube at 5 in the positive Z axis, so assuming the cube is one unit square, front and back boundaries of 4.5 and 5.5 should enclose it. When View_Cube is displayed, the screen will show a cube seen in true perspective.

Note that the angle you enter in the FOV command does not alter the severity of the perspective imposed on the object. That is determined by the distance between your eye and the object and depth of the object itself. Instead, the angle lets you see more or less of the world coordinate system. The larger the angle, the larger the portion of the world that will be included in the view.

In a perspective view created using the FOV command, the line of sight established by the LOOK command is always perpendicular to the front and back boundaries of the frustum and passes through their centers. The viewing "pyramid" is always right rectangular. This is shown in Figure 56.

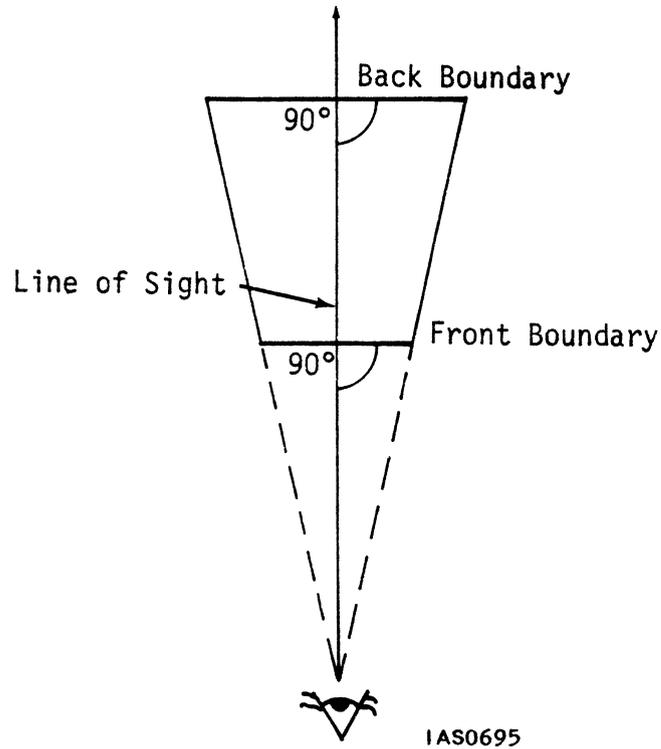


Figure 56. The FIELD_OF_VIEW Viewing Pyramid

The EYE command is used to create a view of an object as it would appear displayed on a screen which is positioned at an angle to your line of sight, not perpendicular to it. This perspective view simulates the "natural" distortion of screen displays that your own eye would see if it were some distance back, up or down, and left or right of the PS 300 screen.

Like the FIELD_OF_VIEW command, the EYE command creates a perspective view of an object. The eye point and the front and back clipping planes specify a pyramid-shaped viewing area. However, if the eye point is offset left, right, up, or down, the pyramid is skewed, unlike the right rectangular pyramid created by the FIELD_OF_VIEW command (Figure 57).

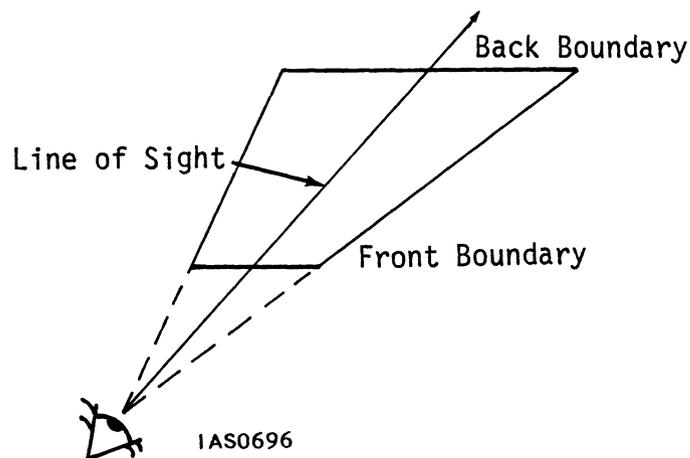


Figure 57. The Viewing Pyramid Created by the EYE Command

The LOOK command must be used first to establish a line of sight on the object to be displayed. Then, clauses in the EYE command let you specify the front and back boundaries of the viewing area in world coordinates and your eye location relative to the center of the screen in relative *room coordinates*. Note the difference between room coordinates and world coordinates. World coordinates are locations in the world coordinate system where models are built and viewed. Room coordinates are locations within the real world (the computer room where the PS 300 lives) and are used to simulate the actual location of your eyes relative to the PS 300 screen. This is a rare instance of when it is permissible to mix the computer's coordinate system and real-world coordinates, since the room coordinate values in the command are used for ratio and proportion operations only.

The following is an example of setting up a viewing area with the EYE command.

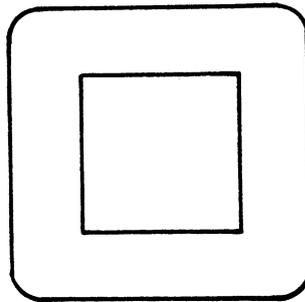
```
Look_Cube := LOOK AT 0,0,0 FROM 0,0,-5 APPLIED TO Cube;
Oblique_View := EYE BACK 20 LEFT 5 UP 12 FROM SCREEN
                AREA 20 WIDE
                FRONT = 4.5 BACK = 5.5 APPLIED TO Cube;
```

In this command, the front and back boundaries are chosen to enclose the cube after the LOOK transformation has taken place. When Oblique_View is displayed, the cube will appear in the correct perspective to simulate an eye position that is back from the screen, over to the left and somewhat high.

The section called Defining Perspective Windows in the "Viewing Operations" modules fully explains the FIELD_OF_VIEW and EYE commands with all of their options.

DISPLAYING AN IMAGE IN SOME AREA OF THE SCREEN

Whenever you instruct the PS 300 to display an object by simply using the DISPLAY command, as long as the object fits within the default window (that is, from 1 to -1 in X and Y and from 10^{-15} to 10^{15} in Z) it will occupy the full screen. For example, a cube defined around the origin with sides 2 units long fits exactly in the default window. When the cube is displayed, an orthographic view will appear which fills the entire display area of the screen, as shown in Figure 58.



IAS0687

Figure 58. Displaying an Object With the Default Window

Specifying a Viewport

When an image is displayed on the screen, the view of the object contained in the viewing area is mapped to a *viewport*. A viewport is an area of the screen with horizontal (X) and vertical (Y) boundaries and an optional intensity range. The intensity range specifies the dimmest and the brightest that lines will be drawn on the screen. Lines at the front clipping plane of the viewing area will be brightest. By default, lines at the back clipping plane will be dimmest. The variation of intensity levels within a viewport creates an effect known as *depth-cueing*.

Perspective views created with the `FIELD_OF_VIEW` or the `EYE` command naturally give the illusion of depth to any object displayed on the screen. This illusion is further enhanced by depth-cueing. When intensity levels have been set for a viewport, the `PS 300` varies the intensity of lines in the view that represent the `Z` dimension (depth) of the object. A line that recedes in the `Z` axis from the eye point gets dimmer as positive `Z` values increase. This gives the impression that objects are being displayed in a place that is brightly lit close to you and more dimly lit farther away. Depth-cueing can be turned on or off. The default is on.

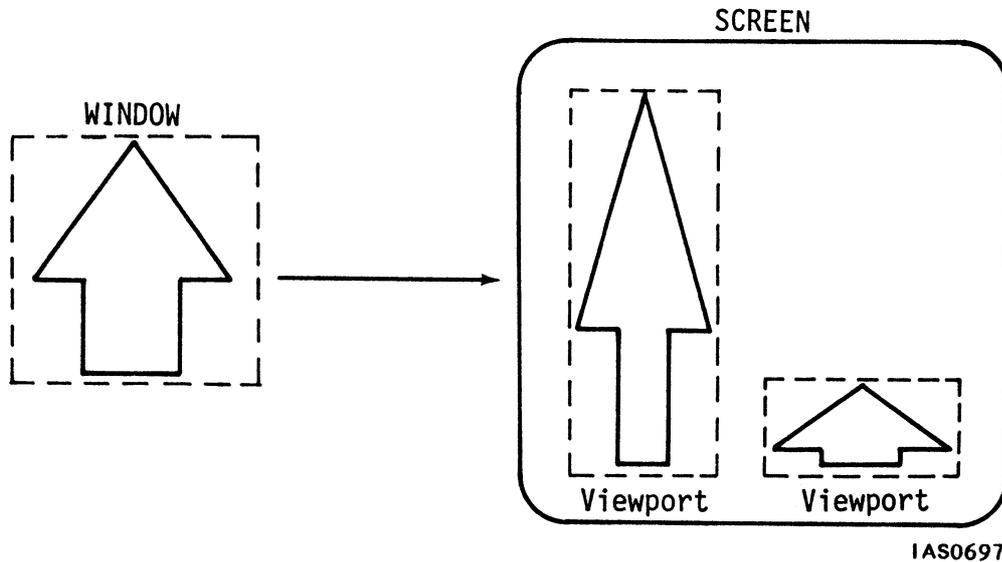
The default viewport to which the `PS 300` maps views of objects in the viewing area is the full screen, and the full intensity range (from 0 to 1) is in effect. The `VIEWPORT` command lets you change the size of the viewport, relocate the viewport anywhere on the screen, and vary the intensity. The following command, for example, creates a viewport in the upper-right quadrant of the screen, and sets an intensity range from .5 to 1.

```
New_Viewport := VIEWPORT HORIZONTAL=0:1 VERTICAL=0:1
                INTENSITY=.5:1 APPLIED TO Cube;
```

When `New_Viewport` is displayed, a cube will appear in the upper-right quadrant of the screen. There will be less contrast between the brightest and the dimmest lines than in the original view of `Cube`.

To obtain an accurate view of an object, the viewport it is displayed in must have the same *aspect ratio* as the viewing area that encloses the object. The aspect ratio is the ratio of height to width. Objects defined in viewing areas with square front and back boundaries and displayed in non-square viewports will appear distorted.

An arrow enclosed in a square viewing area and displayed in non-square viewports, for instance, may look like this (Figure 59).



IAS0697

Figure 59. Distorted Views of the Arrow

Distortion also occurs when non-square viewing areas are displayed in square viewports. The `FIELD_OF_VIEW` and `EYE` commands always create viewing areas with an aspect ratio of 1:1, a square. The `WINDOW` command can be used to create a viewing area with an aspect ratio that is not 1:1, a non-square viewing area.

Any size and any number of viewports may be displayed at the same time. In this way the screen can be used to show multiple views of the same object or different views of different objects.

Note that viewport operations are the only viewing operations which are not matrix transformations of graphical data. When the contents of a viewing area are mapped to a viewport, this is a ratio and proportion operation, not a transformation of coordinates in the world coordinate system.

VIEWING TRANSFORMATIONS AND DISPLAY TREES

When views of objects are created, viewing operation nodes are added to the display tree.

Consider, for example, the group of objects shown in Figure 60.

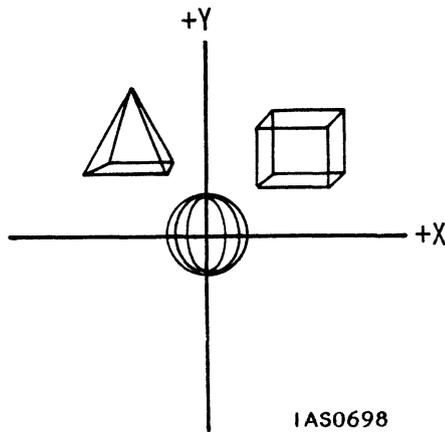


Figure 60. A Group of Objects in the Coordinate System

The group consists of three primitives: a sphere centered around the origin, and a cube and pyramid translated off the origin. These primitives have been grouped as an instance called Shapes. The display tree for Shapes is shown in Figure 61.

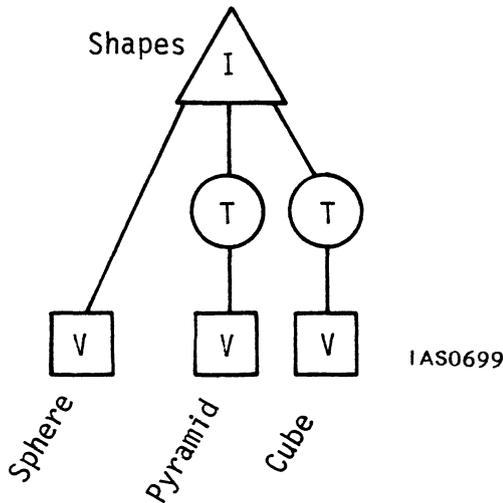


Figure 61. Display Tree for Shapes

If you use the DISPLAY command to view Shapes, the picture on the screen will be as shown in Figure 62.

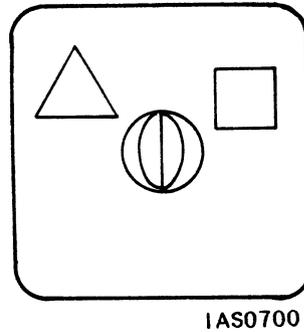


Figure 62. DISPLAYing Shapes

The default viewing space is a window for orthographic projection, the default LOOK is in effect, and the default viewport is the full screen. To get any other view of Shapes on the screen, you must explicitly use the viewing commands.

To establish a different line of sight, for instance, use the LOOK command as follows. Look towards the origin from a position that is left (negative X), up (positive Y), and back (negative Z) from the origin.

```
View_Shapes := LOOK AT 0,0,0 FROM -1,1,-5 APPLIED TO Shapes;
```

The LOOK command adds the following node to the display tree (Figure 63).

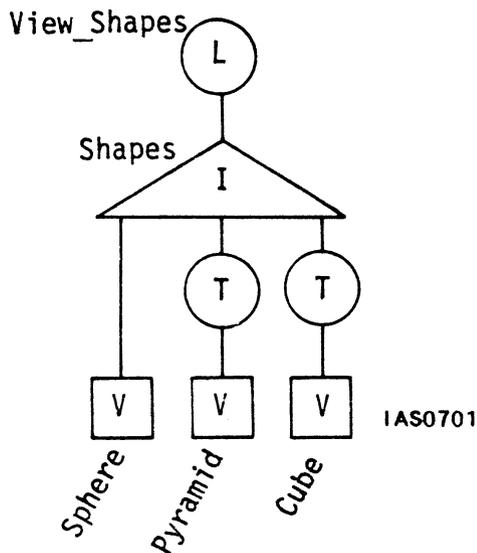


Figure 63. Adding the LOOK Node

Now build a viewing area around Shapes so that the objects can be seen in perspective projection. First calculate where the LOOK command has actually placed the objects in the coordinate system. Remember that all coordinates are translated and rotated so that the FROM point is at the origin and the AT point is in the positive Z axis. The new location of an object in Z is found by taking the square root of the following equation.

$$(X_a - X_f)^2 + (Y_a - Y_f)^2 + (Z_a - Z_f)^2$$

In this equation, "a" is the AT point in X, Y, and Z and "f" is the FROM point.

In a LOOK command with a FROM point of 0,0,0 and an AT of -1,1,-5, the new location in Z of the sphere (the one object exactly at the origin) is the square root of 27, or 5.1962. This is shown in Figure 64.

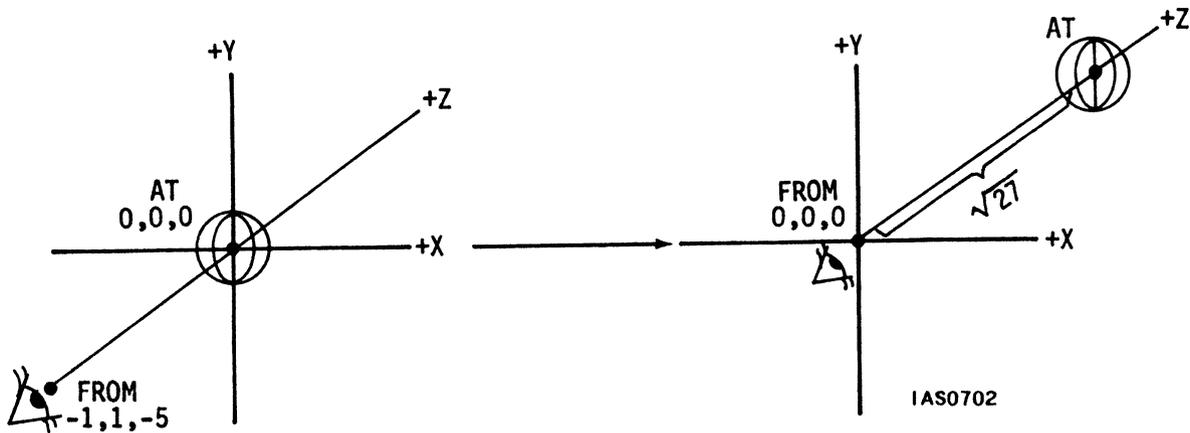


Figure 64. The LOOK Transformation

For maximum depth-cueing of the objects, the front and back boundaries of the perspective viewing area should be close to the objects. The sphere is a primitive with a radius of .15, so the front boundary should be placed at 5.1962 - 0.15, which is 5.0462. The back boundary can be placed further back at about 6. This is shown in Figure 65.

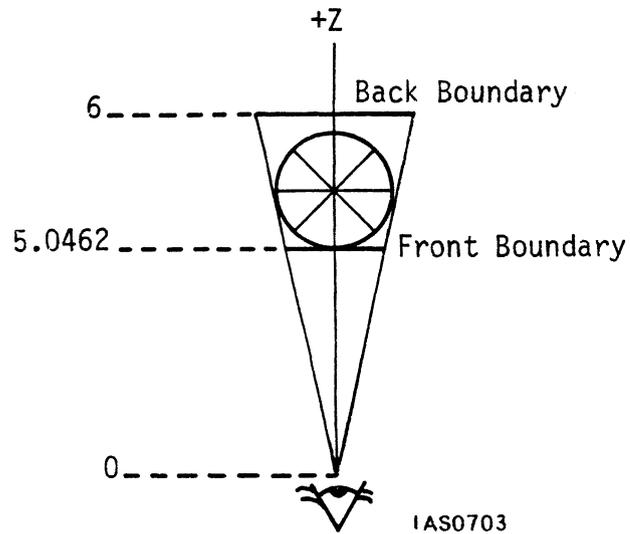


Figure 65. Calculating the Front and Back Boundaries

Finally a viewing angle must be chosen. An angle of about 28° should suffice. The command to create the perspective view, then, is as follows.

```
Perspective_View := FIELD_OF_VIEW 28 FRONT = 5.0462 BACK = 6
                    APPLIED TO View_Shapes;
```

This adds a viewing matrix node to the display. The new structure is shown below (Figure 66).

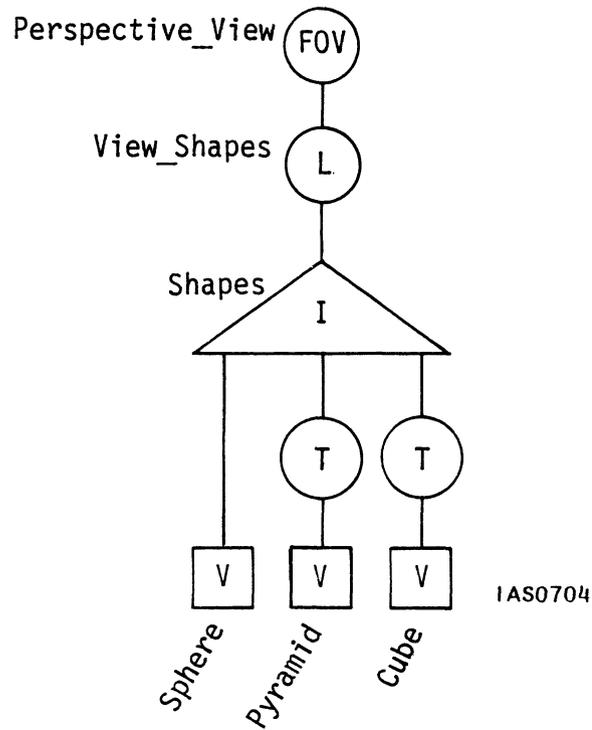


Figure 66. Adding the FOV Node

Next, create a viewport in the upper-right corner of the screen. This is where the view of Shapes will be displayed. Do not use the optional intensity clause, so Shapes will be displayed with the full intensity in effect for maximum depth-cueing.

```

Final_View := VIEWPORT HORIZONTAL=0:1 VERTICAL=0:1
              APPLIED TO Perspective_View;
  
```

The display tree for the final view is shown in Figure 67.

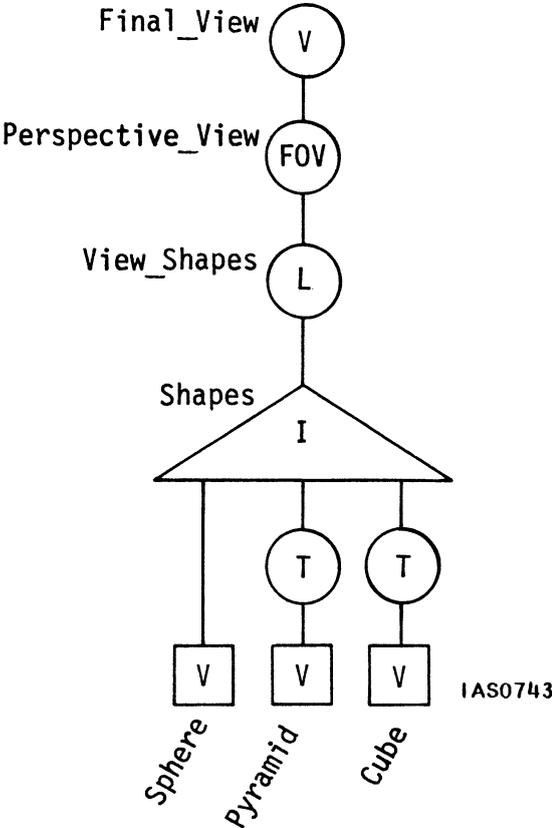


Figure 67. Adding the VIEWPORT Node

When Final_View is displayed, the PS 300 screen will appear as shown in Figure 68.

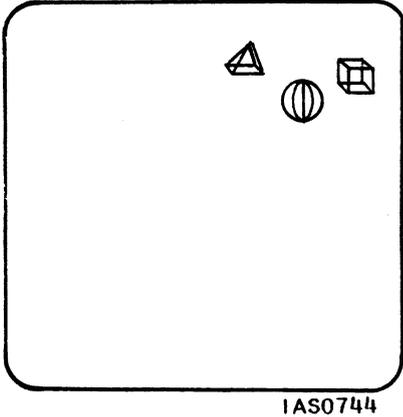


Figure 68. The Final Display

SUMMARY

New Information Presented

1. Viewing operations are matrix and non-matrix operations that let you create a variety of views of objects and display those views anywhere on the PS 300 screen.
2. A complete "view" is created by establishing a line of sight, defining a viewing area in the world coordinate system, and defining a viewport on the PS 300 screen. The PS 300 assumes default values for all three if they are not explicitly specified.
3. A line of sight is a matrix operation which specifies a point to look from and a direction to look at. You can also specify which direction is up. Whatever values you assign to these variables, the PS 300 translates coordinates so that the "look from" point is at the origin and the "look at" point is somewhere in the positive Z axis. It also rotates all coordinates so that "up" is in the YZ plane.
4. Viewing areas result from matrix transformations which produce orthographic or perspective views of objects. For an object to be visible, it must be enclosed in a viewing area. Objects or parts of objects that lie outside the viewing area are clipped, and do not appear in the view displayed on the screen.
5. A viewport is the area of the screen in which the contents of a viewing area are displayed. Viewports are not matrix operations. Any number of viewports and any sized viewports can be displayed at the same time. A difference between the aspect ratio (height to width) of the viewing area and the aspect ratio of the viewport will result in a distorted view of the object.
6. Viewing transformations add operation nodes to the display tree for an object.

What Next?

The data base now contains display trees that represent many different views of the basic models that have been created. By displaying these views, any number of images can be displayed on any part of the PS 300 screen.

In the next section, you will see how *attributes* can be assigned to the objects you create.

6. USING ATTRIBUTES

Modeling operations let you create objects of any complexity with the PS 300. Using viewing operations, you can create an infinite number of different views of the objects and display them anywhere on the screen.

Another set of operations add a further range of possibilities to the images that are displayed. They let you assign *attributes* to an object to enhance its usefulness in modeling and analysis applications.

ATTRIBUTES

All modeling and viewing operations (other than viewports) transform the coordinates of objects in the world coordinate system to create new objects. Each transformation adds an operation node which applies matrix operations to the object definitions.

The PS 300 lets you add other operation nodes to a display tree which do not transform graphical data and so do not create transformation matrices. These nodes assign *attributes* to an object.

Attributes offer a variety of possibilities for changing the characteristics of a displayed image. These include:

- Determining aspects of the image such as color, intensity, and the character font in which text appears.
- Referencing objects or parts of objects for display only when certain conditions are met.
- Marking parts of the displayed image as capable of being "picked" with a stylus, puck, or other pointing device.

Attribute settings are different from transformations because they are not matrix operations. Attribute nodes set and change values which are stored in registers. These registers record the *current state of the machine*. When the PS 300's Display Processor encounters an attribute node in a display tree, the contents of the node are used to check and sometimes to change the register representing that attribute. For example, an attribute node which sets depth-clipping can enable or disable depth-clipping, depending on the Boolean value contained in the node.

There are three classes of attributes: *appearance attributes*, *structure attributes*, and *picking attributes*.

APPEARANCE ATTRIBUTES

Appearance attributes govern the following aspects of an object when it is displayed.

- The colors of lines that form the image.
- The intensity at which lines are drawn.
- Whether or not depth-clipping is performed on the image.
- The character font for any text in the image.

Displaying Objects in Color

If you have the optional Color Shadow Mask (CSM) Calligraphic Display, you can display objects or parts of objects in different colors. Color is specified as a *hue* and a *saturation*. The hue is the color itself. There are 360 hues to choose from. These correspond to values on a color wheel as shown in Figure 69.

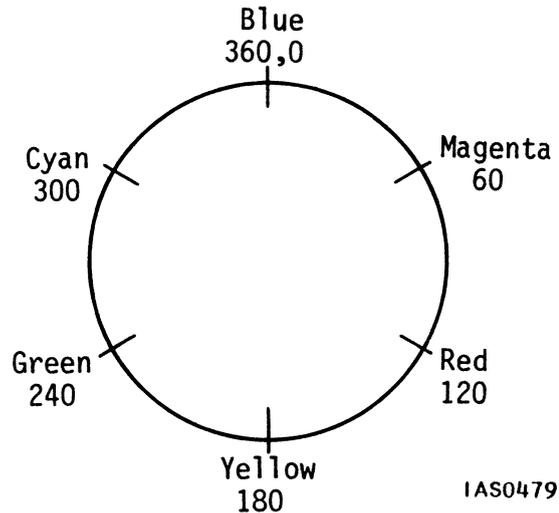


Figure 69. The Color Wheel

Blue has a value of 0 and 360, red is 120, and green is 240. The saturation is the amount of color versus the amount of white in the hue, and is specified as a range from 1 to 0. Blue at high saturation is deep-toned. At low saturation, it is sky blue, and at 0 saturation it is white.

There are two options for displaying images in color. One is to make all the vectors that compose an object the same color. The other is to "color blend" the individual vectors of an object.

Displaying All Vectors in the Same Color

Color is applied to an object using the SET COLOR command. A cube can be colored red by applying the following command to Cube.

```
Red_Cube := SET COLOR 120,1 APPLIED TO Cube;
```

When Red_Cube is displayed on the CSM, the lines that form the cube will appear in full-bodied red on the screen.

With complex objects, different parts of the object can be displayed in different colors. Each SET COLOR command creates an operation node in the display tree for the object. Consider the Mechanical Arm discussed in Section 4. A simplified display tree is as shown in Figure 70.

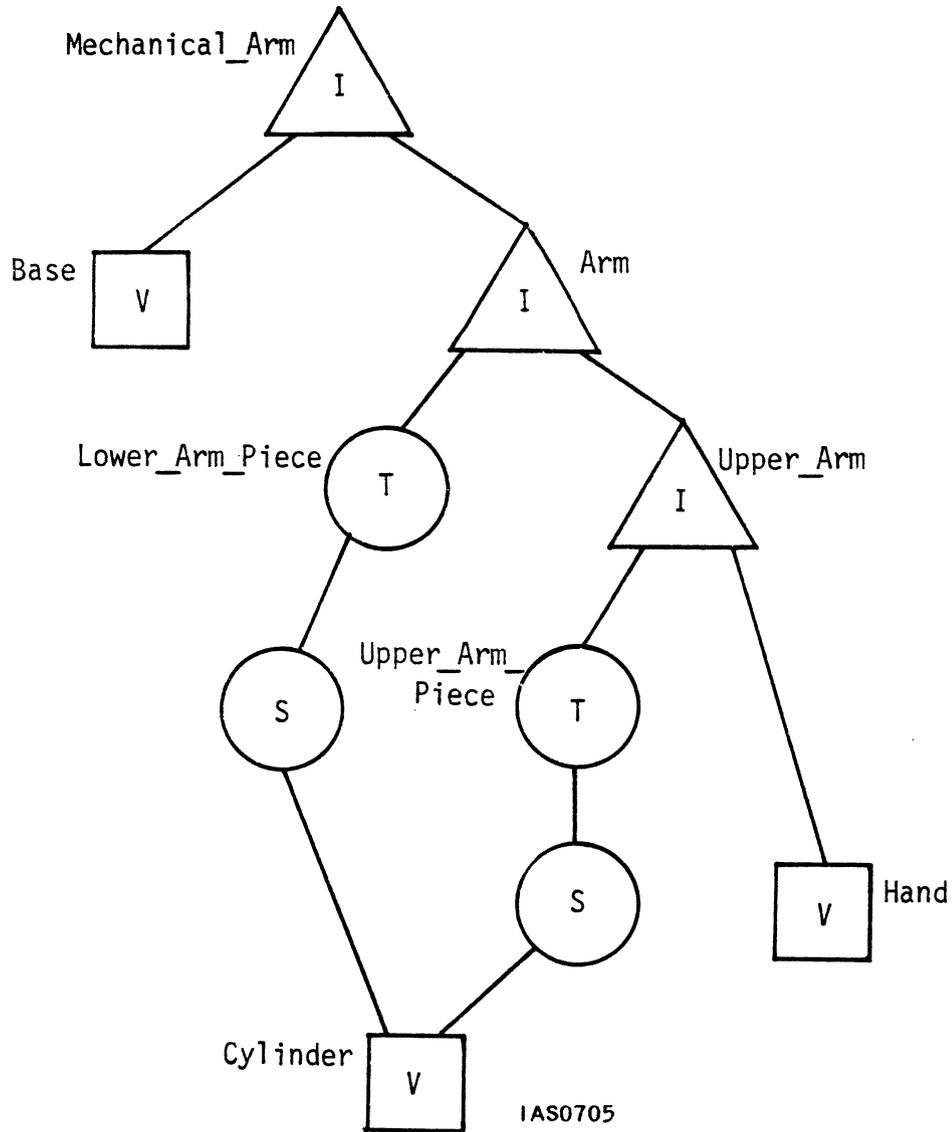


Figure 70. A Simplified Display Tree for the Mechanical Arm

You can use the SET COLOR command to color the parts of the model separately. For example, the base can be colored red, the arm pieces blue, and the hand green. The display tree with the SET COLOR nodes added is as shown in Figure 71.

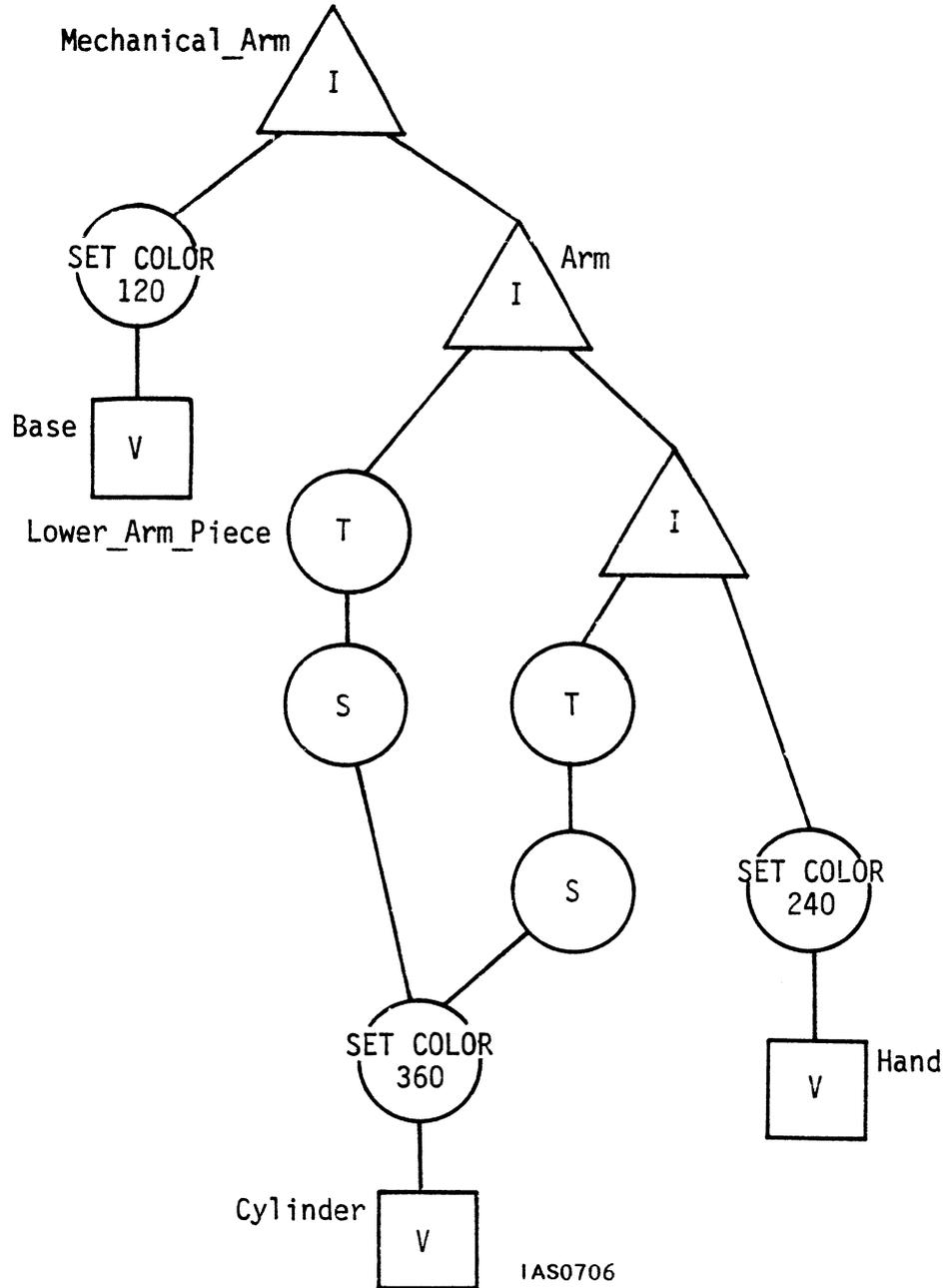


Figure 71. Display Tree With Color Nodes

For more information on color nodes, refer to the section called Setting Color in the "Viewing Operations" module.

Color Blending Vectors

The other way to use color with the PS 300 is to *color blend* each of the vectors in a vector list. This form of coloring is known as color blending because if two adjacent vectors are of different hues, the hue of the line segment between them is blended continuously between the endpoints.

Two commands are used to color-blend vectors: the SET COLOR BLENDING command and the VECTOR_LIST command.

For example, to display a line that has red and green endpoints and shades of red and green between them, a vector list is created for the line using the optional COLOR clause and a hue value (H) for each endpoint. The following command will create the line.

```
Line := VECTOR_LIST COLOR N = 2 0,0,0 H=120, 1,1,0 H=240;
```

To initiate color blending of the vector, the SET COLOR BLENDING command is used. This command has a clause which lets you determine the saturation of the color displayed. A 1 represents full saturation for deep-toned colors; a 0 represents no saturation, that is, white. The following command initiates full-saturation color blending for the vector list named Line.

```
Colored_Line := SET COLOR BLENDING 1 APPLIED TO Line;
```

When Colored_Line is displayed, assuming that the default LOOK, WINDOW, and VIEWPORT are in effect and that the contrast has been set to 0, a diagonal line will be displayed from the center of the screen to the upper-right corner of the screen. The line will be red at the center of the screen and will gradually blend colors along its length to green at the other endpoint.

For more information on color blending by vector, refer to the SET COLOR BLENDING and the VECTOR_LIST commands in the *Command Summary* in Volume 3A.

Setting and Changing Intensity Levels

The PS 300 can be programmed to vary the intensity at which line segments are drawn between endpoints. This ability is used to good effect in the process known as *depth-cueing*. Depth-cueing enhances the illusion of three-dimensional views by varying the intensity of any line that recedes in the positive Z axis. Lines in an image which are "farther away" from the viewer appear dimmer.

Intensity levels are associated with viewports. An option of the VIEWPORT command allows you to specify the intensity variation for lines drawn within the viewport. A minimum and a maximum intensity are specified as values from 0 to 1. When objects enclosed in an orthographic or perspective window are mapped to the viewport, lines closest to the front boundary of the window are drawn at maximum intensity, and lines closest to the back boundary are drawn at minimum intensity.

The PS 300 also has a SET INTENSITY command which allows intensity to be specified as a separate attribute of an object. The command creates an attribute operation node in a display tree which overrides the intensity specification of a VIEWPORT command.

A SET INTENSITY node in a display tree is often used as an interactive node. The node has two inputs. One accepts a Boolean value to enable or disable the effect of the node. The other accepts a 2D vector to change the intensity range. Thus a SET INTENSITY node in a display tree can be used to interactively change the intensity setting of a displayed image. The following command creates a node named Change_Intensity.

```
Change_Intensity := SET INTENSITY OFF 0.0:0.5 APPLIED TO Car;
```

The display tree which contains this node might be structured as in Figure 72.

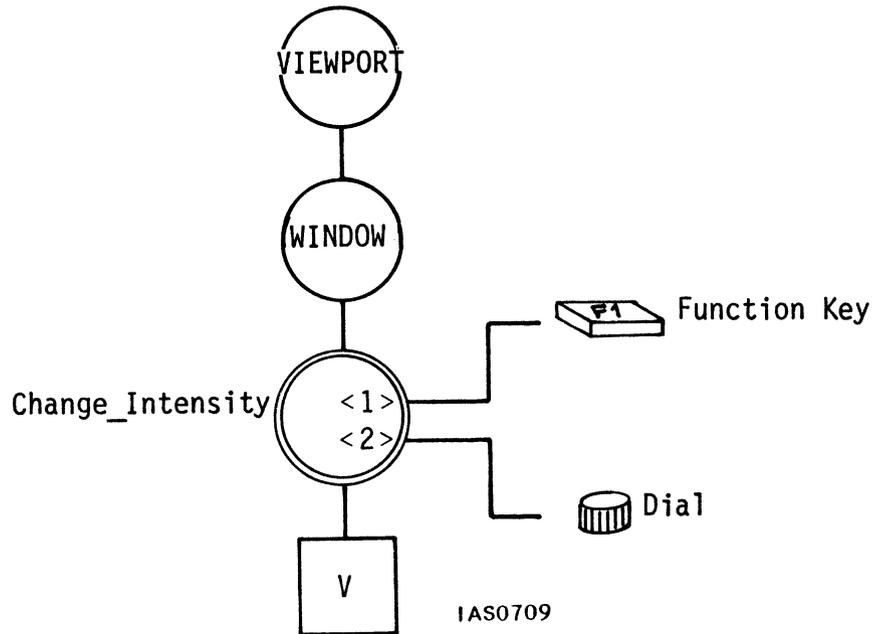


Figure 72. An Interactive Intensity Node

Function networks connect the two inputs of the node to interactive devices. Until the SET INTENSITY node is enabled, the intensity setting of the viewport (the default setting of 0:1) is in effect. A Boolean TRUE sent to input <1> from a Function Key will enable the SET INTENSITY node. New intensity settings can then be supplied from a dial, so that the operator can interactively change the intensity setting while viewing an image.

For more information on setting intensity, refer to the section called Setting Intensity in the "Viewing Operations" module.

Enabling and Disabling Depth-Clipping

Depth-clipping is the operation of clipping (removing from the screen) objects or parts of objects that extend outside the viewing area in Z. The PS 300 automatically clips objects or parts of objects which extend beyond the X and Y boundaries of a window. Depth-clipping (or Z-clipping) is an optional feature which is not in effect when the system is initialized. It is specified as an attribute of an object.

Orthographic and perspective windows are defined with front and back boundaries or *Z-clipping planes*. When depth-clipping is enabled, only objects or parts of objects that lie within the area bounded by the *Z-clipping planes* will be displayed in the viewport. This is illustrated in Figure 73.

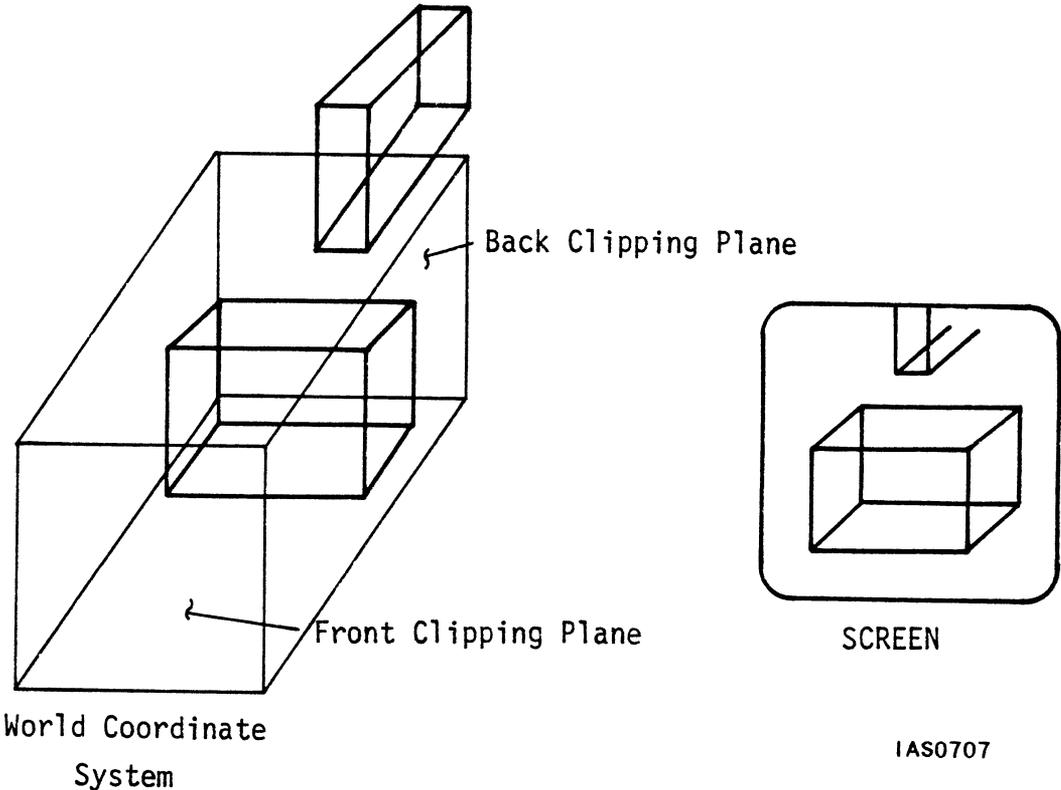


Figure 73. Depth-Clipping Enabled for a Viewing Area

When depth-clipping is disabled, objects that lie outside the *Z-clipping planes* in the positive or negative *Z* axis will be visible. Consider the objects in Figure 74.

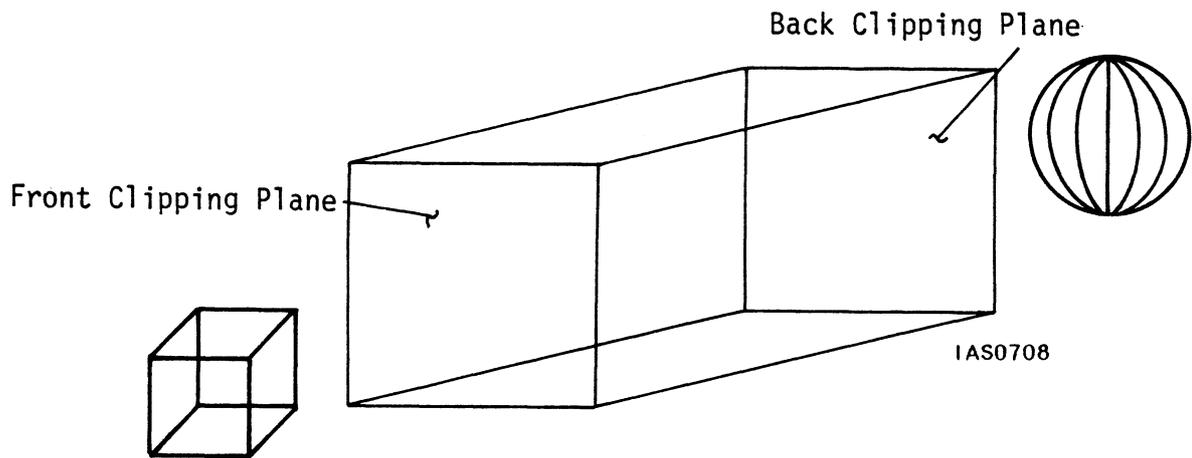


Figure 74. Objects Outside the Front and Back Boundaries

The cube and the sphere will not be displayed if depth-clipping is on, because they lie outside the front and back boundaries. When depth-clipping is turned off, however, they will be displayed.

Objects in front of the front boundary, such as the cube, will be displayed at maximum intensity. Objects behind the back boundary, such as the sphere, will be displayed at minimum intensity.

The following command enables depth-clipping for an object called Rotated_Car.

```
Z_Clip := SET DEPTH_CLIPPING ON APPLIED TO Rotated_Car;
```

A display tree into which the SET DEPTH_CLIPPING node is inserted is shown in Figure 75.

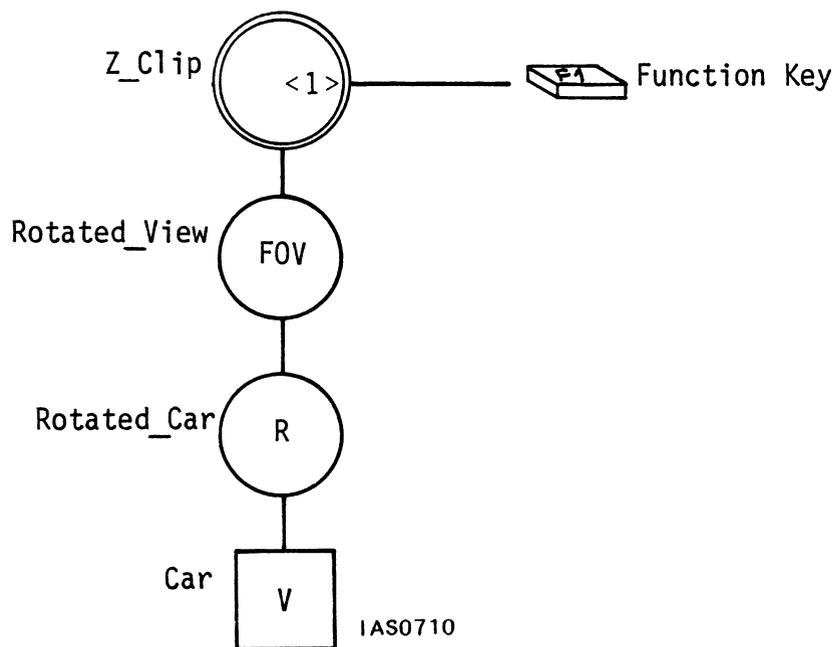


Figure 75. Display Tree With Depth-Clipping Node

The node can be turned on or off interactively. It has one input which accepts a Boolean true or false. TRUE turns depth-clipping on; FALSE turns it off. A Function Key can be connected to the node to toggle depth-clipping on and off.

Displaying Images on Different Screens

Another optional attribute of an object is whether or not it will appear on a particular screen when it is displayed. Users of a PS 300 system with more than one display station can channel an image to any of the screens by adding SET DISPLAY nodes to the display tree of the object.

Initially, when an object is displayed it will appear on all the screens in the configuration. For example, an installation that has one workstation and two display stations has three displays: 0, 1, and 2. When the programmer creates an object at the workstation and enters a DISPLAY command, the image will appear on all three screens if the SET DISPLAY command has not been used. The command can be used to channel the image to selected screens only.

The following command stops an object called Cotter_Pin from being displayed on displays 1 and 2.

```
Single_Screen := SET DISPLAY 1,2 OFF APPLIED TO Cotter_Pin;
```

When Single_Screen is displayed, Cotter_Pin will appear on the workstation screen (display 0) but not on the display station screens (displays 1 and 2). An alternate version of the command has a parameter "ALL" which refers to any displays lower in the same branch of the display tree that are set using the SET DISPLAY command. This can be used to reset all the displays in a branch to the same state.

For more information on selectively displaying objects, refer to the section called Setting Displays ON and OFF in the "Viewing Operations" module.

Choosing a Character Font for Text

If text forms part of an object as a label, a menu item, or annotation, for example, you can add attribute nodes in the display tree to allow different character fonts to be used.

The PS 300 has a standard character font in which all text appears. You also have the ability to create alternate fonts. There is a command which allows you to design any number of other character fonts. This command (BEGIN_FONT END_FONT;) lets you enclose up to 128 separate vector lists defining characters within a named structure. Each vector list defines a letter, character, or number in the character font. For more information on the BEGIN_FONT ... END_FONT; command, consult the *Command Summary* in Volume 3A and the "Text Modeling and String Handling" module. Also, in Volume 4 there is a user's guide to MAKEFONT, a graphical character font editor program. This program allows you to create new character fonts, to combine fonts, and to change existing fonts.

Once an alternate font has been created, it can be used by setting an attribute node in the display tree. Suppose that the alternate fonts Italic and Modern have been created, and that character strings in a display tree are to be displayed in the standard font and in Italic and Modern. Consider the display tree in Figure 76 for a group of labelled objects.

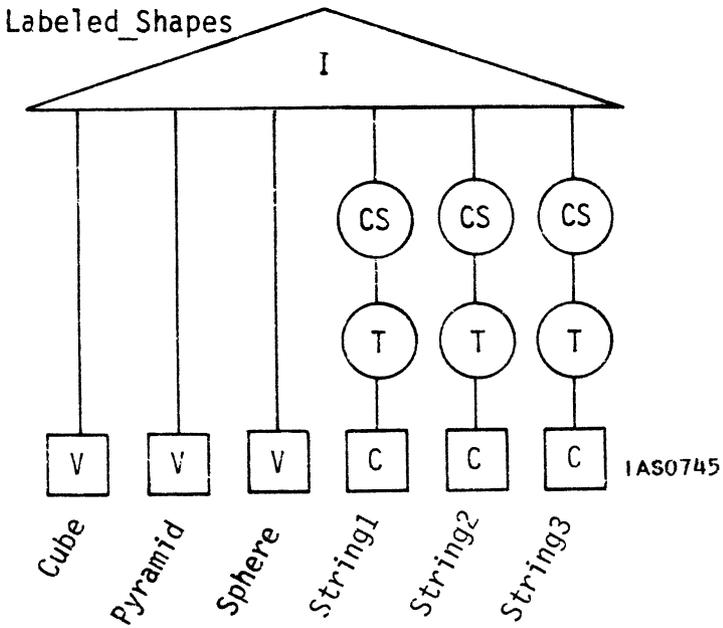


Figure 76. Display Tree for a Group of Labeled Objects

The instance node `Labeled_Shapes` groups vector lists for three objects (a cube, a sphere, and a pyramid) and character strings ("Cube", "Sphere", and "Pyramid") to label the objects. Each character node is created by the `CHARACTERS` command. The character nodes are preceded by a `CHARACTER SCALE` and a `TRANSLATE` node to scale the characters and move them to their correct location. When `Labeled_Shapes` is displayed, the three objects will appear labeled in the standard font.

Suppose you want the word "Cube" to appear in the *Italic* font and "Sphere" to appear in *Modern*. Two `CHARACTER FONT` nodes must be inserted above the data nodes for the cube and sphere labels. The following commands create those nodes.

```
Cube_Label := CHARACTER FONT Italic APPLIED TO Cube;
Sphere_Label := CHARACTER FONT Modern APPLIED TO Sphere;
```

The modified display tree with alternate fonts specified is structured as shown in Figure 77.

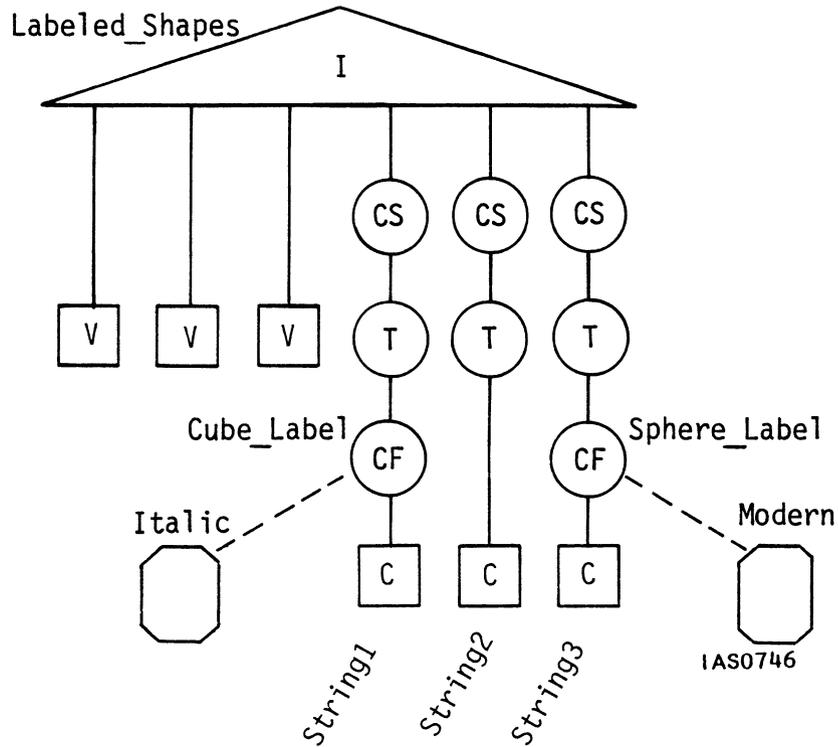


Figure 77. Display Tree With Character Font Nodes

The CHARACTER FONT nodes called Cube_Label and Sphere_Label are pointers to the fonts called Italic and Modern. The branch of the tree which ends at the CHARACTERS node for labeling the pyramid has no CHARACTER FONT node in it, so the string "Pyramid" will appear in the standard font.

STRUCTURE ATTRIBUTES

A display tree for an object is composed of branches which determine the paths that the Display Processor must take when the object is being displayed. Each branch is unconditionally traversed during each display processing cycle. *Structure attributes* create nodes in a display tree at which "branching" may occur only if certain conditions are met. These attributes allow you to:

- Reference objects or parts of objects by setting conditional bits and testing those bit settings further down the display tree.
- Add or remove detail from an object by setting Level-Of-Detail bits and testing for them further down in the display tree.
- Control blinking or alternate displaying of images by setting a rate and an on/off phase, then testing for the phase further down the display tree.

Conditional Referencing

Conditional referencing is generally used to display or blank parts of a complex structure by selectively traversing or bypassing branches of a display tree. Two commands are needed to set up and use conditional referencing. The SET CONDITIONAL BIT command sets any of fifteen conditional bits numbered 0 to 14. This creates a SET CONDITIONAL BIT node, or SET node for short, in the display tree. Below the SET node, an IF CONDITIONAL BIT node, or IF node, is created. This node tests a conditional bit setting and branches to the name it is APPLIED TO if the condition is met.

For example, consider a display tree for a car which, for simplicity, consists of four wheels, a chassis, and a body, as shown in Figure 78.

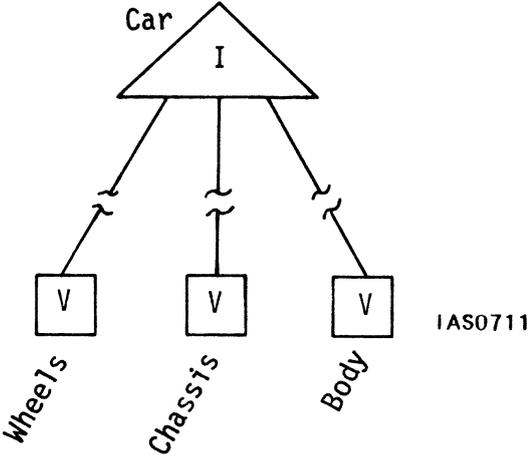


Figure 78. Simplified Display Tree for a Car

For some reason, you want to be able to display or blank the car body at your whim. Use the SET command and IF command to create a pair of SET and IF nodes in the branch which ends with Body.

```
Set_Condition := SET CONDITIONAL_BIT 1 ON THEN Condition_Met;
Condition_Met := IF CONDITIONAL_BIT 1 IS ON THEN Body;
```

Notice that the THEN form of the command is used. This is synonymous in all cases with the APPLIED TO form of the command, but makes more syntactic sense to readers.

Figure 79 shows the display tree with conditional referencing nodes added.

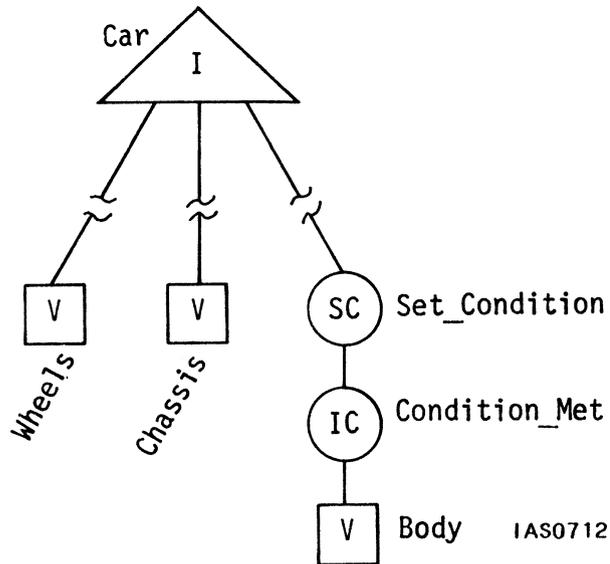


Figure 79. Display Tree With Conditional Referencing Nodes

Initially, when Set_Condition is displayed, all the components of the car, including Body, will be displayed. The condition that bit 1 be set on is met and the path to the data node Body is made. The ON/OFF clause lets you control the display of the car body. A Function Key can be connected to Set_Condition to turn it ON (Boolean TRUE) or OFF (Boolean FALSE). When the bit is off, the car body will not be displayed.

Refer to the section Using Conditional-Bit Attribute Settings in the "Conditional Referencing" module for more examples of this sort.

Level-Of-Detail

Level-of-detail is another form of conditional referencing that is built into a display tree using pairs of SET and IF nodes. This form of conditional referencing is normally used to unfold detail in a complex display. For example, a display for a geological or seismological application might show various levels in the earth's crust. SET and IF level-of-detail nodes can be placed in the display tree to allow the picture to be displayed or blanked layer by layer.

Unlike conditional-bit referencing where 15 bits may be set, level-of-detail uses only one variable. This is an integer from 0 to 32767. The SET LEVEL_OF_DETAIL command creates a SET node in the display tree. The IF LEVEL_OF_DETAIL command creates an IF node to test the level-of-detail setting and complete the path to a named entity accordingly.

Consider as an example a display tree for a three-dimensional contour map of an area of land. You want to be able to turn a dial and add contour lines in 50 foot increments from sea level to 250 feet. Before any level-of-detail nodes are added, the display tree is simply a collection of vector lists, one for each contour line, under a single instance node, as shown in Figure 80.

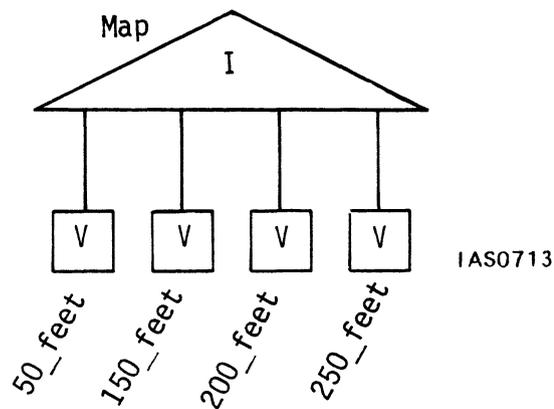


Figure 80. Display Tree for a Contour Map

This structure was created by the following command which grouped the vector lists.

```
Map := INSTANCE OF 50_Feet, 150_Feet, 200_Feet, 250_Feet;
```

Begin allowing for level-of-detail displays by adding a SET node at the top of the display tree.

```
Set_Level := SET LEVEL_OF_DETAIL TO 1 THEN Map;
```

Each branch of the display tree out of the instance node Map can now be prefixed by an IF node. Unlike conditional-bit referencing IF nodes, LEVEL-OF-DETAIL nodes do not test an on/off state, but a relationship. These relationships are as follows.

Less Than	<
Less Than or Equal To	<=
Equal To	=
Not Equal To	<>
Greater Than or Equal To	>=
Greater Than	>

A different value can be assigned to the IF node for each contour line in the map. If the level-of-detail is 1 or greater, the fifty-foot contour is displayed. If it is 2 or greater, the hundred-foot contour is displayed, and so on. For example, the following command creates a node called If_1 which tests whether or not the level-of-detail is 1 or greater and completes the path to the 50-foot contour line.

```
If_1 := IF LEVEL_OF_DETAIL >= 1 THEN 50_feet;
```

The complete tree with all IF nodes is as shown in Figure 81.

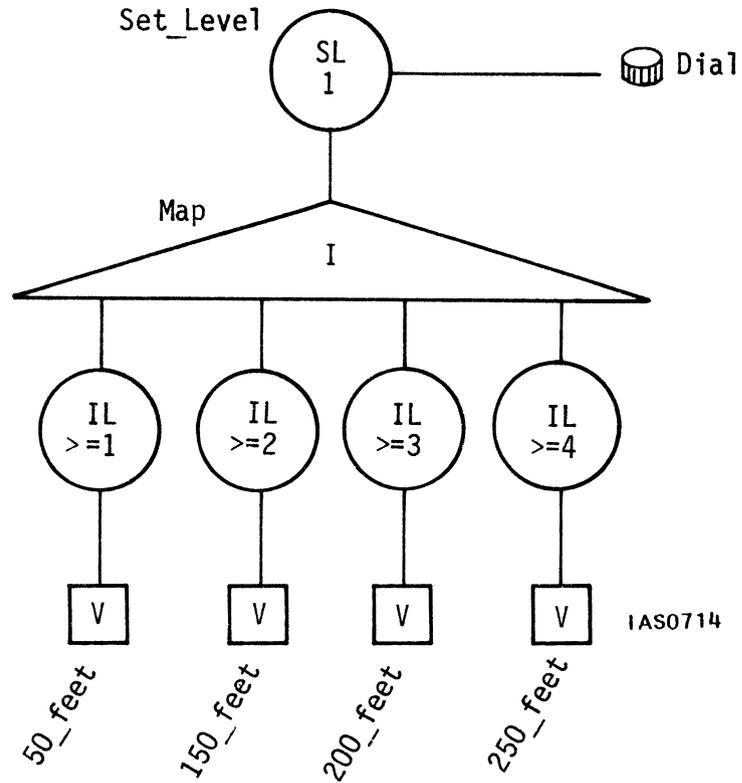


Figure 81. Display Tree With Level-Of-Detail Nodes

A function network can be connected to the SET node to supply new values to the level-of-detail setting from a dial. As the dial is turned and the level-of-detail changes, more of the contours will be displayed.

For more examples of this sort, refer to the section on Using Level_Of_Detail in the "Conditional Referencing" module.

Blinking or Alternating Displays

Making an object blink or alternating the display of different objects is another form of conditional referencing which involves SET nodes and IF nodes in the display tree. The SET node sets a rate for displaying and blanking the object. This rate can be under control of the refresh rate of the PS 300 display, an internal PS 300 clock, or an external clock generated by a function network or the host computer. The IF node determines what will be displayed during the on phase and what will be displayed during the off phase. The commands are SET RATE and SET RATE EXTERNAL (for an external clock), and IF PHASE.

The SET RATE commands specify durations for the on phase and the off phase, an optional initial state (either on or off), and an optional clause called the delay, which specifies the number of refresh frames in the initial state. The IF PHASE command determines what will be displayed during the on phase and what will be displayed during the off phase using the APPLIED TO or THEN clause to indicate a path to a named structure.

For example, to cause the label associated with an object to blink by being displayed for 120 refresh frames and blanked for 60, the following commands can be used.

```
Blink_Rate := SET RATE 120 60 THEN Phase;
Phase := IF PHASE ON THEN Object_Label;
Object_Label := CHARACTERS 'THIS IS THE OBJECT YOU CHOSE';
```

These nodes would be placed in a display tree as shown in Figure 82.

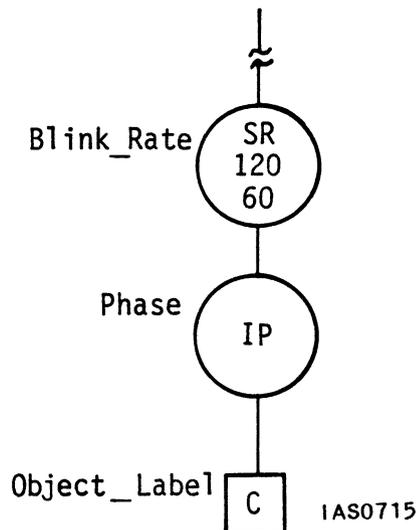


Figure 82. Conditional Nodes for Blinking

The words *THIS IS THE OBJECT YOU CHOSE* will be displayed for 120 refresh cycles (about two seconds) and blanked for 60 (about one second) when this tree is traversed.

The SET RATE and IF PHASE commands can also be used to display alternately two different objects. A display tree can be created with SET and IF nodes to display one object during the on phase and another during the off phase. Figure 83 shows such a display tree.

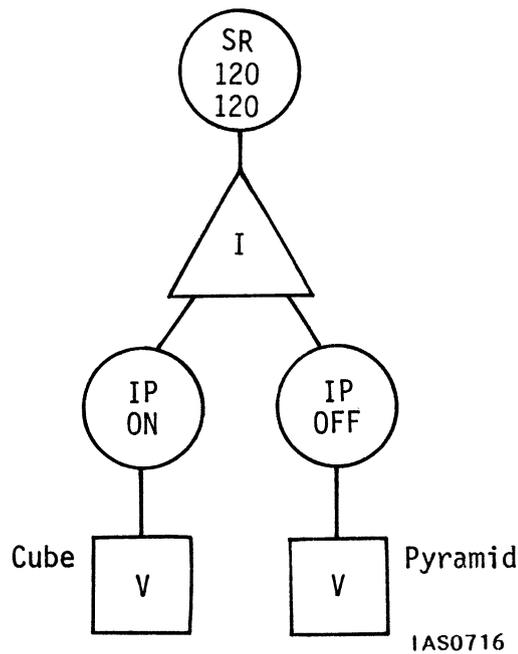


Figure 83. Display Tree for Alternate Display of Two Objects

During the on phase, the cube will be displayed for two seconds. During the off phase, the pyramid will be displayed for two seconds.

Refer to the "Conditional Referencing" module for more information on blinking.

PICKING ATTRIBUTES

In computer graphics terms, *picking* means selecting by means of a stylus, a cursor, or some other pointing device, a line, set of lines, or piece of text in a display. When the pick occurs, the computer generates information in the form of a *pick list* which identifies the line(s) or text picked no matter how the object may be oriented on the screen. This information is reported for programming purposes. For example, in the "Tutorial Demonstration Package", when a menu item is picked, the information returned by the pick is used to run the correct Demonstration Program.

Picking attributes must be assigned to an object before it or any part of it can be picked from a screen display. These attributes are nodes in the display which:

- Mark objects or parts of objects as candidates for picking, and turn picking on or off.
- Assign a name (*pick identifier*) which will be reported as a text string when a pick occurs.

The highest attribute node in the display tree must be the node that turns picking on and off for the object. For example, to make an object called `Space_Shuttle` capable of being picked from the screen, the following command can be used.

```
Pick := SET PICKING ON APPLIED TO Space_Shuttle;
```

Assuming that `Space_Shuttle` is an instance node grouping the various parts of the craft, the top level of the display tree will be structured as shown in Figure 84.

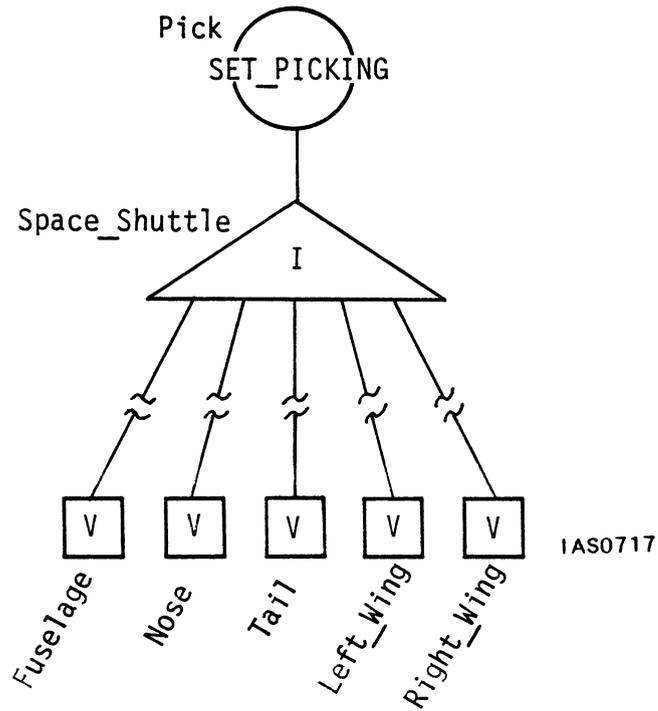
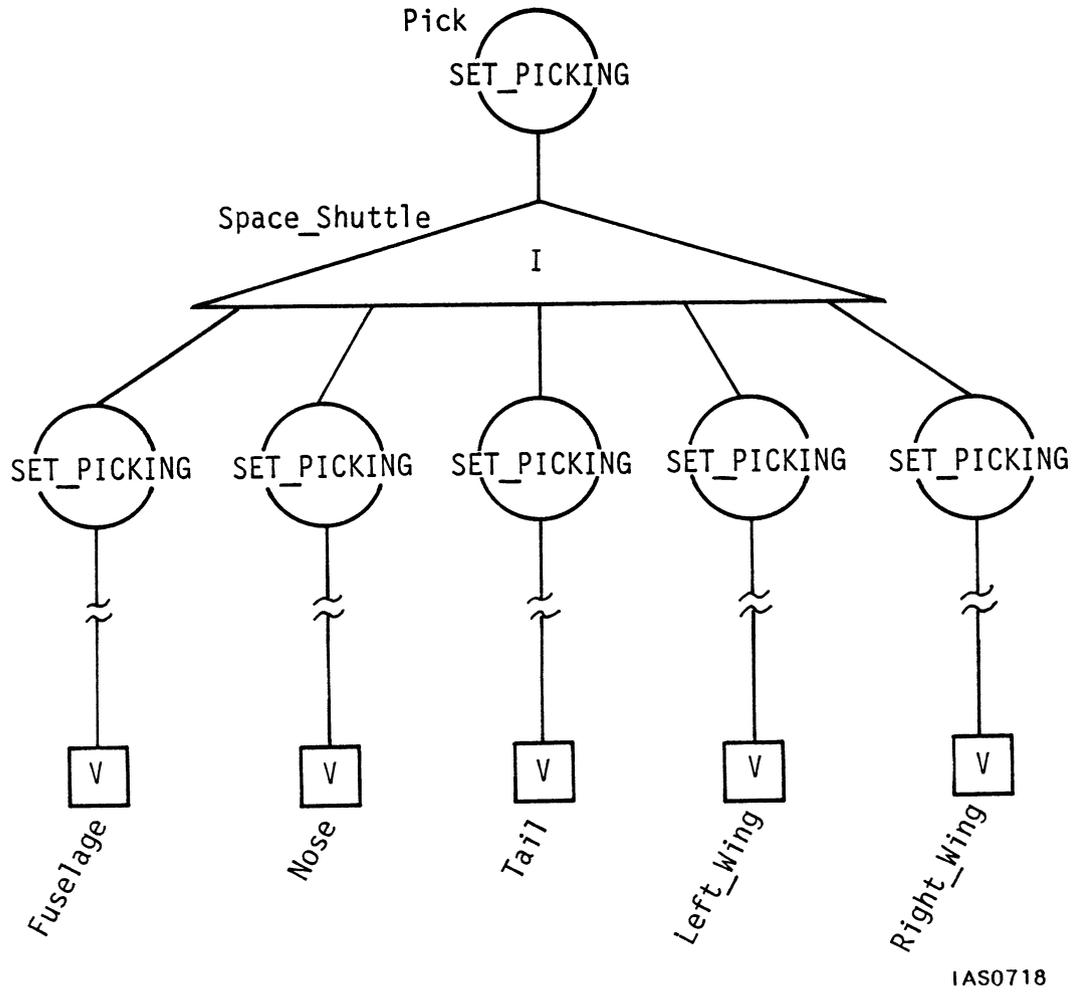


Figure 84. The SET PICKING ON/OFF Node

This node can be used interactively and should be created in the OFF setting. Picking is enabled by a Boolean TRUE, sent to the node through a Function Network.

The node created in the command above makes the whole object called Space_Shuttle capable of being picked. If you want the separate components of the object to be pickable, nodes must be included in the display tree as shown in Figure 85.



IAS0718

Figure 85. Making the Components Pickable

Now the fuselage, nose, tail, left wing, and right wing can be made individually pickable.

The other attribute node that must be added to the display tree assigns the pick identifier (or *pick ID*) that will be reported in the pick list when a pick occurs. Two names identify a picked object. The first is the pick ID--a character string assigned by the SET PICKING IDENTIFIER command. The second name is the name of the data node that contains the line or character that was picked from the screen.

The following command, for instance, assigns a pick ID to the fuselage.

```
Fuselage_Pick := SET PICKING IDENTIFIER = Shuttle_Fuselage  
                APPLIED TO Fuselage;
```

If any line in the fuselage section of the Space Shuttle is picked when picking is enabled, the system will generate a pick list which reports the pick ID as "Shuttle_Fuselage" and the data node as "Fuselage."

To use picking with the PS 300, function networks must be built to report any picks that occur. Refer to the "Picking" module for complete information on setting up picking networks.

SUMMARY

New Information Presented

1. Attribute nodes are another type of operation node in a display tree. They allow you to specify characteristics of the displayed image of the models you create.
2. There are three types of attributes: appearance attributes, structure attributes, and picking attributes.
3. Attribute nodes differ from transformation nodes in a display tree. Transformation nodes create transformation matrices which are applied to the geometrical data in the data nodes. Attributes, however, are non-matrix operations. They set and change values in registers in the PS 300.

What Next?

You have now seen all of the types of nodes that can be included in a display tree. Data nodes define primitive shapes. Modeling operation nodes shape and position parts of complex models in the world coordinate system. Instance nodes group separate primitives and transformations into larger named entities. Viewing operation nodes create views of objects from any angle and from any perspective and specify areas of the screen in which the view will be displayed. Attribute operation nodes change aspects of the model's appearance, allow conditional referencing, and set up picking.

In the next section, you will see how the PS 300's interactive devices are programmed to allow interactive manipulation of models. Function networks are created to complete the path between the devices and interaction nodes in the display tree. These networks take values from the Control Dials, Function Keys, and so on, and convert them to the correct type of data for the interactive node they are connected to.

7. INTERACTING WITH THE PICTURE

A display tree contains three types of operation nodes. Modeling nodes represent translation, rotation, and scale transformations that are applied to primitive data to shape and position the parts of a model in the world coordinate system. Viewing nodes transform the model through viewing matrices to create numerous views of the model from different vantage points. Attribute nodes determine aspects of the model's appearance on the screen, control which parts of a model will be displayed, and set up picking.

Operation nodes can be set up for modeling purposes or for interaction.

For modeling purposes, translation, rotation, and scale nodes, viewing nodes, and attribute nodes are all created with fixed values. For example, a primitive might be rotated 60 degrees around the X axis to a permanent location in the coordinate system. Or a model might have a permanent perspective view imposed on it with a viewing angle of 45 degrees. Or the intensity range might be fixed at .5 to 1, and separate parts of the model might be designated as always pickable.

Interaction nodes, on the other hand, are put in the display tree to allow you to interactively manipulate the entire model or any separate part of it. To achieve this interactive manipulation, the contents of these nodes must be updated with new values. These values are supplied from a physical device such as a dial through data-handling software called a *function network* to the interactive node. The network might feed a rotation node with a series of new rotation matrices, a viewing node with a new viewing matrix, or an attribute node with information to change its function.

EVANS & SUTHERLAND AND INTERACTIVE GRAPHICS

At Evans & Sutherland, interaction has always been the most important feature of our graphics systems. For us, interaction means the ability to change the picture being displayed in an easy manner and in *real time*.

We provide ease of manipulation through offering a variety of interactive devices. A Data Tablet and stylus can be used to control a cursor on the screen for pointing at and selecting parts of the display. Eight Control Dials can be programmed to translate, rotate, and scale objects and to zoom and pan. A bank of 32 Function Buttons can be programmed to select different displays or change details of the same display. Twelve programmable Function Keys can act as toggle switches between different functions. These devices are all easy and natural to use and can be arranged comfortably at your work place. Refer to the E&S Product Data Sheets in Volume 1 for more information on the interactive devices.

Real time interaction means that the effect of an interactive device--for example, turning a dial or pressing a button--is seen instantly in the picture. If a dial is correctly programmed to rotate a model around the Y axis, then you perceive no delay between turning the dial and seeing the model respond. If you turn the dial slowly the model turns slowly, and if you turn it fast the model turns fast. When the devices are correctly programmed, minute, precise changes can be made to the orientation of a model on the screen as you watch.

Since every owner of an interactive graphics system has a different reason for using interactive graphics and different requirements and expectations of the machine, the interactive devices must be programmed to suit individual needs. Users themselves decide how they want to interact with the models they have created, and they program the devices accordingly.

In previous Evans & Sutherland systems, the host computer controlled the interactive devices as well as running the application programs and calling the routines that created the graphics. Interactive devices were checked regularly by the host computer programs to see if their state had changed. If the state had changed, the host program had to determine how and what to do about it.

The PS 300 unburdens the host by handling the interactive devices locally. The host computer never has to intervene in setting up the devices or interpreting data from them. In addition, each device contains its own microprocessor. This distribution of some intelligence to the devices themselves in turn unburdens the PS 300's Graphics Control Processor (GCP). Devices send data that has already been interpreted to the GCP. So, for example, instead of the Control Dials unit sending a stream of data whenever a dial is turned, it sends significant information only (such as which of the eight dials was turned) at significant times (every sixteenth of a turn, for instance).

PROGRAMMING THE INTERACTIVE DEVICES

The common end product of programming an interactive device is to have it change the displayed picture in some way or send information back to the host. For example, you might want the object being displayed to start and stop blinking when you press Function Key 3. Or you might want Dial 2 to rotate only the wrist joint of a mechanical arm, and Dial 4 to translate the whole model from left to right across the screen. Or when you pick an object on the screen, you may want information from the pick to be reported back to an application program on the host.

Planning for Interaction

The first step in planning for interaction is designing the display tree for the model. You must decide what sort of interaction you want and structure the display tree accordingly. For most applications of interactive graphics, you will want to interactively translate, rotate, and scale the model. For other purposes, you may also want to change the viewing matrices dynamically. And in many cases you will want to use conditional referencing, level-of-detail, and picking in interactive operations.

Interactive nodes, unlike modeling operation nodes, are created with values that will later be updated from an interactive device. Consider, for example, the simple display tree in Figure 86 for a star that can be rotated interactively.

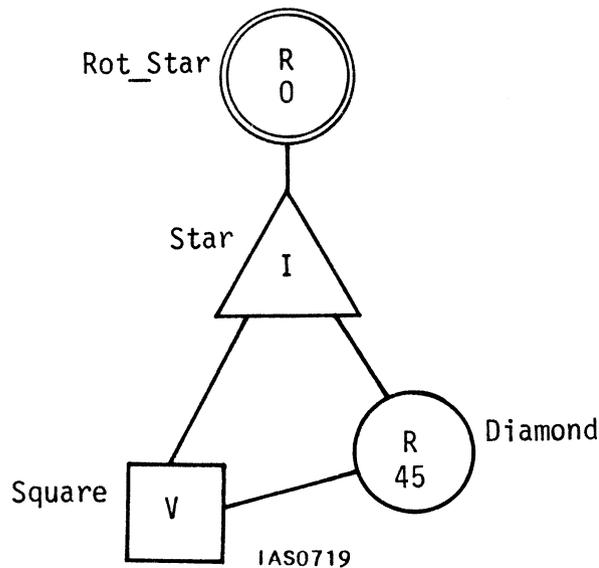


Figure 86. Display Tree for Simple Interaction

The instance node called Star groups a data node called Square and a rotation node called Diamond. The rotation node is a modeling node. It applies a 45 degree rotation matrix to the Square to create a diamond shape. Its contents never change. The rotation node Rot_Star, however, is not in the display tree for modeling purposes. It is drawn as a double circle to indicate that it is an interaction node. This rotation node is initially created with a rotation of zero degrees, so that at first it will not have an effect on the structure. Its contents will eventually be updated with a new rotation matrix from a function network as a dial is turned.

The following commands will create the display tree shown in Figure 86.

```

Rot_Star := ROTATE 0 APPLIED TO Star;
Star := INSTANCE OF Diamond, Square;
Diamond := ROTATE IN Z 45 APPLIED TO Square;
Square := VECTOR LIST N=5 .5,.5 .5,-.5, -.5,-.5, -.5,.5, .5,.5;

```

Updating a Node

Not every node in a display tree can be updated and so not every node can be an interactive node. Instance nodes, for example cannot be updated. Their function is to point to other places in the structure of the display tree. An instance node can be redefined using the INCLUDE and REMOVE commands, but new values cannot be sent to an instance node through a function network because instance nodes do not contain data.

Operation nodes, however, do contain data, in the form of matrices, vectors, numbers, and Boolean values. Most operation nodes can have their contents changed, as long as those nodes have a direct name by which they can be accessed. Data nodes contain vector lists, polygon lists, special vector lists for curves, and text in various forms. There are ways to change the contents of these nodes interactively too.

The *Command Summary* in Volume 3A shows the type of node a command creates and indicates if that node has inputs which allow it to be updated. Figure 87 shows a representation of the SET_DEPTH_CLIPPING node.

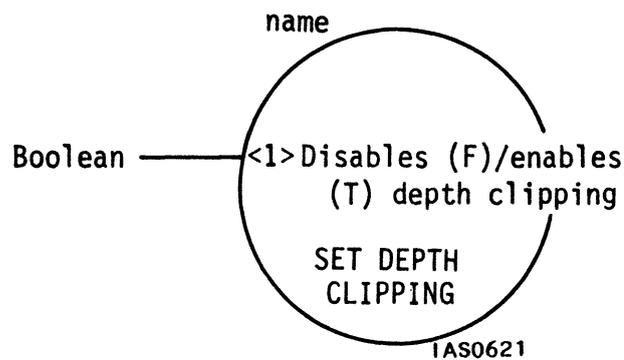


Figure 87. The SET_DEPTH_CLIPPING Node

This node has one input which accepts a Boolean true or false. A TRUE enables depth-clipping for an object and a FALSE disables it.

Supplying the Correct Type of Data

The Boolean value which the SET DEPTH_CLIPPING node requires is supplied by an interactive device. Logically, a two-state device such as a Function Key or Function Button would be programmed to act as a toggle switch, setting depth-clipping on the first time it is pressed and setting it off when it is pressed again. However, when a Function Key is pressed it generates an integer which identifies the key, not a Boolean value. Some method is needed of programming a path between the Function Key and the SET DEPTH_CLIPPING node, and of converting the integer to a Boolean value.

PS 300 FUNCTIONS

With the PS 300, interactive devices are not programmed using a standard programming language. Instead, the PS 300 uses *functions* which are combined into *networks*. The individual functions which compose a network are actually Pascal procedures, but are thought of as "black boxes" with numbered input queues and outputs, as shown in Figure 88.

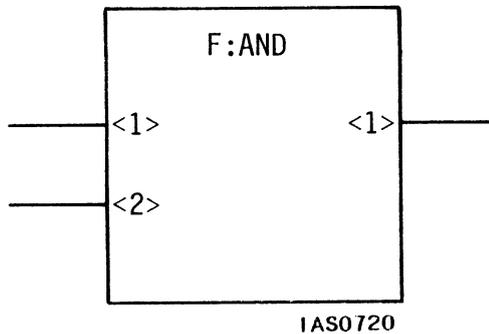


Figure 88. Representation of a Function

Each function accepts data on its input queues, performs a mathematical, logical, data conversion, routing, or selecting operation, and sends data out of its outputs. Inputs accept data from interactive devices, from the host, or from the outputs of other functions. Outputs connect to inputs of other functions or interactive nodes in a display tree.

Functions are chosen and combined so that the final network will accept data from a device and manipulate and convert the data into types that will be accepted by the interactive nodes. There are nine categories of functions available with the PS 300. These are as follows.

- Data Conversion

Data conversion functions, for example, change matrices into rows, rows into scalar elements, and real numbers to integers or vectors. Data can be output in decimal or exponential format.

- Arithmetic and Logical

These functions perform all arithmetic operations (add, divide, subtract, multiply, square root, sine, and cosine) and logical operations (and, or, exclusive-or, and complement).

- Comparison

Comparison functions test whether values are greater than, less than, equal to, not equal to, greater than or equal to, and less than or equal to other values.

- Data Selection and Manipulation

These functions are used to selectively switch functions, choose outputs, and route data.

- Viewing Transformation

Viewing transformation functions connect to viewing operation nodes in display trees to change line-of-sight, window size, and viewing angle, interactively.

- Object Transformation

Object transformation functions connect to modeling operation nodes in display trees to interactively rotate, translate, and scale objects.

- Character Transformation

These functions are used to interactively position, rotate, and scale text.

- Data Input and Output

These functions set up and control the interactive devices (dials, function keys, function buttons, data tablet, and keyboard) and output values to the optional LED labels which several of the devices have.

- Miscellaneous

Other functions set up and control picking, clocking, timing, and synchronizing operations.

The complete set of functions is loaded into memory when the PS 300 is booted. The *Function Summary* in Volume 3A is a reference to all available functions.

There are three types of functions: *intrinsic functions*, *initial function instances*, and *user-written functions*.

Intrinsic Functions

Intrinsic functions are the set of "master" functions which you can instance to create networks. Their names reflect the operation they perform, and are preceded by F:, for instance F:AND, F:ROUTE, F:MATRIX.

Functions are instanced using the NAME := F:function_name command. For example, the following command creates an instance of the ADD function (F:ADD) and assigns it the unique name Adder.

```
Adder := F:ADD;
```

Intrinsic functions are always instanced in this way. The intrinsic function name itself, in this case F:ADD, is never used in the network. The name of the function instance (i.e., Adder) is used instead.

Initial Function Instances

When the PS 300 is booted, the system itself instances (i.e., names) certain functions as initial function instances. Among other things, these functions connect to the interactive devices, connect to the host, and connect to error detection logic. For example, inputs to the initial function instance called DIALS are connected to the Control Dials unit at system initialization. DIALS has eight outputs on which it sends real numbers from one to eight, corresponding to the numbers of the eight dials. It sends values generated by the dials out of the output that corresponds to the number of the dial.

Unlike intrinsic functions, which must always be assigned a unique name, initial function instances are used with their system-assigned name. The name reflects the operation the function performs, but is not preceded by F: (for example, TABLETIN, WARNING, KEYBOARD).

User-Written Functions

You are not limited to the set of intrinsic functions and initial function instances supplied with the system. If the functions that are available do not suit all your needs, you can write your own using the optional User-Written Function facility. User-written functions are instanced in the same way as intrinsic functions. E&S provides documentation on writing Pascal procedures to create user-written functions and documentation and software files that aid in producing and transporting these procedures from the host to the PS 300. To understand user-written functions, you should know Pascal well and you should have experience in programming PS 300 function networks.

For complete information, refer to the *User-Written Functions* section in Volume 4.

Creating Networks

Networks are created by connecting initial function instances, instances of intrinsic functions, and interactive nodes in display trees using the CONNECT command. For example, the following group of commands create a simple network to rotate the star diagrammed in Figure 86 around the Z axis.

```
Rotate := F:DZROTATE;  
CONNECT DIALS<2>:<1>Rotate;  
CONNECT Rotate<1>:<1>Rot_Star;
```

The first command

```
Rotate := F:DZROTATE;
```

creates an instance of the intrinsic function F:DZROTATE named Rotate. This intrinsic function is represented in Figure 89.

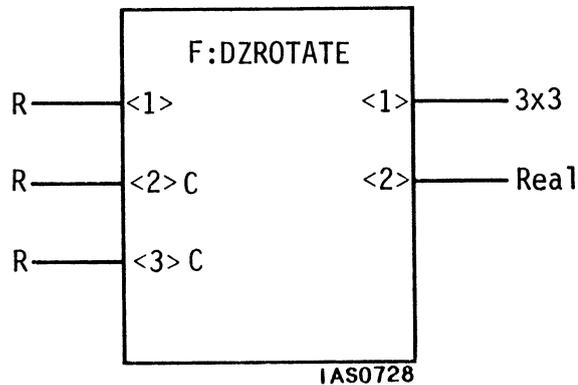


Figure 89. The F:DZROTATE Function

This function has three inputs. Input <1> accepts real numbers, usually directly from the initial function instance DIALS. Input <3> is a magnification factor. The very small numbers (from 0 to 1) that arrive at input <1> from the dial are multiplied by this factor. Input <2> is an accumulator set for the values received on input <1>. The function creates a matrix from an angle of rotation, which is derived from the accumulator contents on input <2> multiplied by the scale factor on input <3>. The matrix is sent on output <1>. Output <2> contains the accumulator contents from input <2>.

The second command

```
CONNECT DIALS<2>:<1>Rotate;
```

connects output <2> of the initial function instance DIALS to input <2> of Rotate. The initial function instance DIALS is diagrammed in Figure 90.

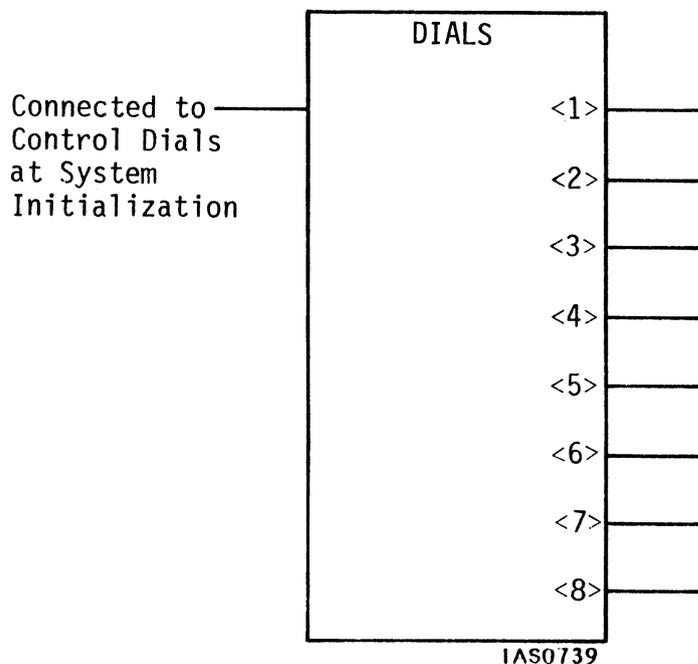


Figure 90. The Initial Function Instance DIALS

This initial function instance is connected to the Control Dials unit when the PS 300 is booted. It produces a real number on each of its eight outputs. Every output corresponds to one of the eight dials. Connecting output <2> of DIALS to input <1> of Rotate feeds values into the Rotate function from Dial 2 whenever the dial is turned.

The third command

```
CONNECT Rotate<1>:<1>Rot_Star;
```

connects output <1> of Rotate to input <1> of an interaction node called Rot_Star. Figure 91 represents a rotation node in a display tree.

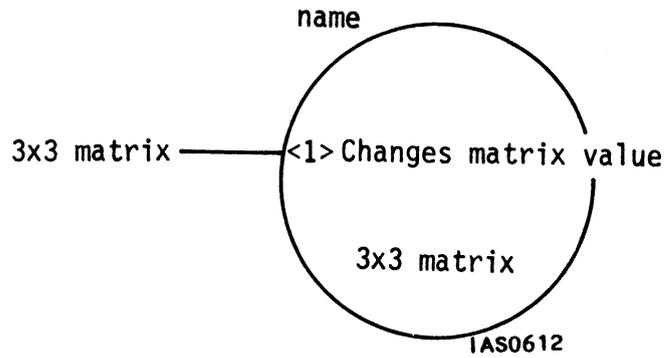


Figure 91. Inputs to a Rotate Node

This connection feeds the rotation matrix from the Rotate function to the interactive rotation node. Figure 92 is a diagram of the simple Z-rotation Function Network which the commands create.

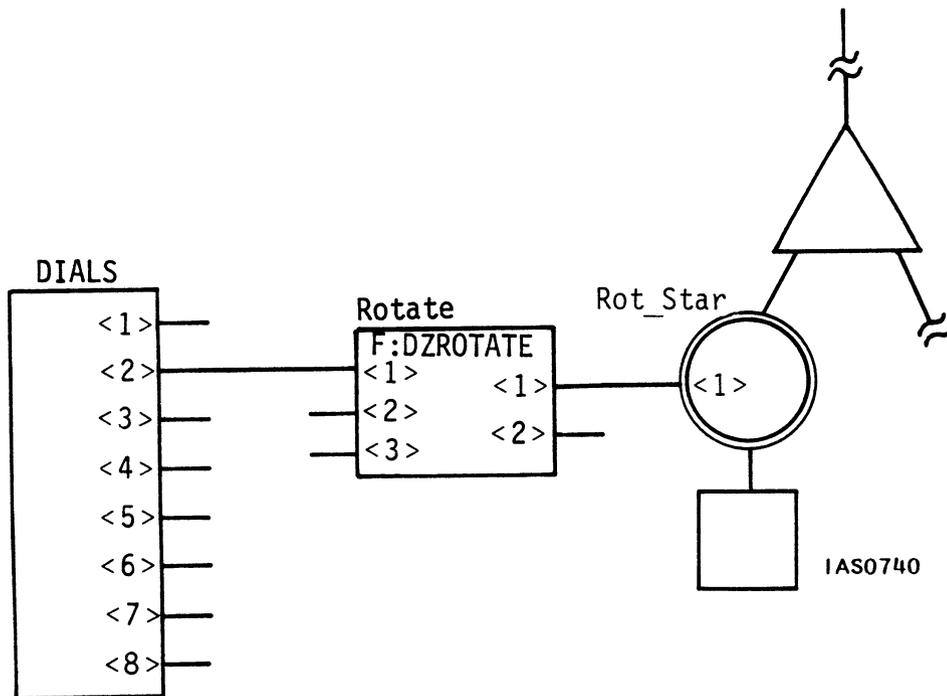


Figure 92. Simple Z-Rotation Network

Before the network will start to accumulate values from the dials correctly, however, inputs <2> and <3> on Rotate must be primed. The SEND command is used to send a magnification value of 50 to input <3> and an initial value of 0 to input <2> to set the accumulator to an initial value.

The final command file, with comments in braces, might read as follows.

```

Rotate := F:DZROTATE;           {Instance of Z-rotate function}
CONNECT DIALS<2>:><1>Rotate;    {Connect dial 2 to rotate function}
CONNECT Rotate<1>:<1>Rot_Star;  {Connect output of rotate function
                                to rotate node}

SEND 0 TO <2>Rotate;           {Set accumulator to zero}
SEND 50 TO <3>Rotate;          {Multiply values from dial by
                                magnification factor of 50}
  
```

Active and Constant Inputs

A function instance can have *active* or *constant* input queues. An active input receives data from an interactive device or from the output of another function instance. Input <1> of F:DZROTATE is an active input, for example. Each datum or *token* that arrives on an active input is a trigger for the function to execute. When the function is triggered, the datum is consumed. Constant input queues, however, are primed with a value, usually by the SEND command, and that value remains on the input queue until it is replaced by another constant value from another SEND command. For example, inputs <2> and <3> of F:DZROTATE are constant inputs. The values that are sent to those inputs prime the function. The value on input <2> sets the accumulator to an initial value. The value on input <3> is a scale factor which is use to magnify the real numbers sent from the dial.

The *Function Summary* in Volume 3A indicates whether a function has active or constant inputs: an input followed by a "C" in the *Function Summary* diagrams is a constant input. There is also a command named SETUP CNESS, which allows you to change the constant or active nature of function instance inputs. Refer to the *Command Summary* for details.

Data-Driven Networks

Individual functions and the networks they comprise are *data driven*. This means that a function only becomes active when data arrive at its inputs to be processed. Once a function has executed its task, it becomes *dormant* again until another set of tokens arrives. An entire network is dormant until activity occurs at the interactive device to which it is connected. As long as values are being sent out from the device, the network is active, converting and routing the data.

Why Function Networks?

Data driven function networks differ from conventional programming languages in that they are active only when an event occurs which produces data to be processed. Conventional programming languages are best suited for data treated as values to be looked at if necessary. Whenever data exist as asynchronous events and when the arrival of such events causes an operation to occur, then data are best handled by data-driven programs, such as function networks. Conventional programs written to handle input from interactive devices must regularly poll all the available devices to see if any activity has occurred. Once activity is detected, the type of activity has to be determined and data have to be processed accordingly.

A PS 300 with a tablet, eight control dials, twelve function keys, 32 function buttons, a keyboard, and a communications line to the host has a total of 55 independent devices which can input data. Programming in a conventional language requires each device to be polled regularly to determine if its status has changed. Function networks, however, capitalize on the fact that few of these devices are ever used at the same time. A human user of the system has only two hands and typically uses only one or two of the devices at a time. The data-driven nature of function networks schedules operations so that devices which are unused at any time do not burden the PS 300's central processing unit, the Graphics Control Processor.

Function networks are designed to filter data and perform data formatting and selection. They filter input data, for example, reducing a stream of data indicating tablet positions to just those data when the tip switch of the stylus is pressed. They reformat input data, converting a dial's value, for instance, into a rotation matrix. And they select and route data, by connecting to a node in a display tree, for example, or transmitting data back to the host application program. They do not operate like conventional computer programs, as single processes whose parts communicate via subroutine calls. Instead, they are collections of autonomous, cooperative processes whose parts communicate via packages of information which are sent out while the originator of the information goes on to do something else.

Creating Function Networks

Function networks are created as ASCII files. They can be entered by hand or generated automatically from the graphical Function Network Editor program, NETEDIT. This program is documented in Volume 4. Briefly, networks are created using a drawing program which lets you select and place symbols which represent functions. Connections are made by routing arcs between outputs and inputs, much like a wiring diagram. When a network drawing is complete, code can be generated automatically.

A network debugging aid, NETPROBE, is also available. It too is documented in Volume 4.

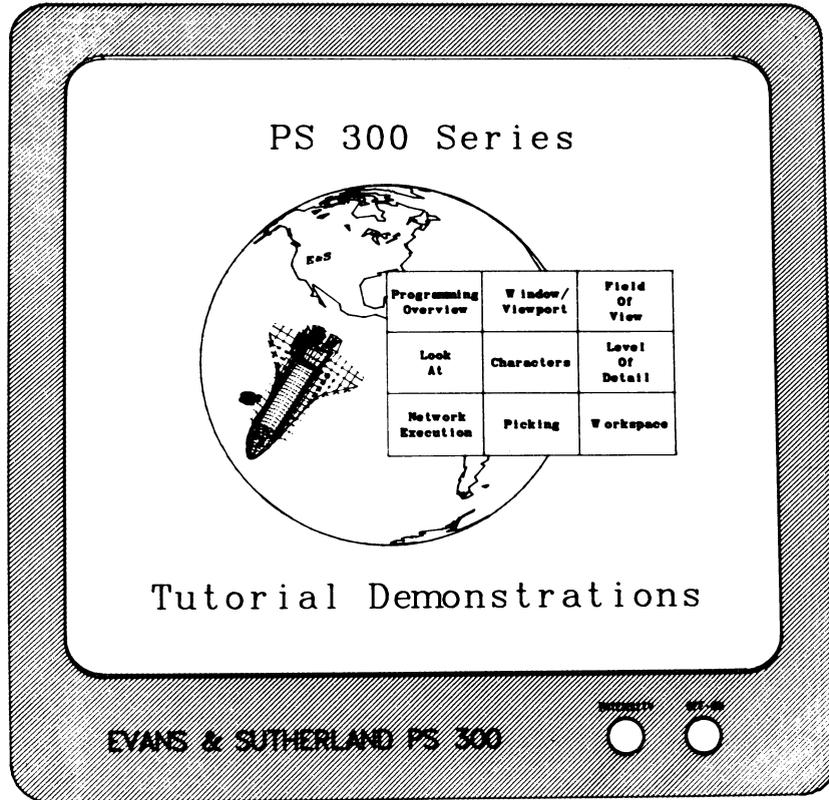
For more information on networks and their use, refer to the tutorial modules "Function Networks I" and "Function Networks II" in this volume.

SUMMARY

New Information Presented

1. Most operation nodes in a display tree can have their contents changed. Nodes that are set up for interaction have their contents updated with values from an interactive device.
2. The path between a device and a node is a function network. The network, composed of individual functions, receives data from a physical device such as a dial, manipulates those data, and produces the correct data type to update the node.
3. Networks are data driven. This means that they are only active when there is data to process.
4. Programming with PS 300 functions allows you to customize the operations of the interactive devices to suit any programming needs.

PS 300 TUTORIAL DEMONSTRATIONS



LIMITED SUPPORT DISCLAIMER

The PS 300 Tutorial Demonstrations are distributed by Evans & Sutherland as a convenience to customers and as an aid to understanding the capabilities of the PS 300 graphics systems. Evans & Sutherland Customer Engineering supports the Tutorial Demonstrations to the extent of answering questions concerning the installation and operation of the programs, as well as receiving reports on any bugs encountered while the programs are running. However, Evans & Sutherland makes no commitment to correct any errors which may be found.

PS 300 DEMONSTRATION PACKAGE

CONTENTS

1. INTRODUCTION TO THE TUTORIAL DEMONSTRATIONS	1
THE COMPONENTS OF THE TUTORIAL DEMONSTRATION PACKAGE	3
REQUIRED INTERACTIVE DEVICES	4
HOST COMPUTER REQUIREMENTS	4
2. ACCESSING THE TUTORIAL DEMONSTRATIONS	5
USING THE TUTORIAL COMMAND FILE	5
USING THE PROGRAMS ON A COLOR SYSTEM	6
3. RUNNING THE TUTORIAL DEMONSTRATION PROGRAMS	7
Tutorial Demonstrations menu – Globe and Shuttle	9
Programming	11
Window/Viewport	15
Field Of View	19
Look At	21
Characters	25
Level of Detail	27
Network Execution	29
Picking	33
Workspace	35



1. INTRODUCTION TO THE TUTORIAL DEMONSTRATIONS

The eight Tutorial Demonstration programs are designed to clarify graphics programming concepts explained in the tutorial modules in Volumes 2A and 2B of the PS 300 Document Set.

The programs display images you can interact with using the data tablet, control dials, and function keys. Typically, the keys and dials are programmed to translate, rotate, and scale the objects displayed and to change the values in the PS 300 graphics programming commands that are being illustrated. Programmed operations are shown in the LED displays above each control dial or function key.

The following concepts are illustrated in the programs.

Programming the PS 300

In three separate areas of the screen, you are shown a sequence of PS 300 commands, a representation of the structures these commands create in memory, and the picture that the commands produce on the screen. As you scroll through the commands, the contents of memory and the screen display are changed when each command takes effect.

Windows and Viewports

This program illustrates the mapping of an orthographic window in the world coordinate system to a viewport on the PS 300 screen. In one area of the screen, a sphere is shown enclosed in a window. In another, the sphere is shown as it appears when displayed on the PS 300 screen. To the side, the variables used in the WINDOW and VIEWPORT commands are listed. Using function keys and dials, you can change the dimensions of the window and the viewport and control the size and orientation of the sphere. The relation between windows and viewports is clearly shown in the resulting changes to the displayed image of the sphere.

The FIELD_OF_VIEW Command

To demonstrate the FIELD_OF_VIEW command, a sphere is shown enclosed in a perspective viewing area. In another portion of the display, the sphere is shown as it would appear on the PS 300 screen. The values entered in the FIELD_OF_VIEW command are listed to one side. Using dials you can change the viewing angle and front and back boundaries of the viewing area to see how the image of the sphere is affected on the screen.

The LOOK Command

This program shows how the LOOK command rotates and translates all points in the world coordinate system to simulate a vantage point and a line of sight towards an object. One area of the screen shows a collection of objects and an eye that can be moved in any direction to change values in the LOOK command. A second area shows the rotations and translations that are performed by the PS 300 to create the view specified in the LOOK command. A third area shows the screen display. Dials are programmed to change the "at" and "from" points in the LOOK command and to change the "up" vector.

Character Modes

The three ways in which characters can be used in an image are illustrated in this program. Three cubes are displayed with each of their faces labeled. The cubes can be rotated, translated, and scaled using control dials. The first cube contains world-oriented characters which are transformed with the cube. The second cube contains screen-oriented characters which always remain at the same size and in a plane parallel to the screen, so that they are always legible. The third cube contains screen-oriented characters which are "fixed" so that they do not vary in intensity as they move forwards and backwards (in the Z axis).

Level of Detail Settings

This demonstration shows how level-of-detail nodes can be used in a structure to display changing images of an object in response to changing values from a function network. A display tree is shown with a SET node connected to a network and IF nodes at the head of each of twelve hierarchical branches. As the value in the SET node is updated from the network, a different branch is traversed. This produces an animation sequence of 12 frames in which the ends of a cylinder twist and untwist in opposite directions.

Execution of a Function Network

This program illustrates the relationship between interactive devices, function networks, interactive nodes in a display tree, and a dynamically changing image. In one area of the screen, an object is shown which consists of two wheels and a tie-bar. A display tree is shown for the structure of this object. The tree contains interactive rotation and translation nodes connected to a dial through a function network. As you turn the dial to rotate the wheels, the function network is shown accepting data, converting it to matrices, and updating the nodes in the display tree.

Picking

To illustrate picking, this program shows a collection of objects consisting of two cubes, a B-spline curve, a character string and a labels block. The display tree for these objects is shown with the required SET PICKING node and pick identifier nodes. A picking network is connected to the display tree. When a vector, character, or label is picked, the branch traversed in the display tree is highlighted and the information returned from the pick on the outputs of the function F:PICKINFO is shown.

This manual explains how to install the Tutorial Demonstrations and how to run each of the programs.

This section lists the components of the Tutorial Demonstrations and explains the interactive devices and host computer requirements for running the demonstrations.

Section 2 explains how to install the Tutorial Demonstration programs on your system.

Section 3 gives complete operating instructions for each of the programs.

1.1 THE COMPONENTS OF THE TUTORIAL DEMONSTRATION PACKAGE

The PS 300 Tutorial Demonstration package consists of several files distributed on magnetic tape.

The tape contains control networks, the Tutorial Demonstrations Menu from which programs are chosen, the programs themselves, and several character fonts. Also included are the vector lists for the primitives used in the tutorial modules in Volumes 2a and 2b of the PS 300 Document Set.

1.2 REQUIRED INTERACTIVE DEVICES

The following interactive devices are required to run the Tutorial Demonstration programs.

- Data Tablet and Stylus
- Keyboard with Function Keys
- Control Dials

The data tablet and stylus are used to pick programs from the menu and to interact with the objects displayed by some of the programs.

The function keys and control dials are programmed through function networks to perform various graphical operations such as scaling, rotating, and translating the images displayed and to change dynamically the values in the PS 300 commands being illustrated. The operation controlled by each function key and control dial is displayed in its red LED label.

1.3 HOST COMPUTER REQUIREMENTS

The eight programs that comprise the Tutorial Demonstrations are run locally on the PS 300. There are no host computer requirements for running the programs.

The files that are distributed on the tape must be loaded onto a host computer and then transferred to the PS 300. There are two requirements for the host computer for storing and transferring the files. First, it must have sufficient memory to contain the files on the tape: approximately 1166K bytes are needed. Second, the host must be able to communicate with the PS 300 so that the files can be transferred.

2. ACCESSING THE TUTORIAL DEMONSTRATIONS

The Tutorial Demonstrations may be run on any member of the PS 300 family of graphics systems. The procedure for loading the files and transferring them from the host is the same for all systems.

The complete Tutorial Demonstrations package takes between 15 and 20 minutes to transfer from the host to the PS 300, depending on the current work load on the host.

2.1 USING THE TUTORIAL COMMAND FILE

System managers at individual sites will set up a method on the host computer to gain access to the Tutorial Demonstrations as well as the objects that are required by some of the tutorial modules in Volumes 2a and 2b and the Sample Programs. A command file displaying the following menu should be available.

PS 300 GRAPHICS PROGRAMMING TUTORIAL

Set to be loaded	First used in module ...
1. Demonstrations	
2. Sports Car	Command Language
3. Molecule	Conditional Referencing
4. Complete Robot	Function Networks I & II
5. Sphere and Cylinder	Command Language & Conditional Referencing
6. Sample Programs	Sample Programs

Enter the number of the selection you want. Loading is complete when the host operating system prompt is displayed again.

2.2 USING THE PROGRAMS ON A COLOR SYSTEM

If you are using a color display (the CSM Calligraphic Display) and the line quality is poor, put the PS 300 in Command Mode by pressing the CTRL and LINE LOCAL keys and then the RETURN key. Now enter the command:

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

You will see the line quality and color improve. Now press the SHIFT and LINE LOCAL keys to return the PS 300 to Interactive Mode so you can make selections from the menu.

3. RUNNING THE TUTORIAL DEMONSTRATION PROGRAMS

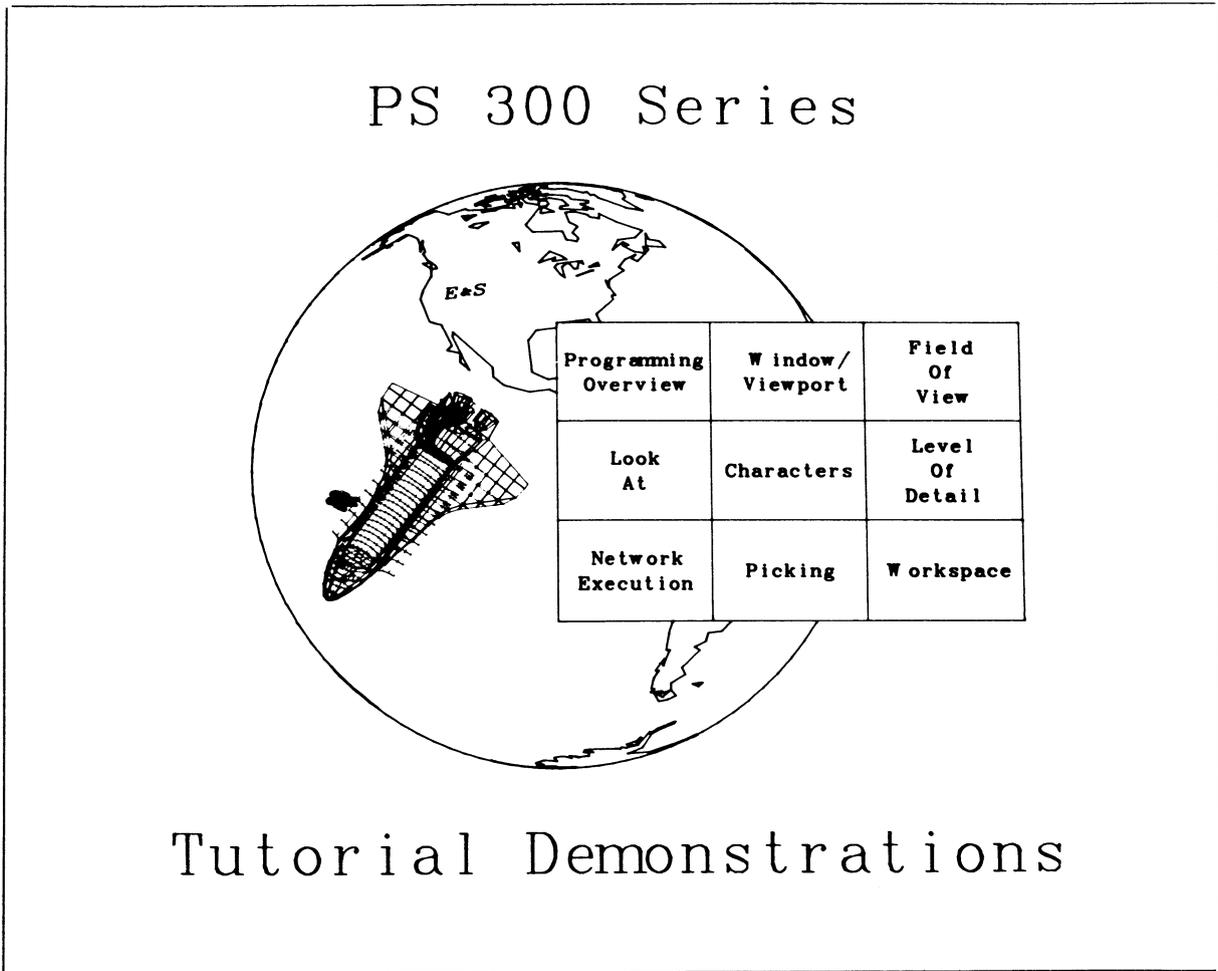
This section describes how to run each of the Tutorial Demonstration programs. Each description is organized as follows.

Typical screen displays are illustrated. An abstract points out some of the features of the PS 300 that are shown in the demonstration. The programmed functions and the LED labels that appear on control dials and function keys are listed. Notes on usage give instructions for running the program.



Program: TUTORIAL DEMONSTRATION MENU - GLOBE AND SHUTTLE

Typical Program Display



Abstract

This program serves both as a demonstration in itself and as the menu from which the other Tutorial Demonstration programs are picked. Several windows and viewports are combined to produce a very complex dynamic image. In the center of the screen is the earth spinning on its axis. Orbiting the earth is a Space Shuttle, and closely hugging the shuttle in a tight orbit of his own is one of the crew in a Manned Maneuvering Unit. Overlapping the globe to the right is the menu from which the programs are selected.

Programmed Functions

Control Dials

D1 - OS X ROT (globe and shuttle)
D2 - OS Y ROT (globe and shuttle)
D3 - OS Z ROT (globe and shuttle)

Function Keys

F10 - STRT/STP
F11 - RESET

Notes on Usage

To use this program as a menu, pick the demonstration you want to run by positioning the cursor over the name and pressing the stylus down on the data tablet. Whenever you exit from a program by pressing F12, you are returned to this display.

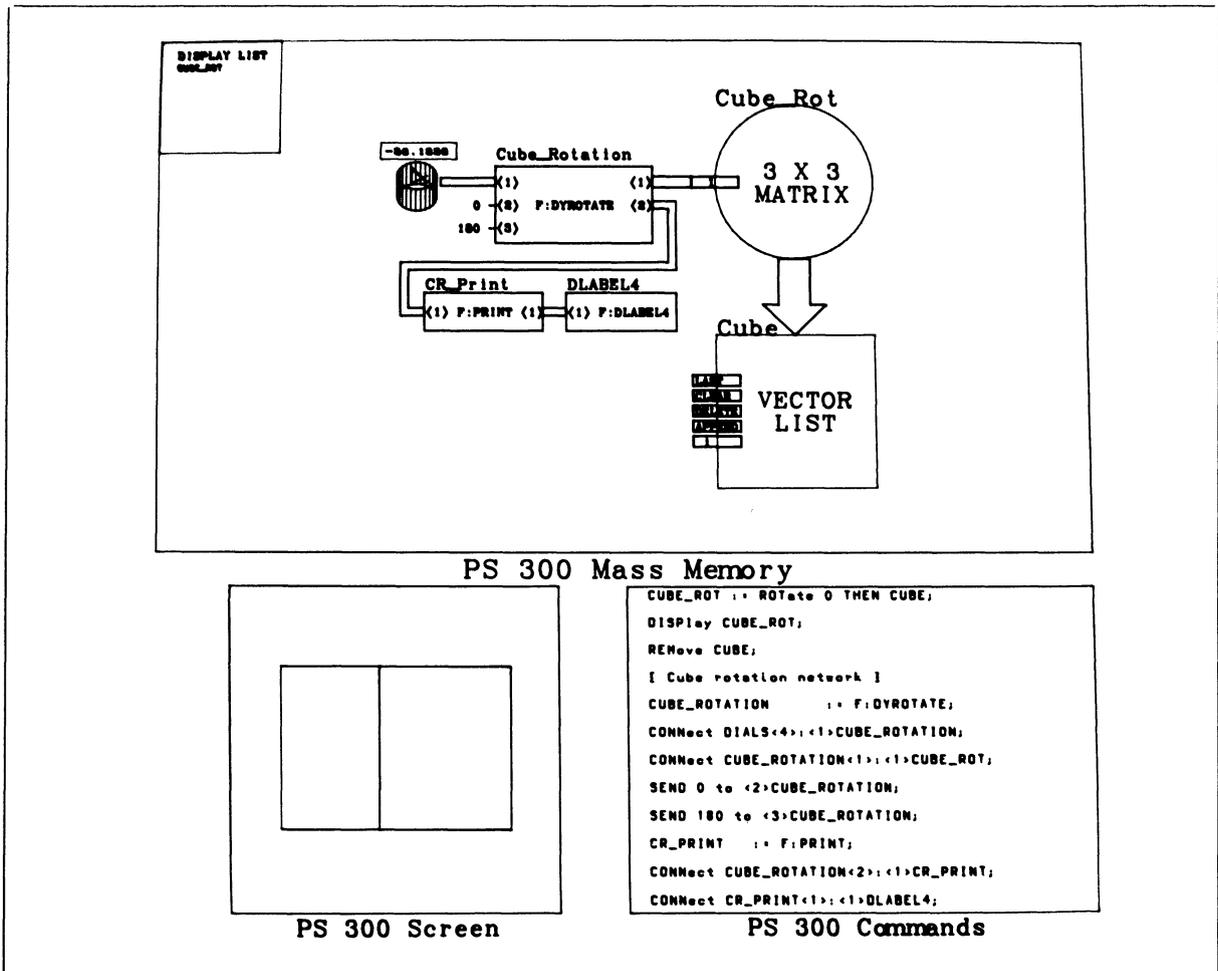
The function keys and dials let you interact with the spinning globe and space shuttle displayed in the center of the screen. Dials 1 through 3 let you rotate the globe and shuttle around the X, Y, or Z axes. The "OS" in the dial labels stands for Object Space. An object rotates in Object Space when it rotates about a set of axes which are different from the world coordinate system axes.

Function key F10 starts and stops the rotation of the globe and shuttle.

Function key F11 resets the orientation of the globe and shuttle.

Program: PROGRAMMING

Typical Program Display



Abstract

This is a graphical introduction to programming the PS 300 with commands and function networks. It illustrates how PS 300 commands create structures in memory and affect images being displayed, as well as how some of the interactive devices are programmed with simple networks.

Abstract (continued)

After an initial introductory message, the screen is divided into four viewports representing the contents of the display list, the contents of mass memory, the PS 300 screen, and commands which are entered. When you turn control dial 8, commands appear in the Commands viewport. Each time a complete command is displayed, the display in the other viewports is changed to reflect updates to mass memory, the display list, or the PS 300 screen.

Programmed Functions

Control Dials

D4 - (rotate the object displayed)
D8 - (progress through program)

Function Keys

F1 - (used with conditional
F2 - referencing commands)
F11 - RESET
F12 - EXIT

Notes on Usage

When the Programming demonstration is chosen, the following message is displayed.

Introduction to
PS 300 programming theory:
- display data structures
- data-driven function networks

Turn Dial 8 (bottom right dial) clockwise to progress through this tutorial. The effects of PS 300 commands on the screen and mass memory will be shown.

Notes on Usage (continued)

Dial 8 controls your progress through this program. As you turn the dial, the PS 300 commands will scroll in the lower right viewport of the screen. As the semicolon terminator for each command becomes visible, the effect of the command will be reflected in the other viewports on the screen. Function keys F1 and F2 and control dial 4 become active as the commands controlling them become visible.

The program starts by showing how the VECTOR_LIST command creates a data node (shown as a square) called CUBE in memory. Nothing appears on the screen or in the Display List, however, until the DISPLAY command is used. A rotation node (shown as a circle) called CUBE_ROT is created in memory using the ROTATE command. Since this command is applied to CUBE, the node becomes part of the same display tree. The two entities are displayed simultaneously as one bright cube, because the display processor is traversing both nodes. CUBE is then removed from the display list and CUBE_ROT is displayed alone.

Next, the capability of the PS 300 to do graphical manipulations locally is shown through the use of functions. An instance of the Y-rotation function F:DYROTATE is created and connected to control dial 4. This dial can now be turned to rotate the cube displayed in the PS 300 Screen viewport. To see the value of the rotation, an instance of the print function (F:PRINT) is created and connected to dial label 4. When dial 4 is turned, the value of the rotation will now be displayed in dial 4's LED and in the mass memory viewport.

A scale node named CUBE_S is now created and applied to CUBE. This shows how one vector list can be displayed in two different ways, one through CUBE_ROT, and one through CUBE_S. Now an instance node (a triangle) called VIEW is created, the display is initialized, and VIEW is displayed. At first, VIEW groups nothing more than the rotation node CUBE_ROT and the data node CUBE. Then the INCLUDE command is used to include CUBE_S in VIEW also, so both CUBE_ROT and CUBE_S are displayed with the one display command.

CUBE_S is then redefined as a 2x2 scaling matrix applied to CUBE_CHAR, a null structure which has not yet been defined. CUBE_CHAR is then defined as the character string "PS 300", which is displayed on the screen. CUBE_S is now redefined to be a special 2x2 skewing matrix to italicize the characters. Using an alternate character font, the string is then displayed in an Old English character set.

The LOOK command is used to view the structure being displayed from an arbitrary point in space. The use of BEGIN_STRUCTURE ... END_STRUCTURE is shown as an alternative to naming every command. The FIELD_OF_VIEW command is applied to the structure to create a perspective view of the cube.

Notes on Usage (continued)

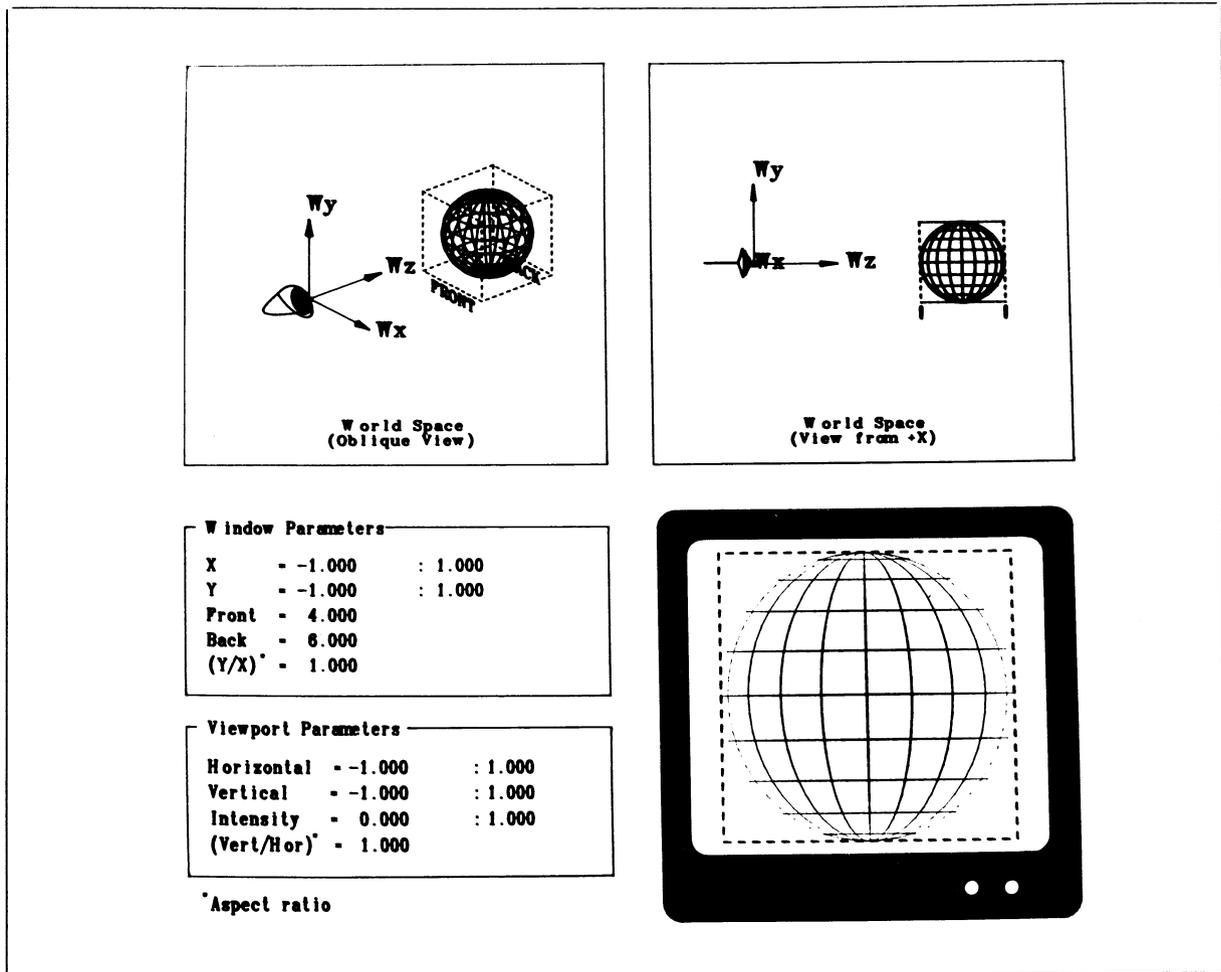
The display tree is next enhanced to include conditional references to different branches of the hierarchy. Function keys are connected to the SET LEVEL_OF_DETAIL node. F1 controls display of one branch of the hierarchy, F2 controls display of the other. A similar operation is performed with the SET CONDITIONAL_BIT node, but now the objects can be displayed independent of each other, as determined by the CONDITIONAL_BIT test. The cube is displayed if conditional bit one is set, the text if bit two is set. Another way to conditionally branch is shown, using the SET RATE node. The number of refresh frames on and off are given, and a phase attribute is set so that for 20 frames, the phase attribute is on and for 40 frames it is off. By doing a test of the phase attribute, the cube is displayed for 20 frames and the text for 40 frames.

Note that you can go back through the program by turning dial 8 in the opposite direction.

Function key F11 resets the screen to the initial display.

Function key F12 leaves this program and displays the Tutorial Demonstrations Menu again.

Program: WINDOW/VIEWPORT

Typical Program DisplayAbstract

This program shows the relationship between windows and viewports and illustrates the use of the WINDOW and VIEWPORT commands.

Two areas of the screen show two views of the world coordinate system, one from the +X axis, and one from an oblique angle. A sphere is shown in the world coordinate system enclosed in an orthographic window. Another area of the screen shows the sphere as it would be displayed on the PS 300 screen in a full-screen viewport. Values for the WINDOW and VIEWPORT commands are shown to the left.

Abstract (continued)

In one mode of operation, dials let you change the X and Y values of the window and the location of the front and back boundaries. In another mode, the dials change the horizontal and vertical values of the viewport and the intensity setting. The aspect ratio for the window (X/Y) and for the viewport (vertical/horizontal) are also shown.

Programmed Functions

Mode 1

Control Dials

- D1 - WIN XMIN (window's minimum X value)
- D2 - WIN YMIN (window's minimum Y value)
- D3 - WIN ZMIN (window's minimum Z value)
- D4 - unused
- D5 - WIN XMAX (window's maximum X value)
- D6 - WIN YMAX (window's maximum Y value)
- D7 - WIN ZMAX (window's maximum Z value)
- D8 - unused

Mode 2

Control Dials

- D1 - W X CENT (move window along X axis)
- D2 - W Y CENT (move window along Y axis)
- D3 - W Z CENT (move window along Z axis)
- D4 - unused
- D5 - OBJ XROT (rotate objects around the X axis)
- D6 - OBJ YROT (rotate objects around the Y axis)
- D7 - OBJ ZROT (rotate objects around the Z axis)
- D8 - OBJ SIZE (scale objects)

Mode 3

Control Dials

- D1 - VP H MIN (viewport's minimum horizontal value)
- D2 - VP V MIN (viewport's minimum vertical value)
- D3 - VP I MIN (viewport's minimum intensity value)
- D4 - VP HCENT (move viewport along X axis)

Mode 3 (continued)

Control Dials

D5 - VP H MAX (viewport's maximum horizontal value)

D6 - VP V MAX (viewport's maximum vertical value)

D7 - VP I MAX (viewport's maximum intensity value)

D8 - VP VCENT (move viewport along Y axis)

Function Keys

F1 - MODE 1

F2 - MODE 2

F3 - MODE 3

F4 - DEPTH CL (depth clipping)

F11 - RESET

F12 - EXIT

Notes on Usage

In Mode 1 (when function key F1 is pressed) the dials change the window's X, Y, and Z minimum and maximum values.

In Mode 2 (when function key F2 is pressed) the dials let you move the window along the X, Y, and Z axes and rotate and scale the sphere.

In Mode 3 (when function key F3 is pressed) the dials let you change the vertical and horizontal minimum and maximum values for the viewport and the minimum and maximum values for the intensity range. In this mode of operation, you can also move the viewport along the X and Y axes.

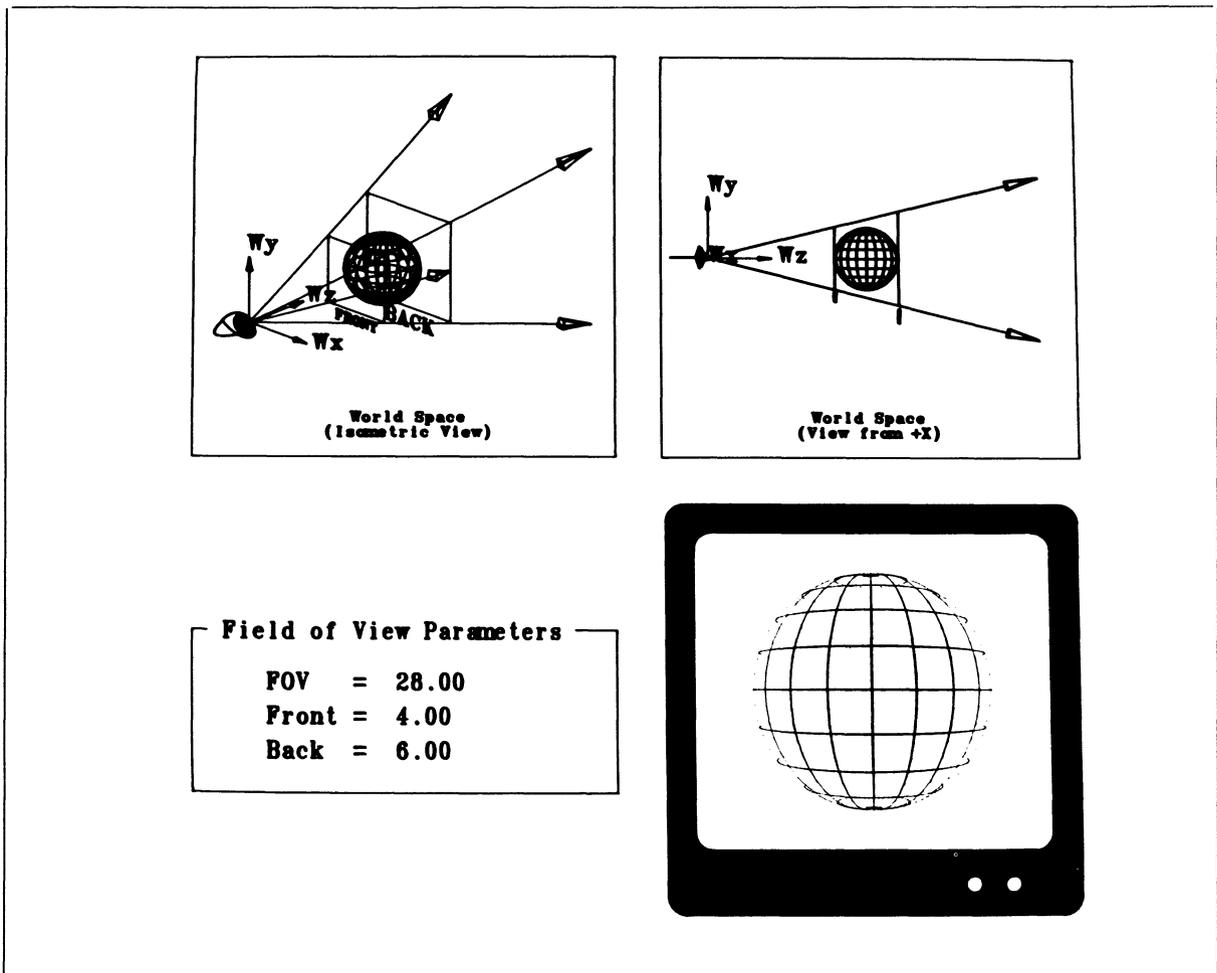
Depth clipping is on when the program is first called. Use function key F4 to turn it on and off.

Function key F11 resets the program.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.



Program: FIELD OF VIEW

Typical Program DisplayAbstract

Perspective views of objects are created using the `FIELD_OF_VIEW` command. This program illustrates how to use that command. Two viewports show the world coordinate system from two different vantage points. In each, a sphere is shown enclosed in a perspective viewing area. A third viewport shows the sphere as it would be displayed on the PS 300 screen. Values for command variables (viewing angle and front and back boundaries) are also shown. Dials allow you to change the viewing angle, the location of front and back boundaries, and the size and orientation of the sphere.

Programmed Functions

Control Dials

D1 – WS X ROT
D2 – WS Y ROT
D3 – WS Z ROT
D4 – SCALE
D5 – FOV ANGL (field-of-view angle)
D6 – FRONT
D7 – BACK
D8 – BOTH

Function Keys

F11 – RESET
F12 – EXIT

Notes on Usage

The FIELD_OF_VIEW command (abbreviated to FOV) encloses an object in a viewing space shaped like a frustum (a section of a pyramid). The eye point, established by the LOOK command, is at the apex of the pyramid. The top and bottom planes of the frustum are the front and back boundaries in the FOV command.

Dials 1 through 4 manipulate the sphere, allowing you to rotate and scale it.

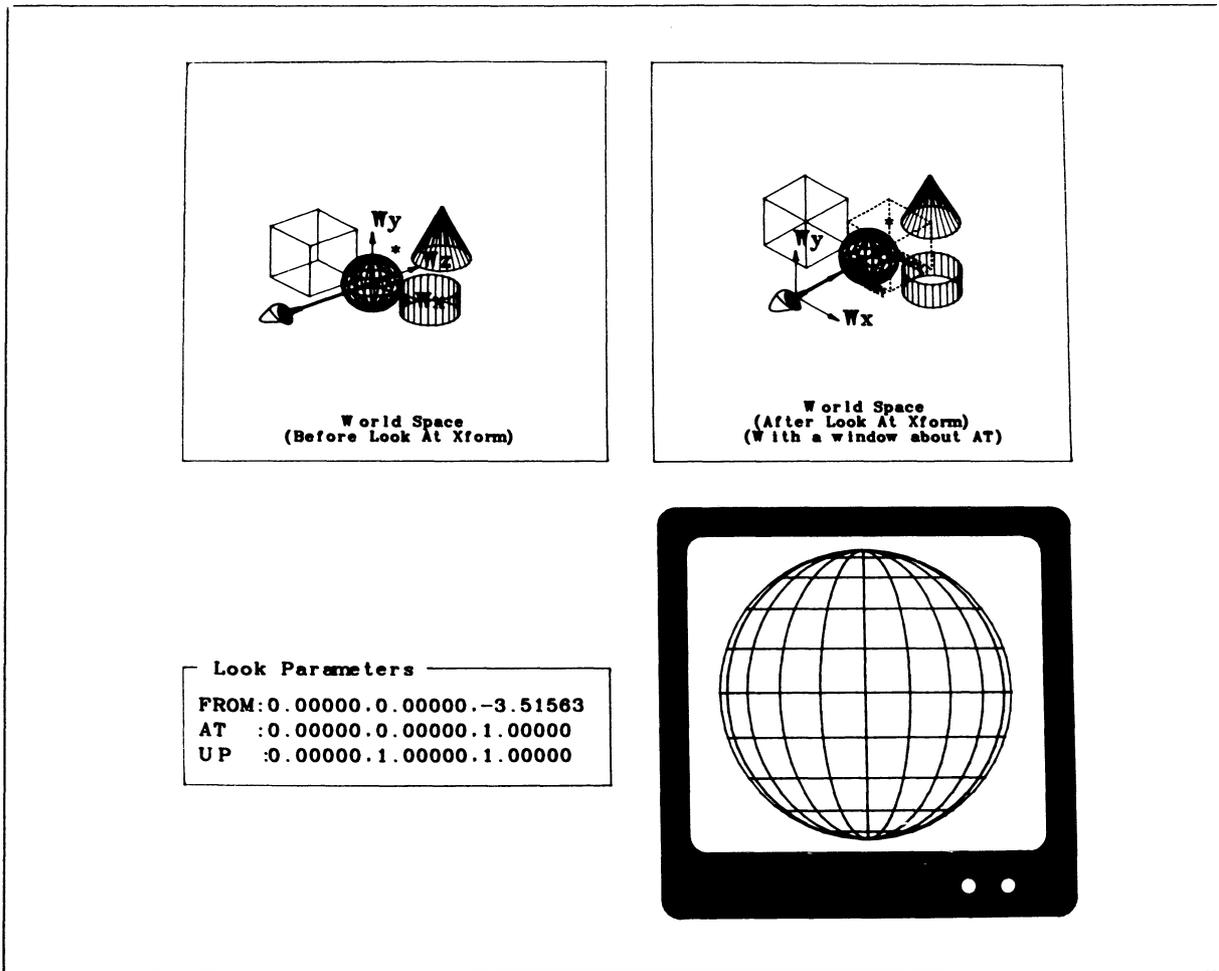
Dial 5 controls the viewing angle. Notice that as the angle increases, the size of the image on the screen shrinks and vice versa. A larger viewing angle encloses more of the coordinate system in the viewing space, a smaller viewing angle encloses less.

Dials 6 through 8 move the front and back boundaries (clipping planes) of the viewing area. Dial 8 moves both boundaries together.

Function key F11 resets the program.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.

Program: LOOK AT

Typical Program DisplayAbstract

This deceptively simple program illustrates how the LOOK command works. In one viewport, the world coordinate system is shown containing a cube, sphere, cone, and cylinder, the three axes, and an eye. This viewport represents the world coordinate system before the LOOK transformation is applied to the objects. A second viewport shows the coordinate system after the transformation has taken place. A third area shows the values for the "from," "at," and "up" points. A fourth area shows the PS 300 screen and the objects being displayed.

Programmed Functions

Mode 1

Control Dials

- D1 – FROM X
- D2 – FROM Y
- D3 – FROM Z
- D4 – WS Y ROT (world space Y rotation)
- D5 – DOLLY X (rotate eye point around the X axis)
- D6 – DOLLY Y (rotate eye point around the Y axis)
- D7 – DOLLY Z (rotate eye point around the Z axis)
- D8 – FOR/BACK (move eye point forward and back along Z axis)

Mode 2

Control Dials

- D1 – AT X
- D2 – AT Y
- D3 – AT Z
- D4 – unused
- D5 – OBJ XROT (rotate objects around the X axis)
- D6 – OBJ YROT (rotate objects around the Y axis)
- D7 – OBJ ZROT (rotate objects around the Z axis)
- D8 – OBJ SIZE (scale objects)

Mode 3

Control Dials

- D1 – UP X
- D2 – UP Y
- D3 – UP Z
- D4 – unused
- D5 – OBJ XTRAN (translate objects along the X axis)
- D6 – OBJ YTRAN (translate objects along the Y axis)
- D7 – OBJ ZTRAN (translate objects along the Z axis)
- D8 – unused

Function Keys

F1 - MODE 1
F2 - MODE 2
F3 - MODE 3
F4 - DEPTH CL (depth clipping)
F5 - MOVE UP (move/don't move "up" point with the eye point)
F11 - RESET
F12 - EXIT

Notes on Usage

The LOOK transformation is a 4×3 transformation matrix. It applies a translation and rotation to every point in the world coordinate system to produce a view which corresponds to the "from," "at," and "up" points given in the LOOK command.

All points are translated so that the eye is at the origin, and rotated so that the "at" point is in the positive Z axis and the "up" vector is in the YZ plane. These transformations are shown in the second viewport.

A window is built around the "at" point in the second viewport so that whatever is being looked at will appear on the PS 300 display in the third viewport. Initially, the sphere is displayed. As you manipulate the "at" point, the window is moved also to maintain a display on the simulated PS 300 screen.

The "up" point is shown as an asterisk. Function key F5 is a toggle which lets you move or not move the "up" point with the "from" and "at" points.

Function key F4 is a toggle which lets you turn depth-clipping on and off.

Dial 2 in Mode 1 rotates the objects and the eye in the first viewport so you can see better where the eye is located.

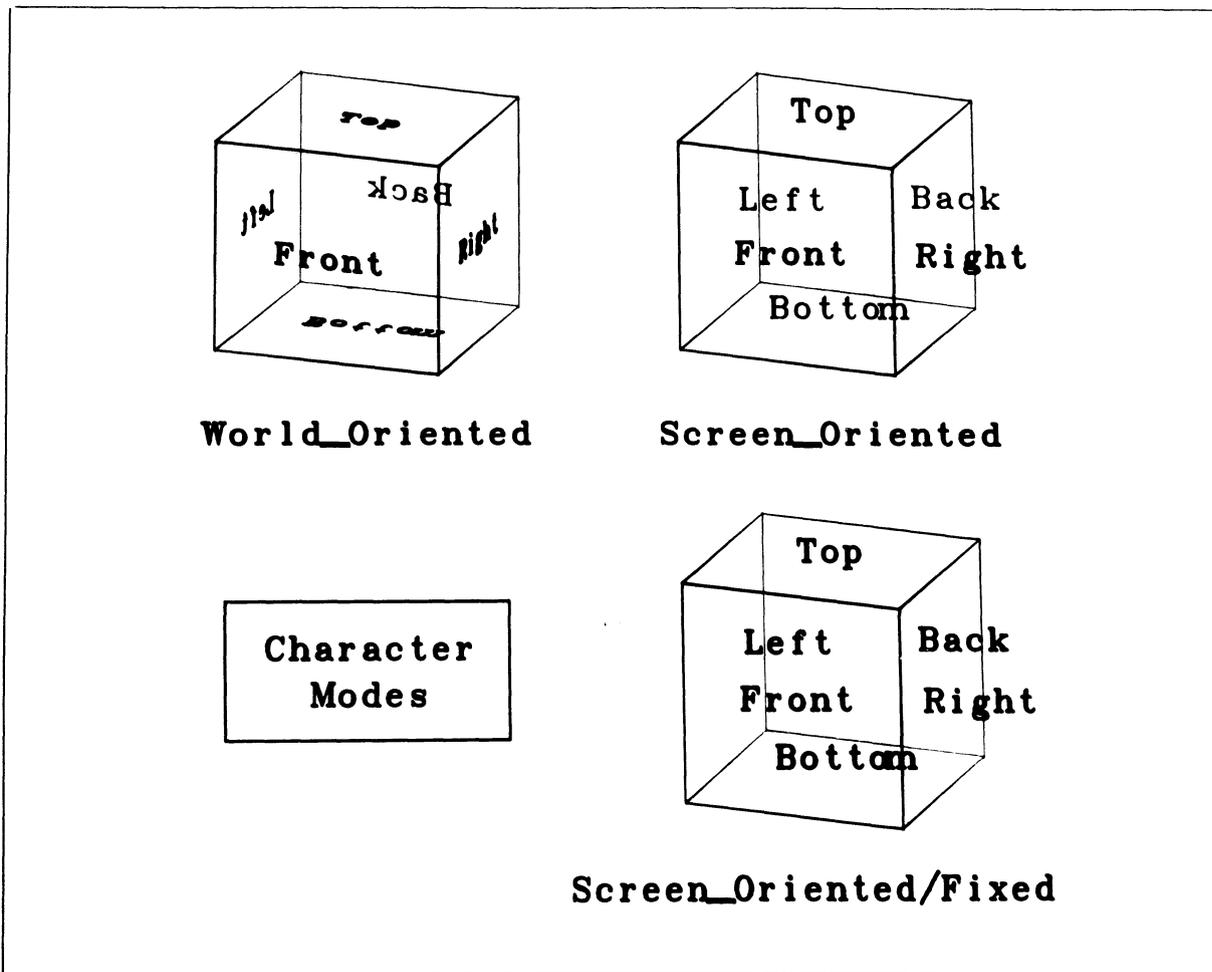
Function key F11 resets the program.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.



Program: CHARACTERS

Typical Program Display



Abstract

This program illustrates the concept of character orientation discussed in the "Character and Text Handling" module. It shows the three ways in which characters can be defined with the SET CHARACTERS command. Three cubes are displayed with their faces labeled. Characters in the first cube are created with the WORLD_ORIENTED clause (the default). They are transformed as an intrinsic part of the cube as if they were painted on the cube's faces. Characters in the second and third cubes are created with the SCREEN_ORIENTED clause (the default setting). No matter how the cube is

Abstract (continued)

rotated, these characters always remain in a plane parallel to the screen. Character size is unaffected by scaling. In addition to being screen-oriented, the characters in the third cube have an additional FIXED clause. This maintains the characters at full intensity, no matter where they are located in the Z axis.

Programmed Functions

Control Dials

D1 - OS X ROT
D2 - OS Y ROT
D3 - OS Z ROT
D4 - SCALE
D5 - TRANS X
D6 - TRANS Y
D7 - TRANS Z
D8 - unused

Function Keys

F11 - RESET
F12 - EXIT

Notes on Usage

As you manipulate the cubes with the control dials, note that the screen-oriented characters remain in a plane parallel to the screen but that they do move along the Z axis when the cubes are rotated in X and Y. Also, when the cubes are scaled, the screen-oriented characters remain at the same size but the starting location of each character string responds to the scaling.

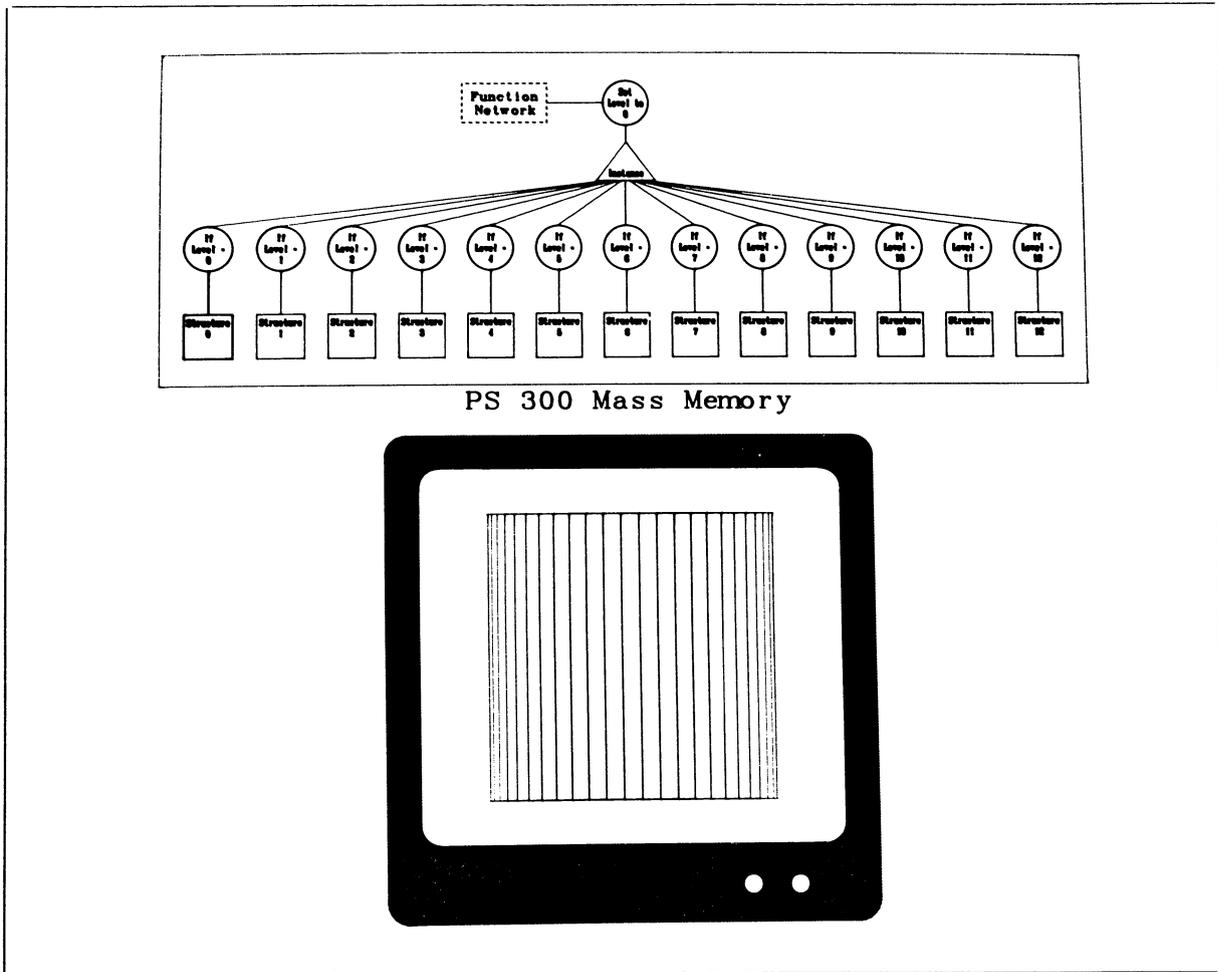
To see more clearly the difference between SCREEN_ORIENTED and SCREEN_ORIENTED/FIXED characters, turn down the intensity of the PS 300 display. If you turn it low enough, only the "fixed" characters will be visible.

Function key F11 resets the orientation of the cubes.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.

Program: LEVEL OF DETAIL

Typical Program Display



Abstract

This program shows how level-of-detail commands are used to set up conditional branching in a display tree.

In one area of the screen, a display tree is shown with a SET LEVEL node at the top. Thirteen different paths are grouped under one instance node following the SET node. Each branch contains an IF LEVEL node and a "structure." To keep the diagram simple, the structure is shown as a square data node, but it actually consists of a vector list and a color node. The IF nodes contain values from 0 to 12. The SET node is connected to a function network.

Abstract (continued)

As new values from 0 to 12 are received from the network, different branches out of the instance node are traversed. The effect of this is seen in the lower part of the display, where a representation of the PS 300 screen is shown. Each of the thirteen structures is a "frame" in a sequence which shows a cylinder whose top and bottom twist and untwist in opposite directions.

Programmed Functions

Control Dials

D1 - WS X ROT
D2 - WS Y ROT
D3 - WS Z ROT
D4 - SCALE
D5 - X TRAN
D6 - Y TRAN
D7 - Z TRAN
D8 - LEVEL

Function Keys

F10 - STRT/STP
F11 - RESET
F12 - EXIT

Notes on Usage

Dials 1 through 7 are just for fun. They let you manipulate the cylinder while it is cycling through its animation sequence.

Function key 10 starts and stops the animation. When the motion is stopped, you can use dial 8 to change the level of detail by one value at a time to step through the animation sequence.

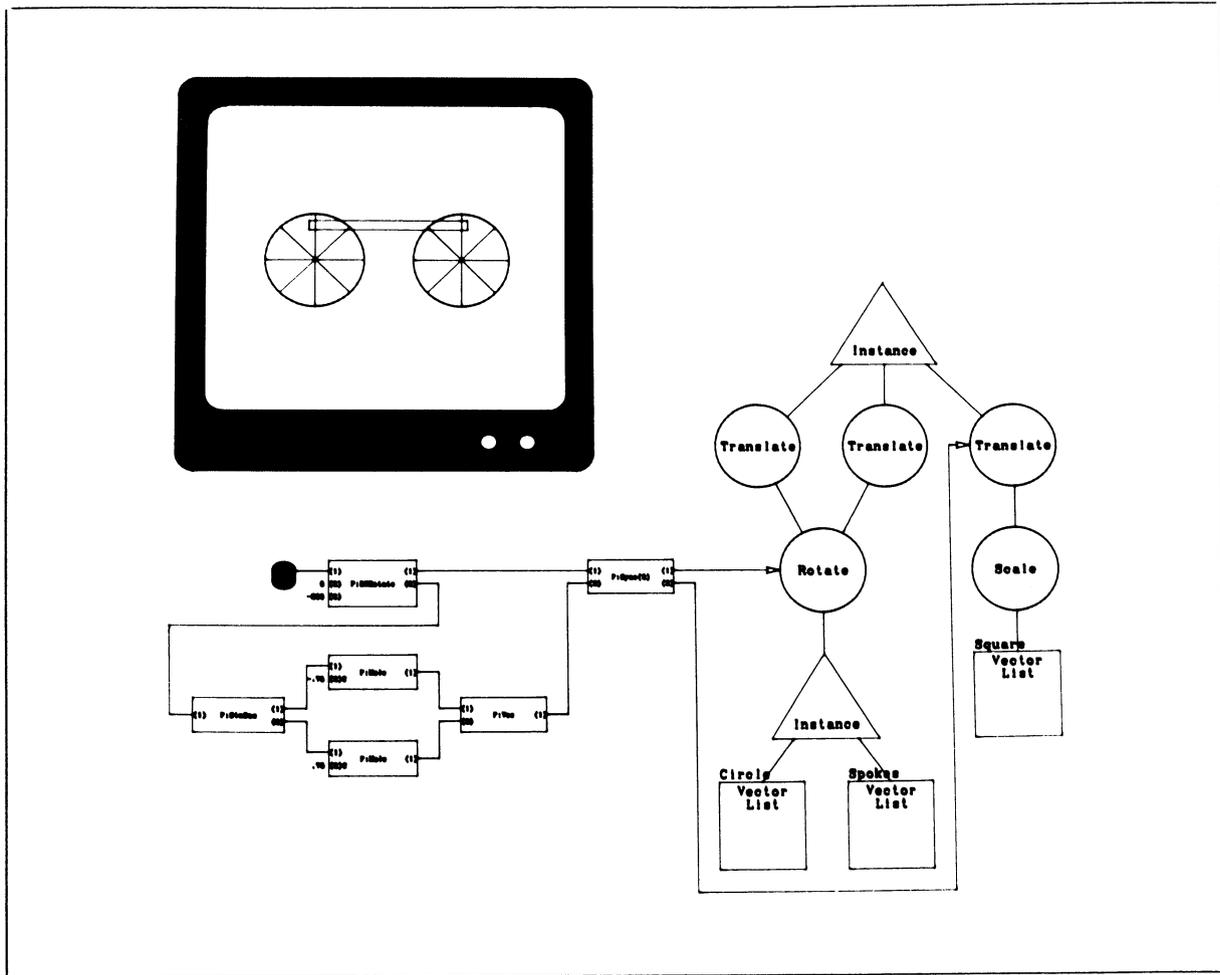
Note that the "WS" in the LEDs for dials one, two, and three stands for World Space. Rotations of this sort happen about the world coordinate axes.

Function key F11 resets the program.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.

Program: NETWORK EXECUTION

Typical Program Display



Abstract

This program shows the sequence of activities when a function network accepts data from a dial, processes the data through the functions that make up the network, and updates interaction nodes in a display tree with the resulting transformation matrices.

Abstract (continued)

A representation of the PS 300 screen is shown in one viewport, displaying the object defined by the display tree: two wheels connected by a tie bar. In another viewport, the display tree is shown with the interactive rotation and translation nodes that will supply motion to the object connected to a function network. The network connects to dial 8. As you turn the dial, the values received from it are passed through the network, converted to the correct data types, and fed into the interactive rotation and translation nodes at the end.

Programmed Functions

Control Dials

D8 - WHEELROT

Function Keys

F11 - RESET
F12 - EXIT

Notes on Usage

There are two parts to the network which supplies new values to the interactive rotation and translation nodes. One part handles the simultaneous rotations of the two wheels; the other part handles the synchronized translation of the tie bar with the motion of the wheels.

The rotation network consists of the function F:DZROTATE connected to dial 8. The magnification value on input <3> of this function increases each tiny value received from the dial by two hundred to create significant numbers to accumulate. The accumulator on input <2> is initially set to zero. As the function accumulates values, it converts them to a Z-rotation matrix which is sent out of output <1>. The accumulator contents are sent out of output <2>.

The translation network calculates the amount in X and Y by which the center of the tie bar must be translated to be synchronized with the motion of the wheels. The accumulator contents from the F:DZROTATE function (representing degrees of rotation around the Z axis) are fed into F:SINCOS. This function calculates the sine and cosine of the angle of rotation. These values are output by F:SINCOS and are multiplied by a constant value of .75 in one case and -.75 in the other to calculate the displacement of the tie bar. (The value is .75 because the center of the tie bar is initially located at 0 in X and .75 in Y.) The resulting values are fed into the F:VEC function and are converted to a 2D translation vector.

Notes on Usage (continued)

The outputs of F:VEC and F:DZROTATE are fed into F:SYNC(2). This synchronizes the updating of the rotation node and the translation node.

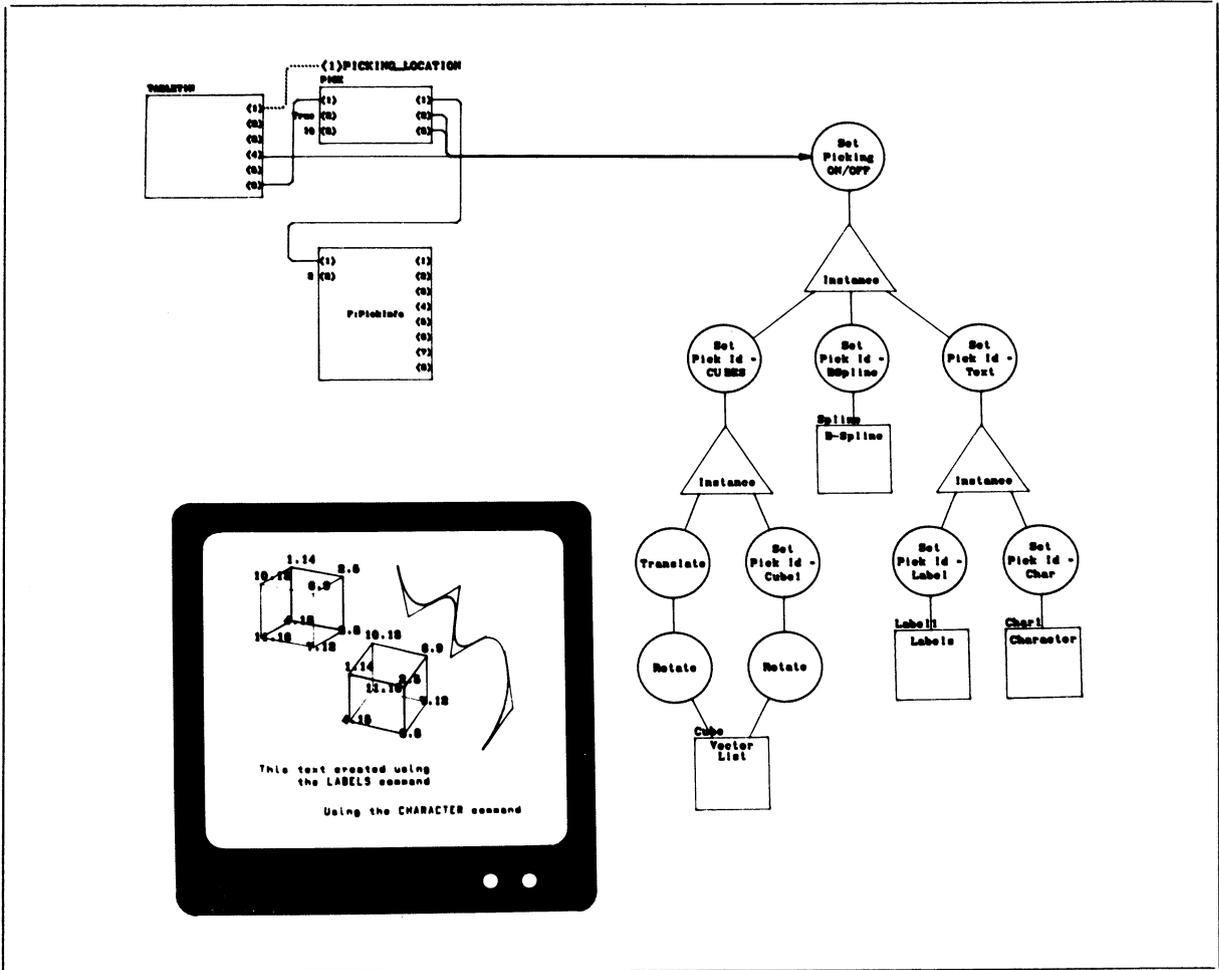
Function key F11 resets the display.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.



Program: PICKING

Typical Program Display



Abstract

This program shows graphically how picking can be performed on a vector list, a curve, a character string, or a label in a labels block.

Picking requires nodes in a display tree to set picking on and off, and nodes to identify the object that was picked (picking identifiers or pick IDs). A picking network must also be built so that a pick can be performed with the data tablet and information about the picked object can be returned for programming purposes.

Abstract (continued)

In one viewport, a representation of the PS 300 screen is shown displaying two cubes, a B-spline curve, a character string, and two labels. In another viewport, the display tree for this group of objects is shown. A SET PICKING ON/OFF node heads the display tree. This node is connected to the picking network. When you pick one of the vectors in the cube or B-spline, one of the characters in the string, or one of the two labels in the block, the path traversed in the display tree will brighten, and the function F:PICKINFO will show on its outputs the information returned by the pick.

Programmed Functions

Control Dials

None

Function Keys

F11 - RESET
F12 - EXIT

Notes on Usage

The display tree shows that there are two requirements for an object to be a candidate for picking. The display tree must have a SET PICKING ON/OFF node that can be enabled, and the object must be identified with a pick ID.

The picking network consists of the initial function instances TABLETIN and PICK, the initial structure PICK_LOCATION, and an instance of the function F:PICKINFO.

As you move the pen over the tablet, notice that output <l> of TABLETIN sends X and Y coordinate values to PICK_LOCATION. This is positioning the invisible pick-box so that it is centered exactly where the cursor appears on the screen.

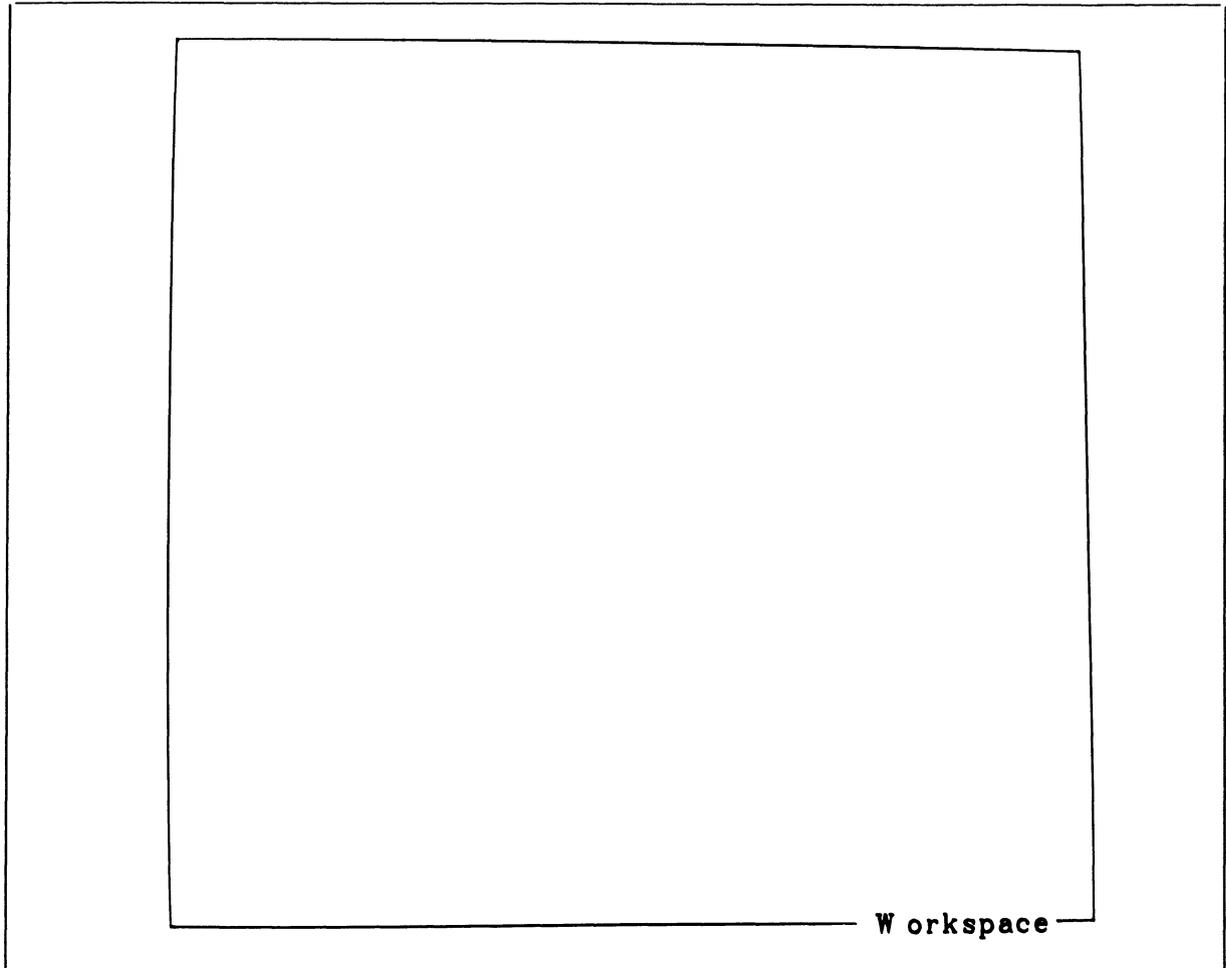
When a pick occurs, the path traversed in the display tree is momentarily brightened, and the outputs of F:PICKINFO show the information returned about the vector picked.

Function key F11 resets the display.

Function key F12 leaves the program and displays the Tutorial Demonstrations Menu again.

Program: **WORKSPACE**

Typical Program Display



Abstract

The work space is not truly a demonstration program, but a blank screen for you to use with the Tutorial Modules. Choose this selection from the menu when you are studying a module such as "Viewing" or "Character and Text Handling" that requires you to display and manipulate objects.

The work space is simply a border with the word "WorkSpace" at the bottom right.

Programmed Functions

Control Dials

None

Function Keys

F12 - EXIT

Notes on Usage

When you go to the work space, you will probably be entering commands to create and display structures as directed in the Tutorial Modules. If you create any other structures on your own, be aware that the names you assign may conflict with named entities in the Tutorial Demonstration files. We recommend that you avoid this by prefixing any name of your own devising with your initials or some other two-letter code.

Here is a reminder of the three modes of operation of the the PS 300 and the key sequences that enter those modes.

Command Mode	CONTROL/LINE LOCAL	Enter PS 300 commands at the "@@" prompt.
Interactive Mode	SHIFT/LINE LOCAL	Use the interactive devices to perform programmed functions.
TE Mode	LINE LOCAL	Enter commands on the host at the host prompt (PS 300 is emulating a host terminal).

When you leave the work space, enter the following command.

INITIALIZE NAMES;

This will clear all object names and function instance names you have created in Command Mode but will not affect names that are contained in the Tutorial Demonstration files. Remember that an INITIALIZE command is specific to a communications line. In other words, structures created through the keyboard in Command Mode can only be initialized with a local command from the keyboard, and structures transferred from the host can only be initialized with a command sent from the host.

If you use the INITIALIZE DISPLAY command, you will have to display the Tutorial Demonstration Menu and programs again. To do this, type the command

```
DISPLAY TUTORIAL_DEMOS;
```

when you are finished at the work space.

Use function key F12 to exit and return to the Tutorial Demonstrations Menu.



PS 300 TUTORIAL MODULES

The remainder of this volume consists of four tutorial modules which detail the fundamental concepts of PS 300 graphics programming. Each tutorial module covers a PS 300 programming concept or group of related concepts. Because each module builds on information contained in the previous module, it is highly recommended that you read the modules in the established order.

The following provides a capsule description of each module:

MODELING presents the first stage of graphics modeling, analyzing the model. This consists of breaking the model into interactive parts, organizing those parts into a hierarchy, and transforming the hierarchy into a PS 300 display tree.

PS 300 COMMAND LANGUAGE details how to translate the hierarchical display tree model into PS 300 command language.

FUNCTION NETWORKS I explains how to connect input devices to the model so you can interact with it.

VIEWING OPERATIONS 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.



MODELING

DESIGNING A CONCEPTUAL MODEL

CONTENTS

INTRODUCTION	1
OBJECTIVES	2
PREREQUISITES	2
DESIGNING AN ORGANIZATIONAL HIERARCHY	3
Exercise	6
DESIGNING A DETAILED DISPLAY TREE	9
Exercise	23
DESIGNING A COMPLEX MODEL	27
Exercise	31
Exercise	33
SUMMARY	53



MODELING

ILLUSTRATIONS

Figure 1.	Mechanical Arm	1
Figure 2.	All-Cylinder Robot and All-Sphere Robot	2
Figure 3.	Robot Made of Cylinders and Spheres	11
Figure 4.	Square and Corresponding Display Tree	13
Figure 5.	Diamond and Corresponding Display Tree	14
Figure 6.	Star and Corresponding Display Tree	14
Figure 7.	Transformed Star and Corresponding Display Tree	15
Figure 8.	Mechanical Arm With Proportions	16
Figure 9.	Cylinder Primitive for Mechanical Arm	16
Figure 10.	Cube Primitive for Mechanical Arm	17
Figure 11.	Hand Primitive for Mechanical Arm	17
Figure 12.	Mechanical-Arm Hand and Corresponding Display Tree	19
Figure 13.	Mechanical-Arm Upper Arm and Corresponding Display Tree	20
Figure 14.	Mechanical-Arm--Final Display Tree	22
Figure 15.	Sportscar	23
Figure 16.	Car Primitive -- Body	24
Figure 17.	Car Primitive -- Radial Tire	24
Figure 18.	Car Primitive--Snow Tire	24
Figure 19.	Tires Scaled and Rotated 180 Degrees	25
Figure 20.	Interaction Nodes for Tire	25
Figure 21.	Final Display Tree for Car	26
Figure 22.	Robot--Orientation	28
Figure 23.	Robot Sphere Primitive	29
Figure 24.	Robot Cylinder Primitive	29
Figure 25.	Robot--Proportions	30
Figure 26.	Robot--Body Pieces	31
Figure 27.	Robot--Informal Hierarchy	32
Figure 28.	Robot--Right Hand Display Tree	34
Figure 29.	Robot--Right Forearm Display Tree	35
Figure 30.	Robot--Right Arm Display Tree	36
Figure 31.	Robot--Shared Nodes for Hand	37
Figure 32.	Robot--Left Forearm Display Tree	37
Figure 33.	Robot--Display Tree for Two Arms	38
Figure 34.	Robot--Head Display Tree	39
Figure 35.	Robot--Upper Body Display Tree	41

MODELING

Figure 36.	Robot--Foot Display Tree	42
Figure 37.	Robot--Rotate and Translate for Foot	43
Figure 38.	Robot--Right Calf Display Tree	44
Figure 39.	Robot--Right Thigh Display Tree	45
Figure 40.	Robot--Shared Nodes for Foot	46
Figure 41.	Robot--Left Lower Leg Display Tree	47
Figure 42.	Robot--Left Leg Display Tree	48
Figure 43.	Robot--Lower Body Display Tree	50
Figure 44.	Robot--Completed Display Tree	52
Figure 45.	Windmill Display Tree #1	56
Figure 46.	Windmill Display Tree #2	56
Figure 47.	Correct and Incorrect Usage of Operate Nodes	57
Figure 48.	Sphere of Influence	59
Figure 49.	Instance Node Pointing to Three Data Nodes	60

One of the benefits of the PS 300 is the ease with which you create a model for display. Essentially, there are three steps to creating a model which can be manipulated interactively on the screen.

- The first step is to design the model on paper, taking into account what it will look like and how it will move.
- The second step is to write the PS 300 code using that conceptual model as a blueprint.
- The last step is to make the model interactive by connecting it to interactive devices. This module details the first step, designing a conceptual model.

Designing a conceptual model is in many ways like creating an outline or blueprint of your model. Like any outline, it allows you to organize your material in a logical, sequential manner. It also helps you design a complex model one step at a time.

Once the conceptual model is completed, it can be analyzed more easily for errors, repetitions, omissions, or flaws in logic because you can see its organization as a whole. Should you find an error, it is easy to correct at this stage of design.

Designing a conceptual model allows your attention to be focused on the problems of design. You need not be concerned with operating procedures for the PS 300 or with the PS 300 command language. In fact, once the model is designed, you already have the framework for the commands necessary to create that model in the PS 300.

Inherent in the design process of any model is the consideration of not only what the model looks like, but also what it does. This is because the way in which you interact with a model is built into the design as part of its organization. Not only can you interact with the model as a whole, you can manipulate different parts of it as well.

Consequently, the model is organized as a hierarchy of interrelated parts. Building this hierarchy entails:

- Knowing what the object to be modeled looks like
- Dividing the object into the pieces that comprise it
- Organizing these pieces according to movement or attributes.

The resulting hierarchy is a representation of the model's organization.

Once you have the model organized into the pieces that comprise it, the next step is to detail the steps that would be necessary to create each piece in the PS 300's world coordinate system.

To create a model in the world coordinate system, you perform a series of transformations (such as scales or rotations) on data primitives.

Using the hierarchy as your basis of organization for the pieces, you build each piece in the world coordinate system, performing whatever transformations are necessary; i.e., shaping a primitive into the desired piece, grouping the piece with other pieces according to their interdependencies, and then moving the pieces into their respective locations within the model.

Each of these steps is detailed in the *display tree* for the model. The modeled primitives are represented in the display tree as *data nodes*. The transformations you perform are represented in the display tree as *operation nodes*. There is a third type of node in display trees called an *instance node* which is used to organize and group the other two types of nodes. An instance node is placed wherever the display tree branches to more than one descending node.

When completed, the display tree represents all the information necessary to create the model, step by step. It even includes operation nodes which allow you to interact with the model as a whole or with any of its select pieces. The display tree can then actually be coded in the PS 300 via the PS 300 command language.

OBJECTIVES

This module details how to design a display tree for a model. You will learn how to:

- Design an Organizational Hierarchy
- Design a Detailed Display Tree
- Design a Complex Model

PREREQUISITES

Before reading this module, you should know basic computer graphics concepts, as developed in "Graphics Principles". You should also have completed the "Hands-On Experience" module.

DESIGNING AN ORGANIZATIONAL HIERARCHY

The first step in building a display tree is to design an organizational hierarchy for the model.

Before you can design an organizational hierarchy, however, you must know exactly what the model will look like. The model's dimensions and proportions may have been provided for you initially, or you may have to provide these yourself by drawing out a rough draft of the model on graph paper.

This rough draft can be used to divide the model into non-divisible pieces. The basis for this division depends on what you want to do with the model.

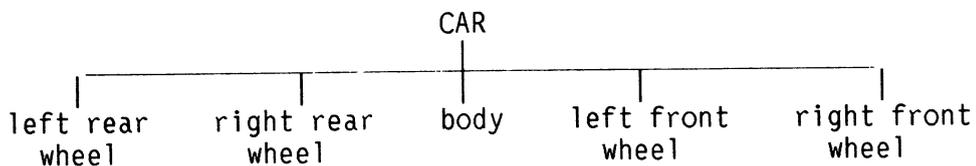
One basis for division might be movement--what pieces you want to move individually. You will also want to consider attributes which might differentiate pieces, such as color, or blinking, or level of detail. For example, you may want to show red fingers on a white hand. These attributes affect the way in which you design the model. It is much easier to make design allowances for them initially than to reconstruct the model later.

Of course it is possible that you may not want to differentiate all the pieces of a model. For example, suppose you are designing a sportscar, and the only moveable parts are the four wheels. In this case, the whole car body can be thought of as one piece. The model as a whole would then consist of five pieces:

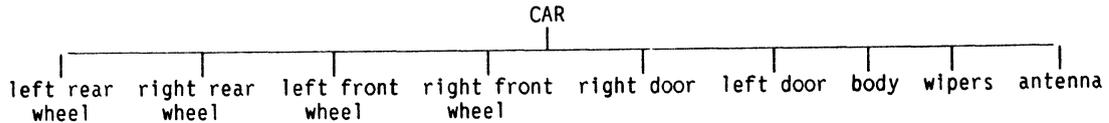
Pieces

1. right front wheel
2. left front wheel
3. right rear wheel
4. left rear wheel
5. car body

The resulting hierarchy would be:



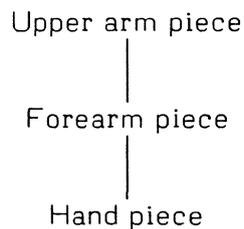
However, if you want the doors to swing open, the front windshield wipers to move, and an antenna to retract, each of these features is distinguished as a separate piece. The following hierarchy would then be:



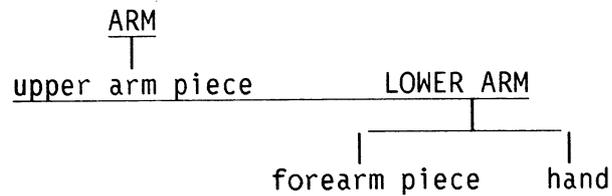
There are also times when you may want to interact with several pieces of the model collectively as well as individually or when the movement of one piece has a direct result on another piece. This kind of grouping or dependency affects the design of the hierarchy.

For example, if you were designing the arm of a robot, the arm could consist of three pieces: the hand, the forearm, and the upper arm. The hand piece can be moved individually. However, moving the forearm necessitates moving the hand, and moving the upper arm necessitates moving the both the forearm and hand. In this example, then, the three pieces have different degrees of independent movement.

Pieces are organized in a hierarchy according to this kind of *sphere of influence*. Those pieces which influence other pieces are above them in the hierarchy. So a simple hierarchy for the robot's arm might be:



To build the capacity for movement into this hierarchy, add two "grouping" names:



Grouping names are used when pieces act collectively. In this hierarchy, "LOWER ARM" is the grouping name used when the separate pieces, forearm and hand, move collectively. "ARM" is the grouping name used when you want to move the upper arm piece and the lower arm.

Grouping names make collective movement of pieces easier. Moving "ARM" is easier than moving each of the three pieces simultaneously. Moving "LOWER ARM" is easier than moving the forearm and hand pieces individually. Grouping names do not identify new pieces of the model--no ARM or LOWER ARM piece exists.

Exercise

Analyze the structure of the simplified mechanical arm in Figure 1 according to dependent and independent movement of pieces. Then organize these into a hierarchy accordingly. Use whatever grouping names are necessary.

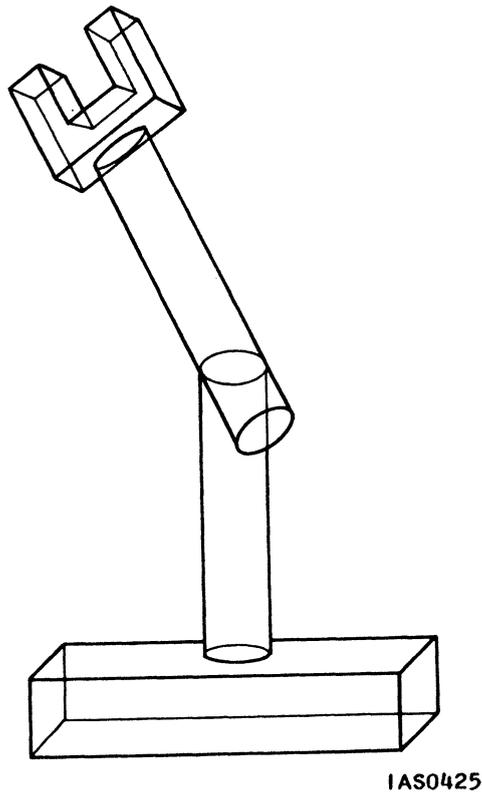


Figure 1. Mechanical Arm

The arm consists of a base, two jointed sections, and a hand. The base is fixed and cannot move. The whole arm can rotate at the base. The two arm pieces and hand are affected by this movement. The movement at the elbow affects the upper arm and hand only. Movement at the wrist only affects the hand.

So the pieces are:

Pieces

base
lower arm
forearm
hand

The pieces can be grouped accordingly: the first movement that affects more than one piece is above the elbow. Group the forearm and the hand together to form "UPPER ARM".

Pieces

Grouping Name

hand
forearm

UPPER ARM

The upper arm moves with the lower arm piece when the whole arm rotates at the base. Group these together to form the "ARM".

Pieces

Grouping Name

lower arm piece
upper arm

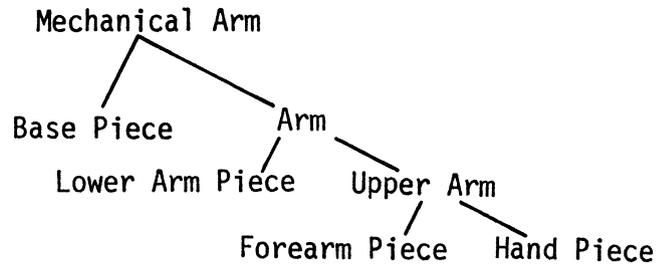
ARM

Finally, the base is a piece on its own. It is unaffected by the movement of the arm and the hand.

Piece

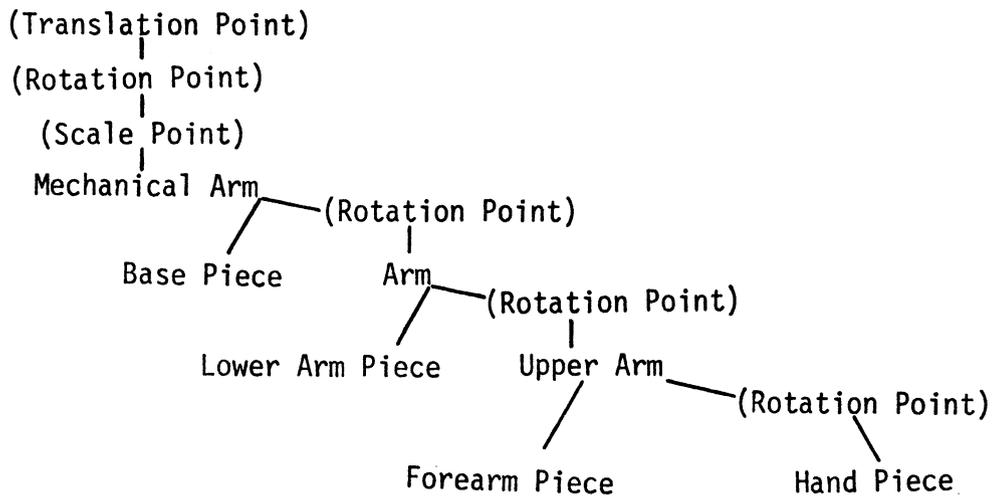
base

Once the pieces of the model have been identified and grouped, an informal hierarchy can be sketched out. The most inclusive pieces, in terms of influencing other pieces, are at the top of the hierarchy.



IAS0426

Since the pieces are divided according to how you can move them, it may be helpful at this point to note in the hierarchy those points where interaction will occur. So since the mechanical arm in the above hierarchy is divided according to rotation movements, note the places where interactions would occur in the above hierarchy. The interaction points are shown in parenthesis in the following hierarchy for Mechanical Arm.



IAS0427

DESIGNING A DETAILED DISPLAY TREE

The informal hierarchy is used as the organizational outline for the actual display tree you will design in this section. This is reflected in the way the model is actually built in the world coordinate system: pieces that are grouped collectively down a hierarchical branch are often built collectively along an axis of the world coordinate system.

The display tree conceptually represents each of the steps performed to build each piece that comprises the model. As you identify each step necessary to build the model, you will draw a corresponding node in the display tree. In other words, modeling the pieces and designing the display tree are simultaneous procedures.

The modeling steps themselves are:

- Shaping the organizational hierarchy pieces from primitives. (For information on how to create primitives, refer to Volume 3A, *Command Summary*.)
- Using modeling transformations to move pieces into position relative to other pieces that are grouped within the hierarchy.
- Adding interactions where needed.

First determine the primitives you want to use. These will depend on the kind of modeling you want to do.

- You may want an *iconic model*--one that looks as much as possible like the object it models. With this kind of model, each body piece is very distinctive and is modeled individually. For example, an iconic model of a man might have details such as facial features, hairstyle, fingernails, and so on. A very large vector list is usually required to provide this kind of needed detail.

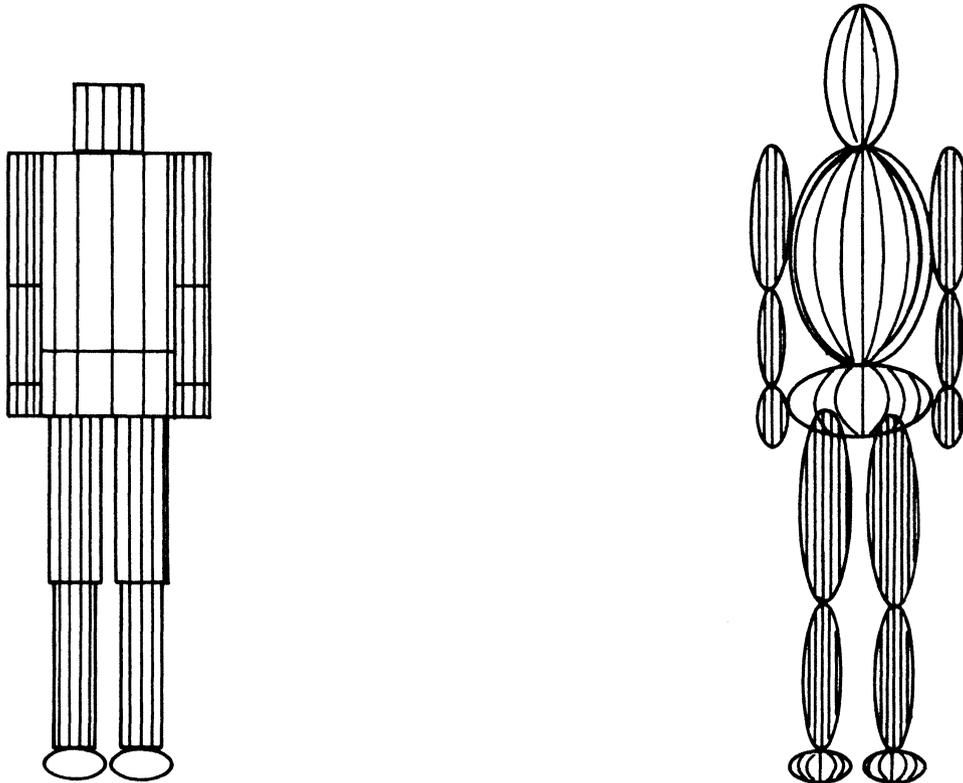
With iconic modeling, not only is more detail needed for each piece, but more pieces are needed. This requires a great many vector lists, a time-consuming and often difficult programming task.

- You may be able to use a less detailed *analog model*. Analog models minimize your task by eliminating unneeded detail. You only define pieces that are really needed. An analog model may be as useful as an iconic one for certain applications use an analog model if you only want to show movement, or relative position or size, for example.

Most graphics programming is a compromise between these two types of modeling. A model needs to be iconic enough to be recognizable, but analog enough to be useful.

For example, a robot model designed for movement might not require a great deal of realism. If there is no need to differentiate individual fingers on the hand, it could be designed as an oval shape. The vector list to create this would be simpler than one to create a detailed hand with fingers. Both the left and right hands could be modeled from this vector list.

In the same way, simpler primitive shapes, like cylinders and spheres, could be used repetitively by many different pieces of the robot. For example, a robot could be made from nothing but cylinders by defining a cylinder primitive and then transforming that shape to create each body piece. Changing the primitive from a cylinder to a sphere would change the appearance of the analog model.

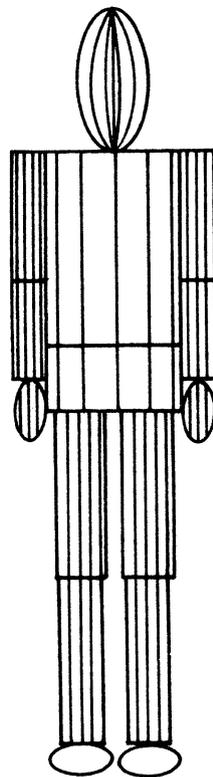


IAS0428

Figure 2. All-Cylinder Robot and All-Sphere Robot

However, both models illustrate movement, position, and size in exactly the same way. Both use only one primitive, easing the programming task immensely.

The model in Figure 3 is only slightly more complex. It uses a sphere primitive for the head and hands, and a cylinder primitive for the rest of the body pieces. Two primitives are required, but the result is more aesthetically pleasing.



IAS0429

Figure 3. Robot Made of Cylinders and Spheres

Clearly, there are numerous ways to model an object. In any modeling application, there is flexibility in deciding how realistic the model will be and how many primitives will be required.

Once you have established what primitives the model will be shaped from, determine the actual dimensions and placement of the primitives in the world coordinate system. To do this, it is helpful to draw the model to scale on graph paper. The model serves as a visual aid as you determine how much to enlarge, reduce, or reshape primitives.

The dimensions of the primitive are often determined arbitrarily and are usually small, whole numbers. For example, it is easy to work with a sphere with a radius of one. If you need an oval four units high and two units wide, scale the sphere by 1 in X and 2 in Y.

When determining the initial position of primitives, consider where it is easiest to define a primitive, and what position it needs to be in most of the time to form pieces of the model. It is usually easiest to work with primitives located at the origin of the world coordinate system. One reason is that rotations take place around coordinate system axes. To apply rotations correctly to a primitive, pieces of a model often need to be centered about, sit on, or hang from an axis.

Any model you create will be defined by coordinate system locations. When positioning the model, it is usually preferable to construct a model near the origin. One reason is that the initial view of the PS 300 world is centered on the origin. Another reason is that it is often easiest to establish symmetry if the model is centered about the X, Y, and Z axes. Finally, because rotations and scalings are performed relative to the origin, building the model there is easier.

Now that you know the dimensions of your model, its position in the world coordinate system, and the primitives which compose it, you are ready to design the model's display tree.

A display tree represents several kinds of information. First, it includes the step-by-step information necessary to create the model in the world coordinate system. Second, it includes the capability to differentiate parts of the model by attributes, such as color, or by movement. Finally, it includes the capacity for interaction with part or all of the model.

PS 300 display trees consist of up to three types of nodes.

- Primitive data, the "building blocks" for the model, are represented in the diagrams by square data nodes. Specifically, these nodes describe the collection of points, lines, and characters that define primitive data. Data nodes are always terminal nodes in a display tree.
- Any operations (such as scaling and rotation) which are performed on an object are represented in the display tree by a circle. These nodes are called operation nodes. An operation node can point to no more than one node below it.

Operation nodes are used in two ways: for modeling and for interaction. Modeling operations are performed strictly to shape the "building blocks" of a model and move them into place. Interaction operations allow you to interact with a model. Any operation node can be either a modeling or an interactive node, or both kinds, depending on how it is used. In this module, interactive nodes are represented by a double circle; modeling nodes by a single circle.

Operation nodes are also distinguished by the fact that once they have been coded into the display tree, you can enable or disable them interactively. (For more information on this, refer to the "Function Networks I" module.)

- Instance nodes join one or more subparts, or hierarchical branches, into a whole, nameable part. Instance nodes are represented by a triangle.

There is a special group of operation and data nodes that represent the commands that allow you to display labels and character strings. For details on these, refer to the "Text Modeling" module.

In the "Hands-On Experience" chapter, each step you took to create and display the Star can be represented by one of the three types of nodes described above.

First, you created the square using a vector list:

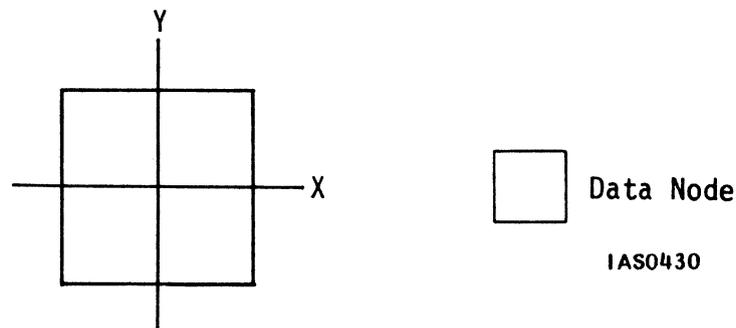


Figure 4. Square and Corresponding Display Tree

Then you displayed a rotated version of that square--Diamond:

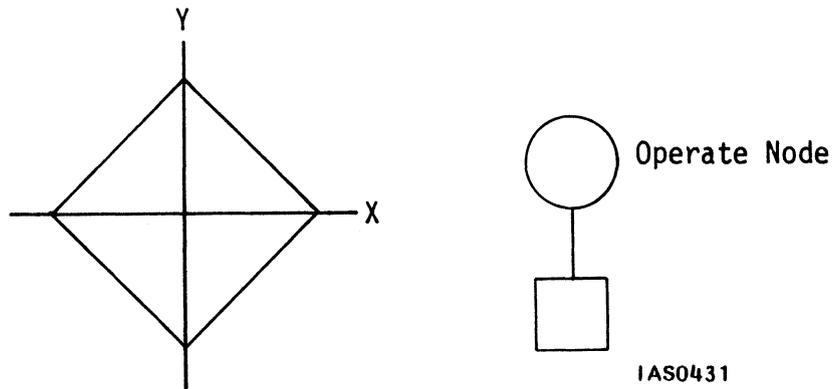


Figure 5. Diamond and Corresponding Display Tree

The diamond and the square were then linked together to form a star:

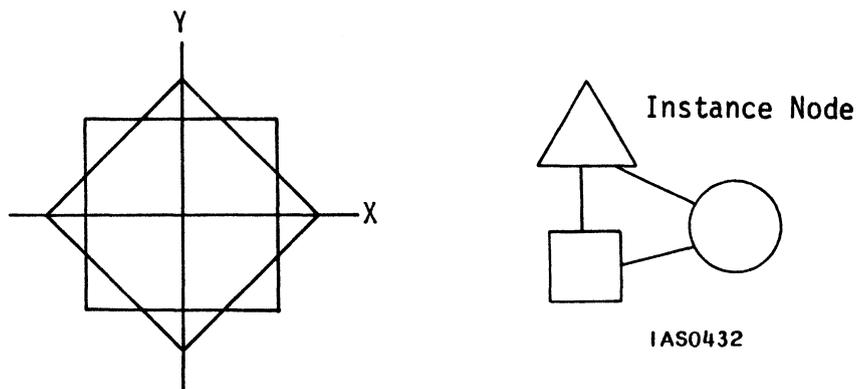


Figure 6. Star and Corresponding Display Tree

The other operations you perform on the star, scaling and translating, are represented in the display structure by these nodes.

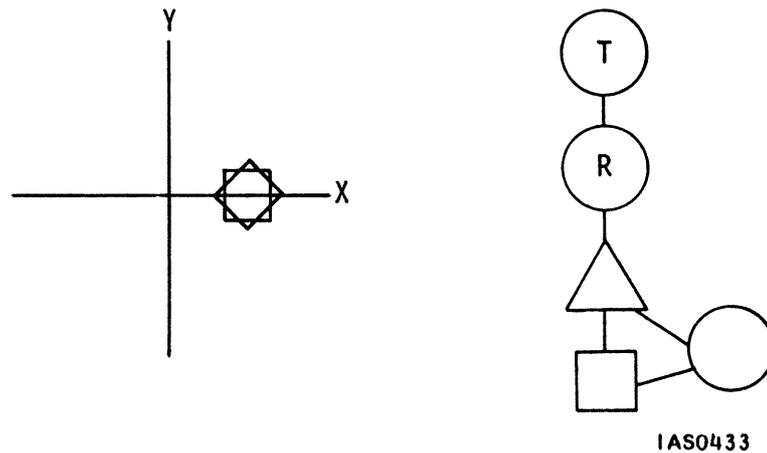


Figure 7. Transformed Star and Corresponding Display Tree

Display trees are designed beginning with the lowest nodes on the tree, the data nodes, and moving consecutively up the tree through each operation that is performed. This assures that the data are modified in the proper order by the PS 300. (Operations such as scale, rotate, and translate are all performed using matrix multiplication. The non-commutativity of matrices means you must order transformations carefully.)

The remainder of this section describes, step by step, how to design a display tree for the mechanical arm used in the first exercise. The mechanical arm and its dimensions are shown in Figure 8.

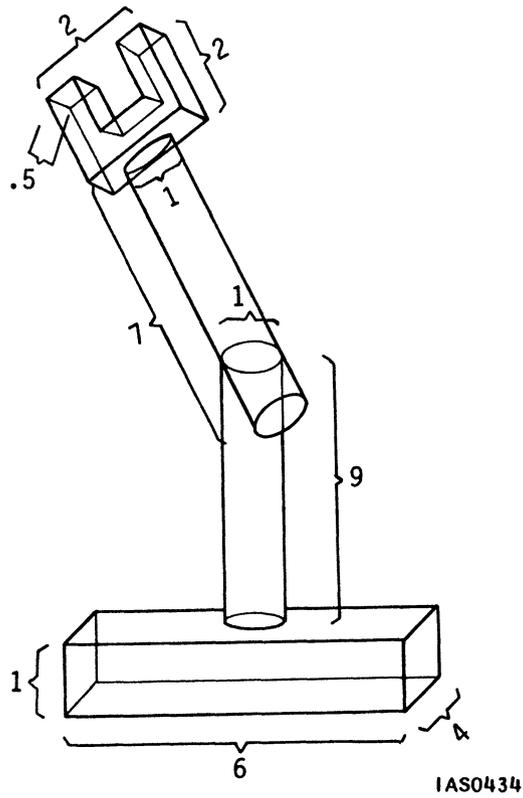


Figure 8. Mechanical Arm With Proportions

The mechanical arm is designed using these primitives:

1. A unit cylinder with its base on the XZ plane, centered on the positive Y axis, with a radius of 1 and height of 1. (Figure 9)

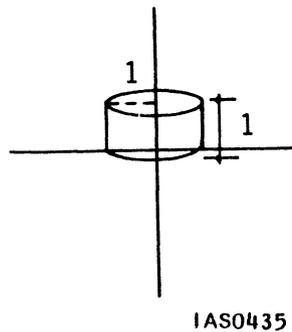


Figure 9. Cylinder Primitive for Mechanical Arm

2. A unit cube with its base on the XZ plane, centered on the positive Y axis, with a length, height, and width of 1. (Figure 10)

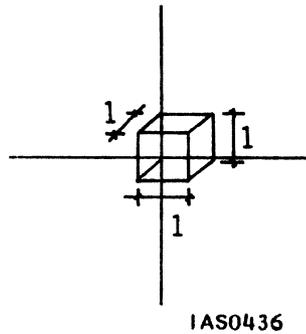


Figure 10. Cube Primitive for Mechanical Arm

3. A primitive consisting of lines which form the hand with its forks pointing up, with the base on the XZ plane, centered on the positive Y axis with a height of 2, width of 2, and depth of .5. (Figure 11).

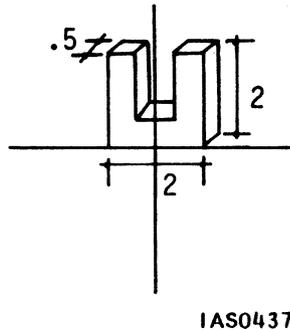
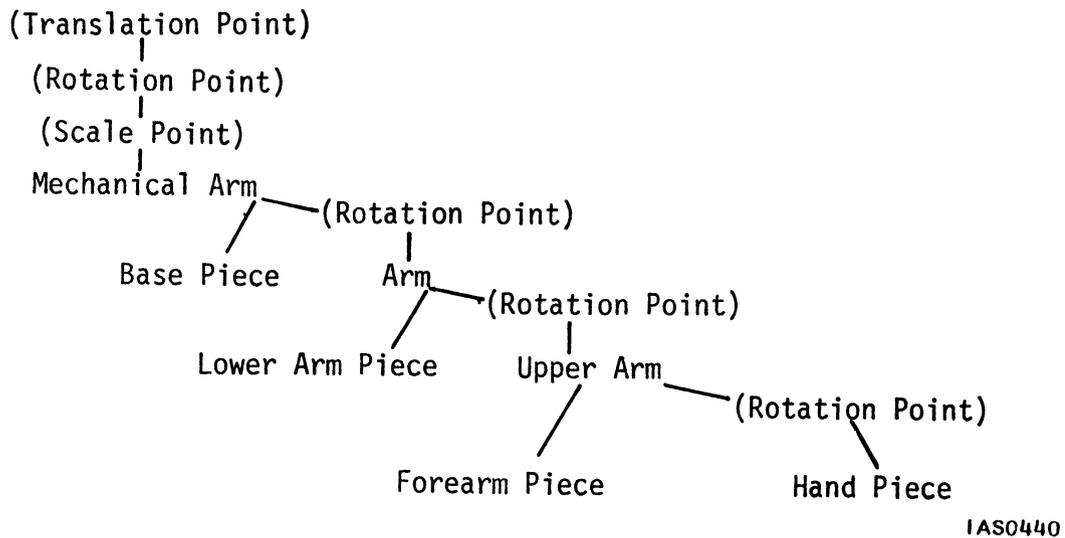


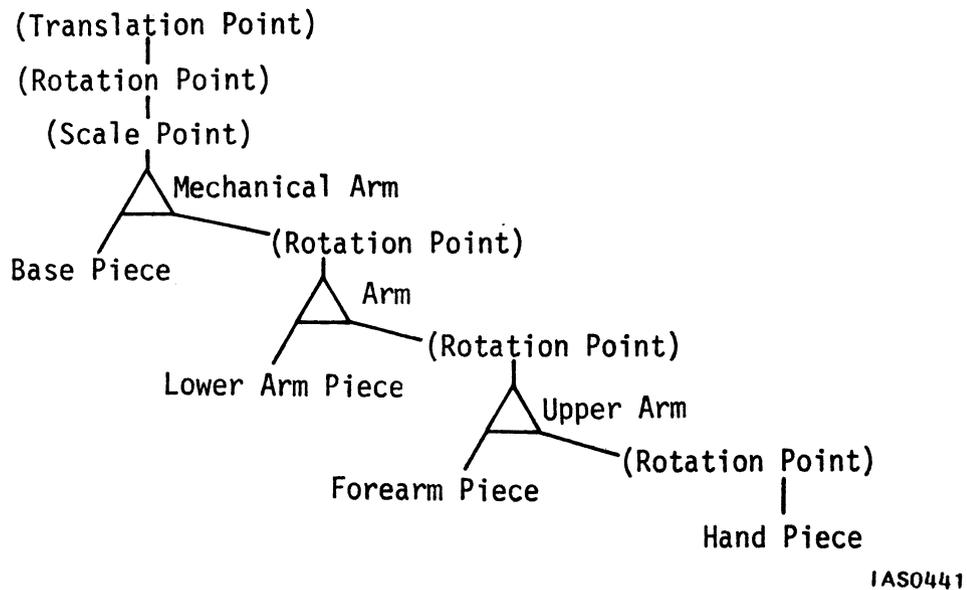
Figure 11. Hand Primitive for Mechanical Arm

As for initial position in the world coordinate system, the mechanical arm will be placed with its base on the XZ plane, centered on the positive Y-axis. It will be easiest to build the model up the Y axis.

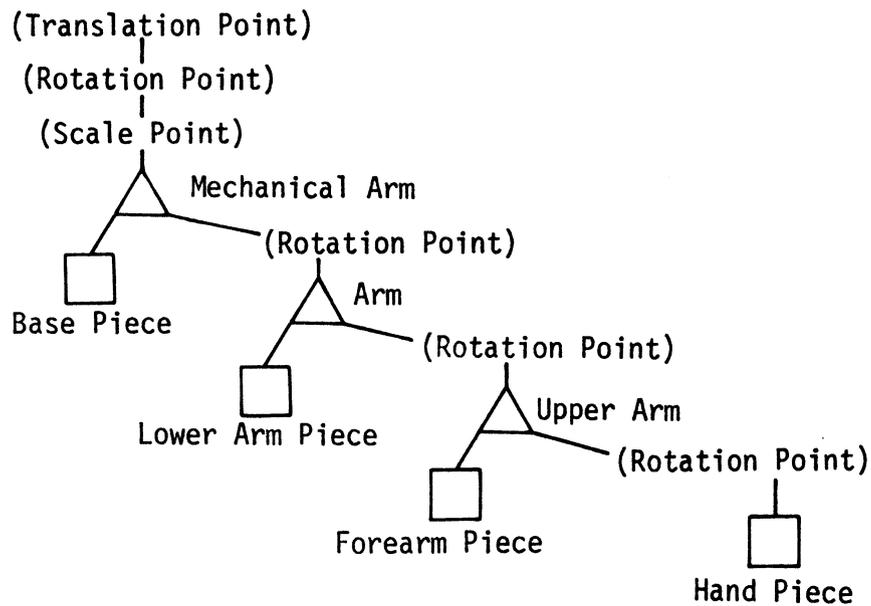
As you model the mechanical arm, create the corresponding display tree using the hierarchy as the basis:



Instance nodes are used to group other nodes in the display tree. The hierarchy branches at three places: where the mechanical arm is divided into a base and arm, where the arm is divided into a lower arm piece and an upper arm, and where the upper arm is divided into a forearm and a hand. The instance nodes are placed accordingly:



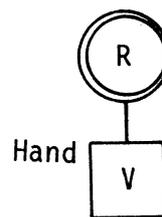
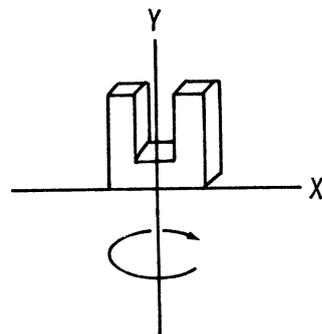
Next remember that all terminal nodes, those which define primitives, are represented with data nodes:



IAS0442

Finally, working up from the bottom of the display tree, we will detail the steps to model each of the primitives in the world coordinate system. Begin with the hand.

The primitive for the hand (data node) is designed so the hand is already the proper size and in the proper place, so no scaling or translating is necessary. According to the hierarchy, however, a rotation node is needed to allow rotation at the wrist. See Figure 12.



IAS0438

Figure 12. Mechanical-Arm Hand and Corresponding Display Tree

Now that the hand is built, it should be positioned so it can be grouped with the forearm piece. You might be tempted to translate the hand to its final position in the model, build the forearm piece, and then translate that piece into its final position. But if you do this, when you group these two pieces into upper arm and then rotate that upper arm, both pieces will "orbit" the axis rather than rotate at the elbow. For proper rotation of the upper arm, the hand must be grouped with the forearm and both rotated while the forearm rests on the axis. (For more information about rotation, refer to the "Graphics Principles" section 2.)

So next, translate the hand up the Y axis the length of the forearm piece, 7 units in +Y. Then build the forearm at the origin by scaling the cylinder (1,7,1), and group both the forearm and hand together as upper arm. Now if you apply a rotation to upper arm, it will rotate properly. See Figure 13.

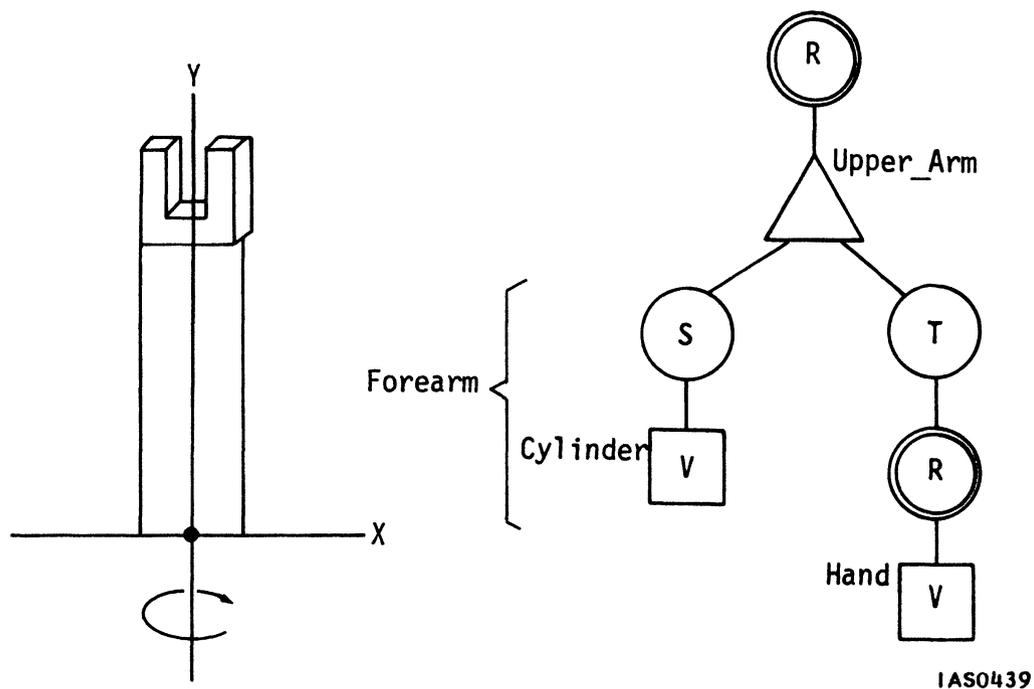


Figure 13. Mechanical-Arm Upper Arm and Corresponding Display Tree

A similar procedure is used to build the remainder of the arm. To assure proper rotation, move the upper arm up the Y axis the length of the arm, build the lower arm piece, and THEN apply the rotation to the whole arm. (Pieces do not have to be grouped just along the Y axis. If it is easier to do so, you can build the pieces along the X or Z axis.)

The modeling steps can be summarized as follows:

1. Move Upper Arm (forearm and hand pieces) 9 units up the +Y axis to make room for the lower arm piece.
2. Scale the cylinder to create the lower arm piece (1,9,1).
3. Group the lower arm and upper arm to form Arm.
4. Apply a rotation to Arm.
5. Scale the cube to create the base (6,1,4).
6. Allow for interactive manipulation of the whole mechanical arm (rotation, translation, scaling) with three interactive nodes.

The final display tree is shown in Figure 14.

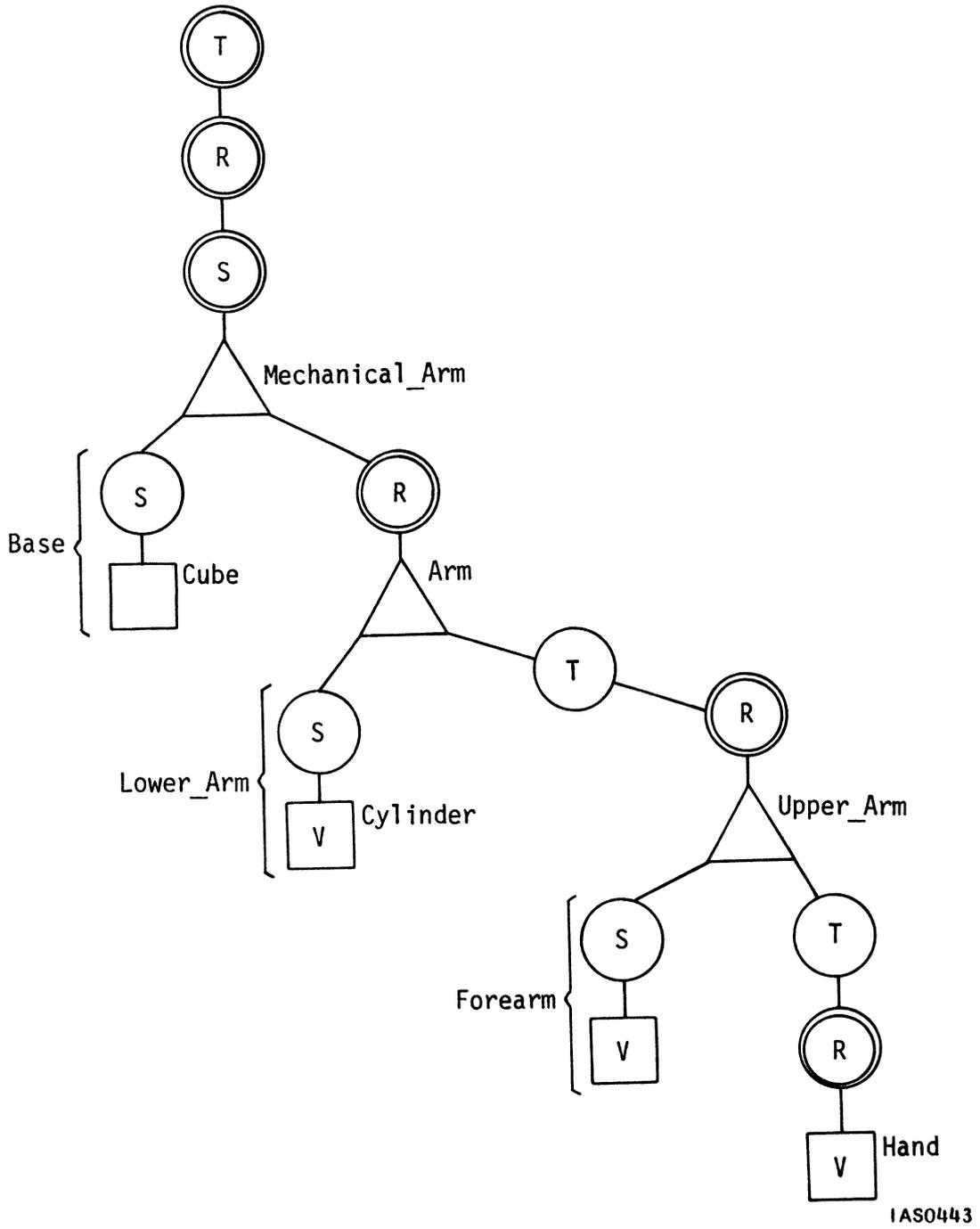


Figure 14. Mechanical-Arm--Final Display Tree

Exercise

Design the display tree for the sportscar in Figure 15.

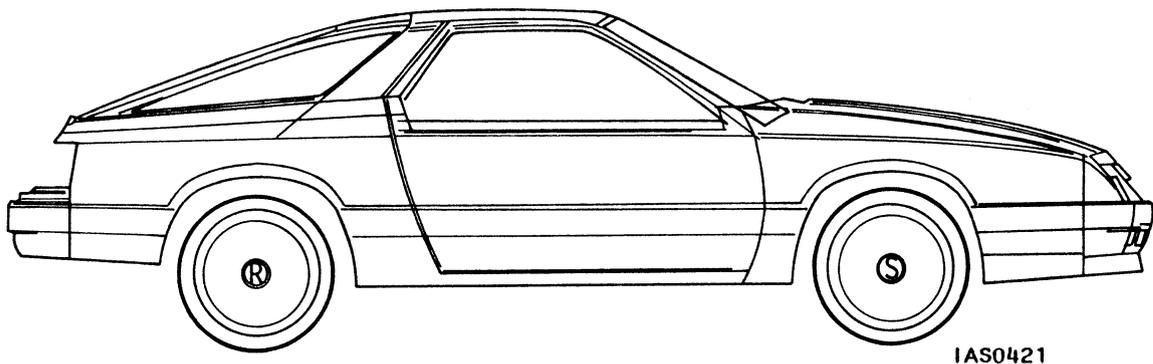
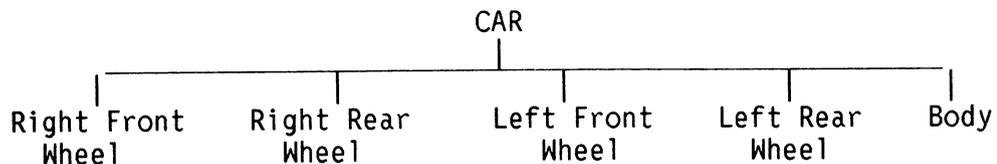


Figure 15. Sportscar

Include the capacity for movement in the four wheels (rotation) and for movement of the car as a whole (rotation, translation).

First design an informal hierarchy. Only five parts are needed for this car: the four wheels and the body. Because the body has no moving pieces, the whole thing can be thought of as one piece. The hierarchy is:



Next, model the primitives and create the display tree concurrently. There are two sets of tires: snow tires on the back and radials on the front. This means three primitives: a vector list for the body of the car, one for the snow tire, and one for the radial tire. These will be represented in the display tree by three data nodes, as shown in Figures 16, 17 and 18.

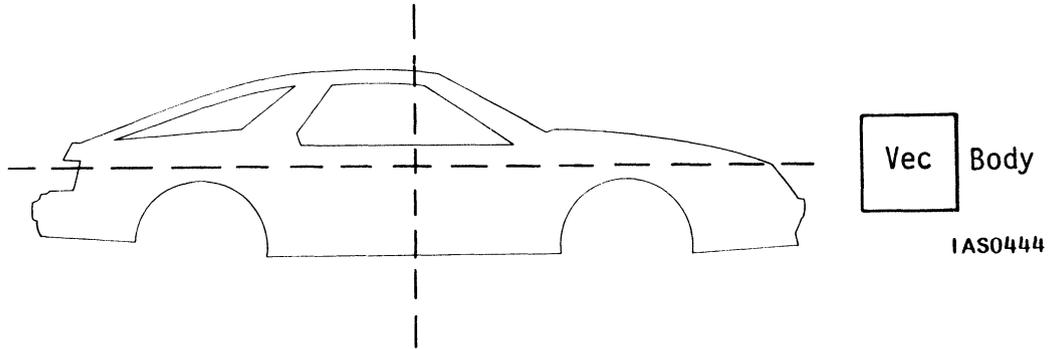


Figure 16. Car Primitive--Body

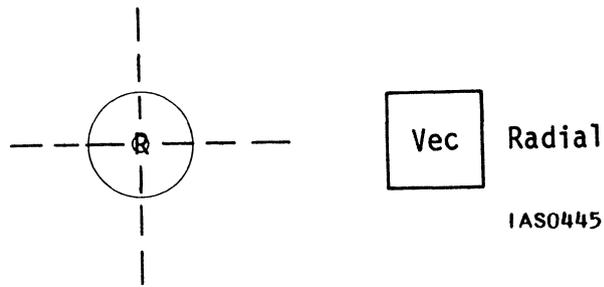


Figure 17. Car Primitive--Radial Tire

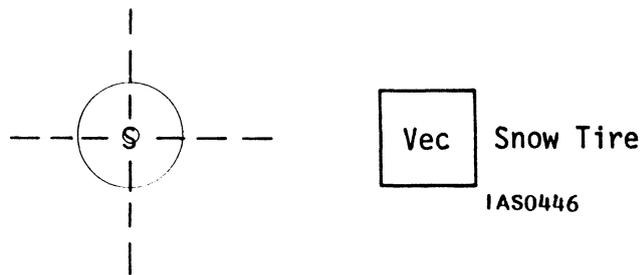


Figure 18. Car Primitive--Snow Tire

The wheels are scaled to fit into the wheelwells of the car. This means the data node for each type of tire has a scale applied to it.

Each wheel also has a hubcap on one side. Rotate the two tires on the left of the car 180 degrees about Y so that the hubcaps face out (Figure 14).

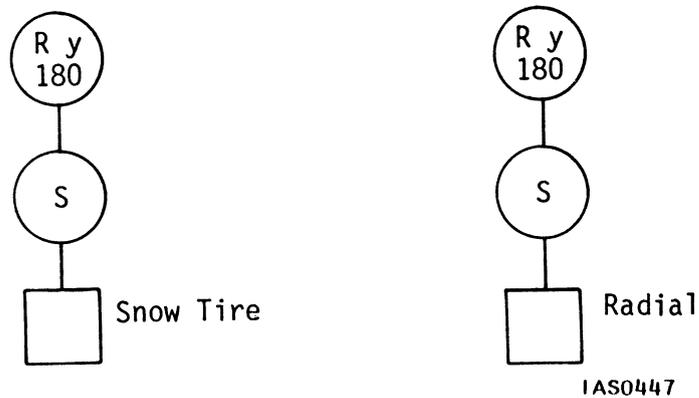


Figure 19. Tires Scaled and Rotated 180 Degrees

To allow for rotation of all four tires around the Z axis, insert interaction nodes which can accept values from an input device or host computer via a function network. Since these rotation values will subsequently be changed interactively, they can be 0,0,0 for now (Figure 20).

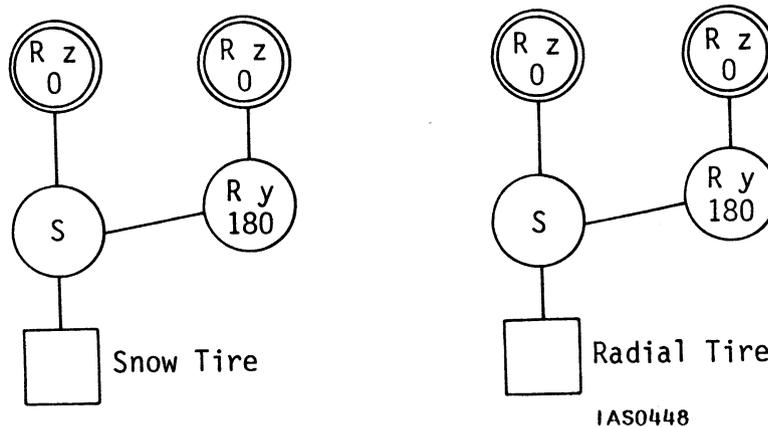


Figure 20. Interaction Nodes for Tire

The order of the operations is important here. Moving up the display tree branch, the interaction node applies AFTER the modeling rotate node has been applied to the data (Rot Z is above Rot Y 180). If you build the interaction node first and then turn the left tire out 180 degrees, it rotates in the wrong direction.

After all the rotations are applied, translate each wheel from the origin to its proper position on the car. Then group all pieces together as Car.

Two kinds of movement might be desirable for the car as a whole: rotation and translation. These nodes are placed at the top of the structure.

The final display structure is shown in Figure 21.

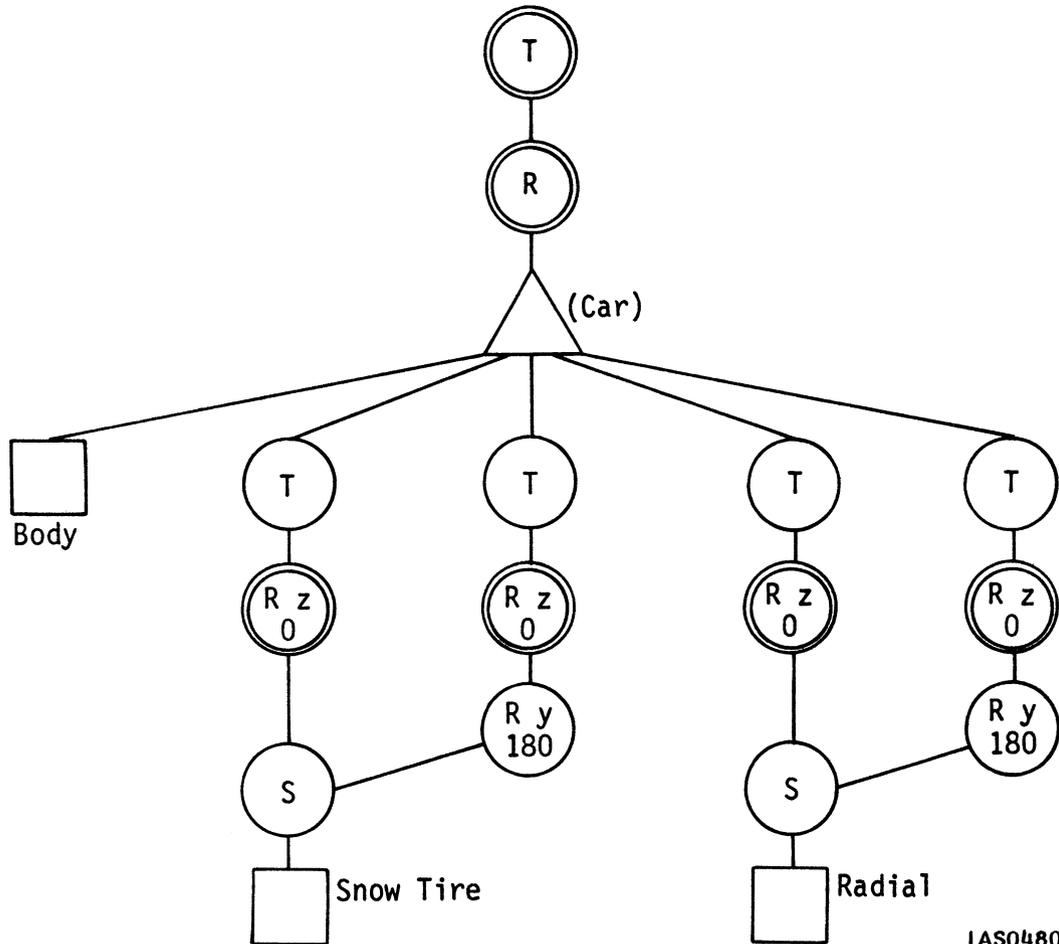


Figure 21. Final Display Tree for Car

DESIGNING A COMPLEX MODEL

This section details the steps necessary to design a complex model, an anthropoid robot. The exercise illustrates the importance of interaction nodes, (in this case, for movement) in the design of display trees. It also allows you to deal with specific kinds of modeling problems, giving you practical experience and allowing for some helpful generalizations about good programming techniques.

Design Robot with movement in mind. Robot moves at the joints: waving, swinging his arms, nodding, bowing, kicking, and so on. All of these movements are rotations of one kind or another about the bases of different body parts such as the waist, shoulder, and wrist.

Robot should look like the one shown in Figure 22. Notice his initial orientation--what position his limbs are in and where he's located in world space coordinates. To make the design task easier, Robot is placed symmetrically about the Y axis with his center at the origin.

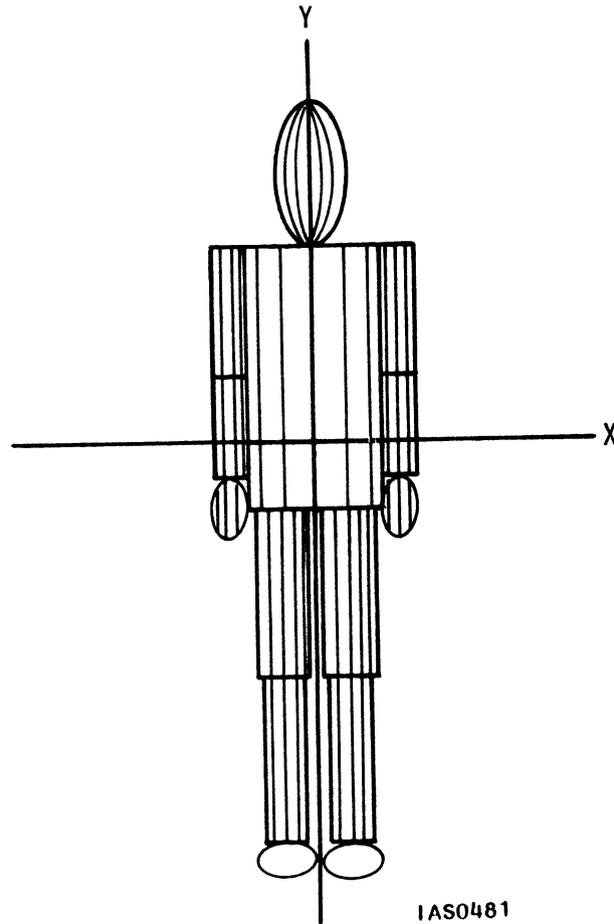


Figure 22. Robot--Orientation

Robot's body pieces consist of two primitives: a sphere for the head and hands, and a cylinder for the remaining body pieces. These two primitives are defined below. Note that they are three-dimensional objects requiring (X,Y,Z) coordinate values.

1. A unit sphere centered at the origin with a radius of 1 (Figure 23).

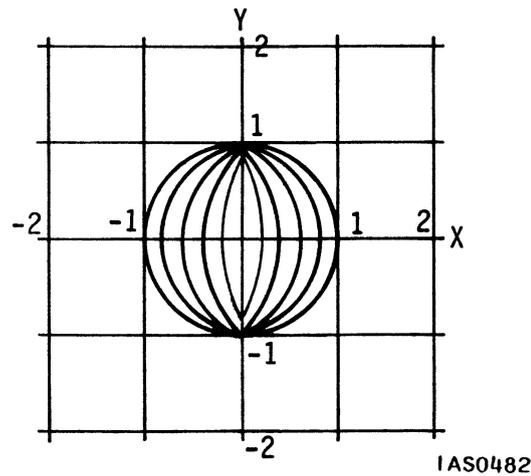


Figure 23. Robot Sphere Primitive

The sphere is centered at the origin because it is easier to calculate the shape in this position. Also, from this central location, the sphere can be translated along axes easily. It will need to be translated up the Y axis when modeling the head and down the same axis when modeling the hands.

2. A unit cylinder with the proportions (2,2,2), hanging on the XZ plane (its top resting on the X axis), centered on the negative Y axis (Figure 24).

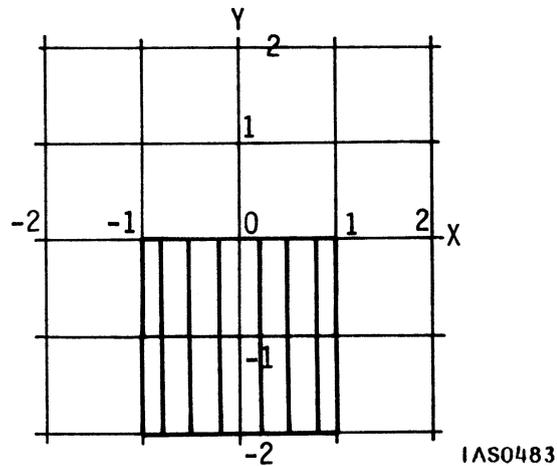
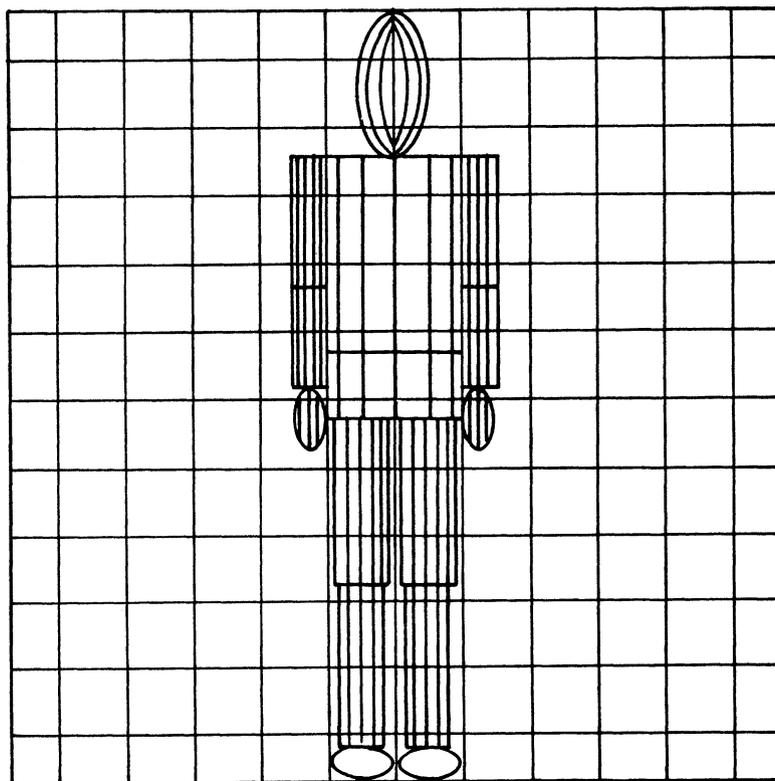


Figure 24. Robot Cylinder Primitive

The cylinder could have been centered about the origin as the sphere was. However, it has been created down the Y axis for a reason. Almost all the body pieces that depend on the cylinder rotate from "above". For example, the lower arm rotates at the elbow and the upper arm at the shoulder. If the cylinder were placed at the origin, each time a body piece was created the cylinder would have to be translated down the Y axis for this rotation to be applied "above" the piece.

Placing the primitive below the origin initially, then, saves separately coding a translate node for each piece you create. This is a good example of creating your primitive to suit your design goals.

Once the proportions for primitives have been established, use these to determine the exact size of Robot's body pieces in the world coordinate system. For example, as shown in Figure 25, Robot's head is twice as tall as it is wide. This means the sphere primitive will have to be scaled (1,2,1) to make the head.



IAS0484

Figure 25. Robot--Proportions

Exercise

Design a hierarchy for Robot. Joints will be at the wrists, elbows, shoulders, ankles, knees, hips, waist, and neck.

Robot is composed of the fifteen individual body pieces shown in Figure 26.

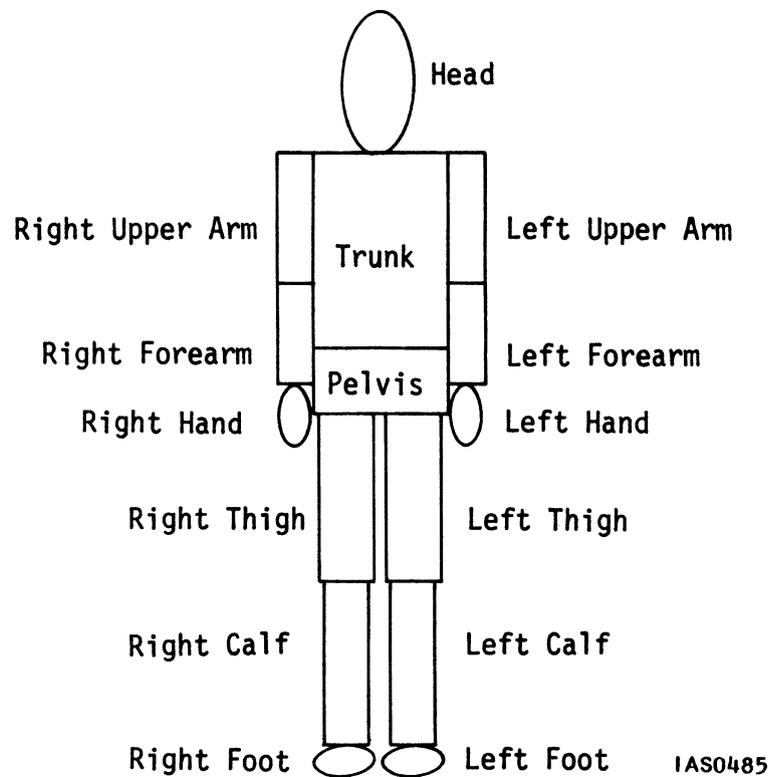
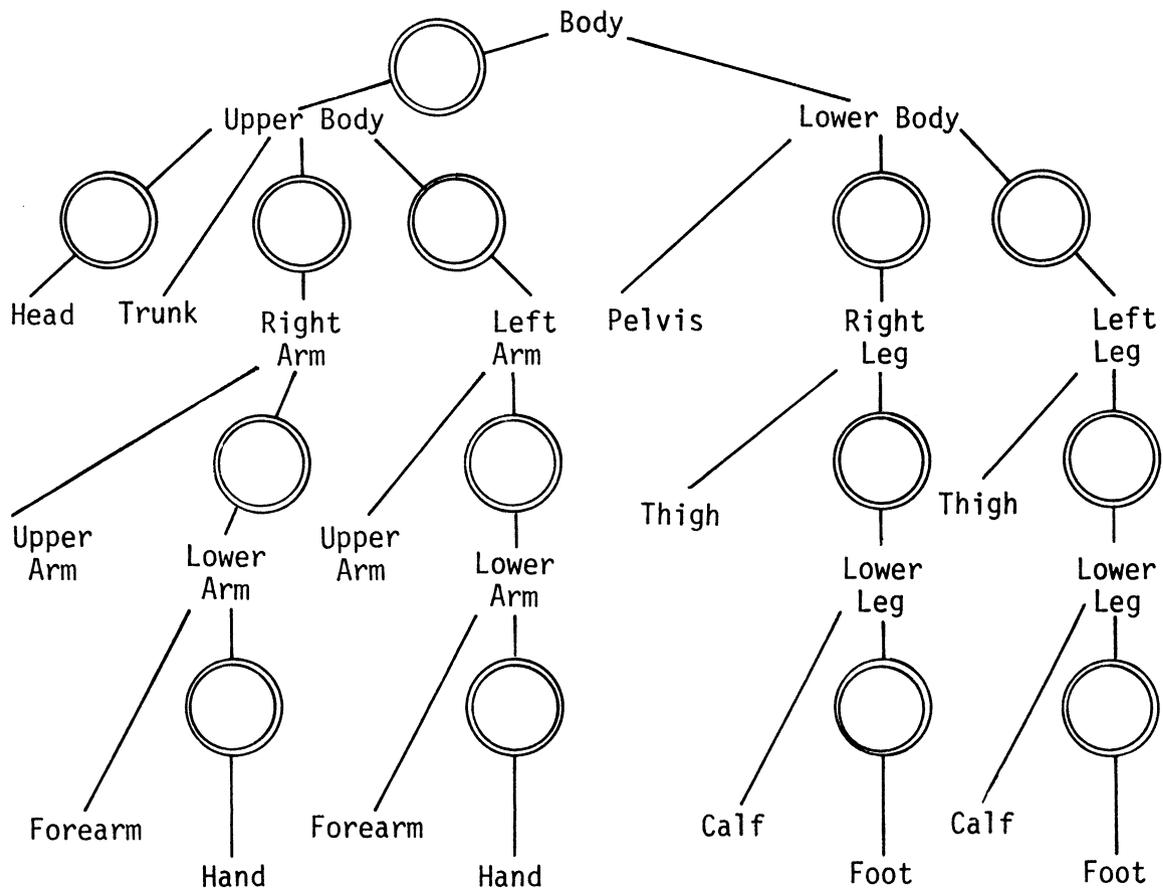


Figure 26. Robot--Body Pieces

The pieces should be organized so that rotating a joint causes all appendages affected by that joint to rotate. For example, rotating "upper body" to make Robot bow should cause the trunk, head, and arms to rotate. Though naming may differ somewhat, the hierarchy of named parts should basically look like the one in Figure 27.



1AS0486

Figure 27. Robot--Informal Hierarchy

Notice the hierarchy includes additional "grouping names" (lowerleg, rightarm, upperbody). As with previous examples, interactive points have been added to the hierarchy where the joints will rotate.

Exercise

Use the informal hierarchy as a guide for the display tree. "Grouping" names can be represented by instance nodes. Data nodes will be terminal nodes at the end of the hierarchical branches. Placement of interactive nodes has already been established. Placement of modeling operation nodes should be carefully worked out as you model each piece in the world coordinate system.

As you design the display tree, make note of where the body pieces for Robot's limbs can be shared by the left and right body pieces. Consider the feet, calves, thighs, forearms, upperarms, and hands. Right and left pieces can share nodes up to the point where the nodes serve to distinguish the two (separate rotate and translate nodes).

Sharing must be done carefully, in a way that allows parts of the model that require individual movement to remain independent. In any given display structure, there are many different ways to share nodes.

The following details the modeling steps and corresponding display tree for Robot.

Creating the Right Hand

Scale the sphere to create the elongated shape of the hand (.5,1,.5). Translate the hand down the Y axis (0,-1) so a rotation can be applied at the origin. Insert the rotate interaction node to simulate the wrist so the hand can "wave". For all interactive nodes, specify a zero value initially because values can be supplied later from interactive devices or a host.

Notice that although you have placed only one rotate node for articulation here, the wrist rotates around three axes (X, Y, and Z axes). (The "Function Networks I" module describes how to allow for rotation in three dimensions with one 3x3 matrix node (rotation node).)

Since the hand must be grouped with the forearm piece, translate the hand down the Y axis the length of the forearm piece (0,-3) rather than translating it into its final position in the model. Then when you apply a rotation to the lower arm, both the hand and forearm will rotate properly "from the elbow," rather than "orbiting" the axis.

Figure 28 illustrates the series of transformations that create the hand from the sphere primitive.

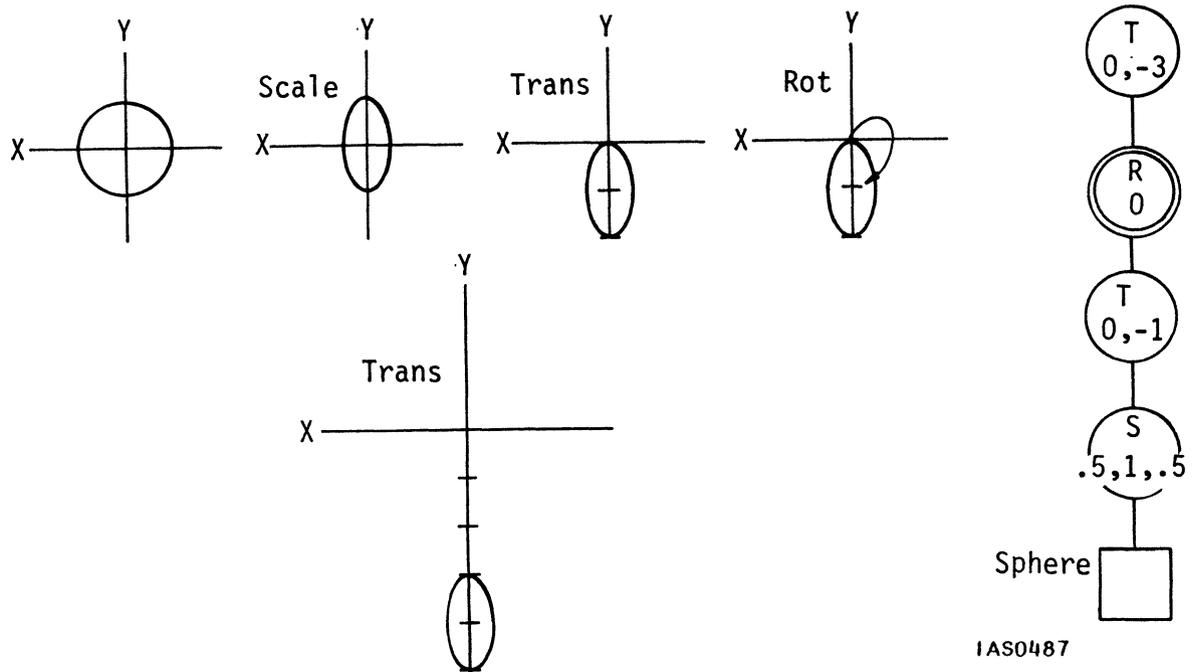


Figure 28. Robot--Right Hand Display Tree

Creating the Right Forearm

The forearm piece is created by scaling the cylinder to the proper size (.5,1.5,.5). Scaling the forearm places it in the proper position to meet the hand; there is no need to translate it.

When you rotate the right forearm from the elbow, you want the entire lower arm (the forearm piece and hand) to rotate. To do that, define right forearm to be an instance of forearm and hand.

Insert a rotate node above the lower arm instance to move lower arm at the elbow. Then translate lower arm down the Y axis--this time the length of the upperarm piece (0,-4).

Figure 29 shows the series of transformations that create the lower arm.

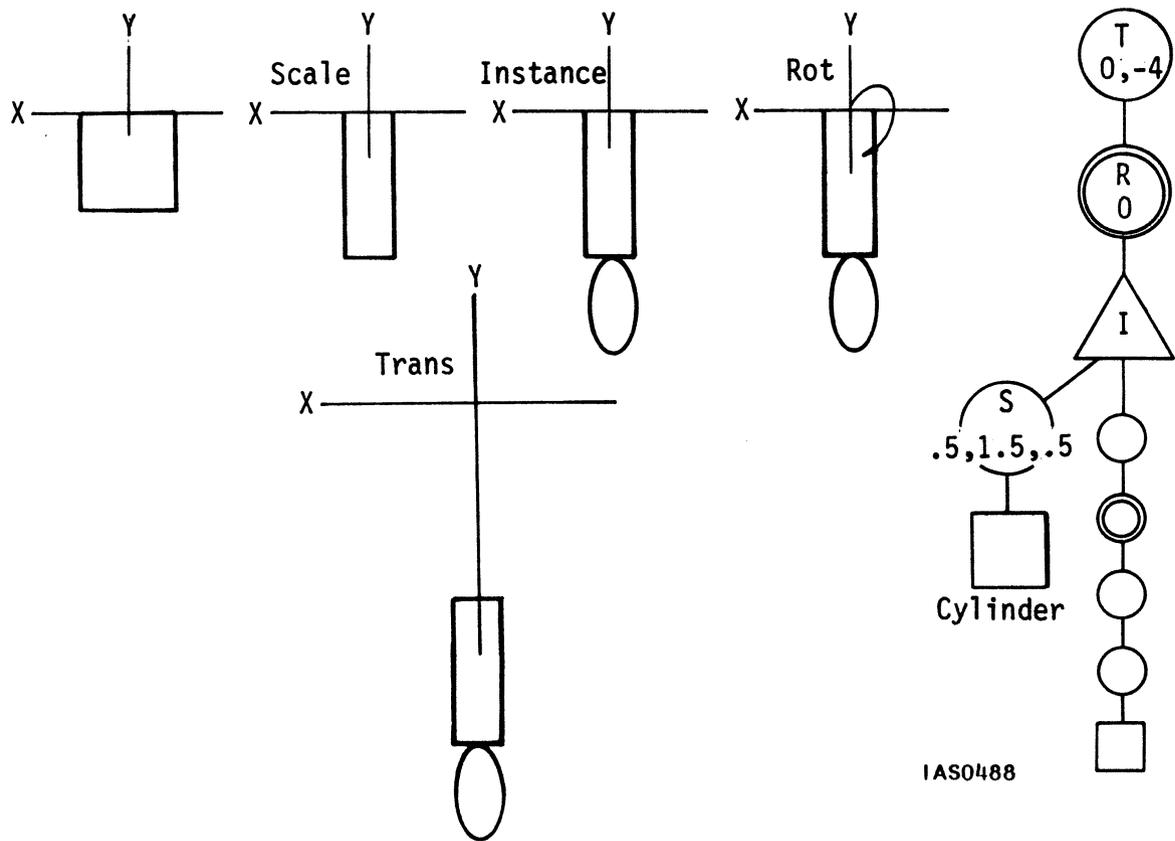


Figure 29. Robot--Right Forearm Display Tree

Completing the Right Arm

Build the upper arm piece so you can link it with the lower arm to make the entire arm. First scale the cylinder primitive (.5,2,.5), then link the upper arm piece and forearm together using an instance node. Allow for manipulation by including a rotate node above that. Then translate the arm out to its final place in Robot (-2.5,6). This translation value is the exact X,Y coordinate location of the shoulder (the rest of the arm "hangs" below that point).

Figure 30 shows the series of transformations that create the right arm.

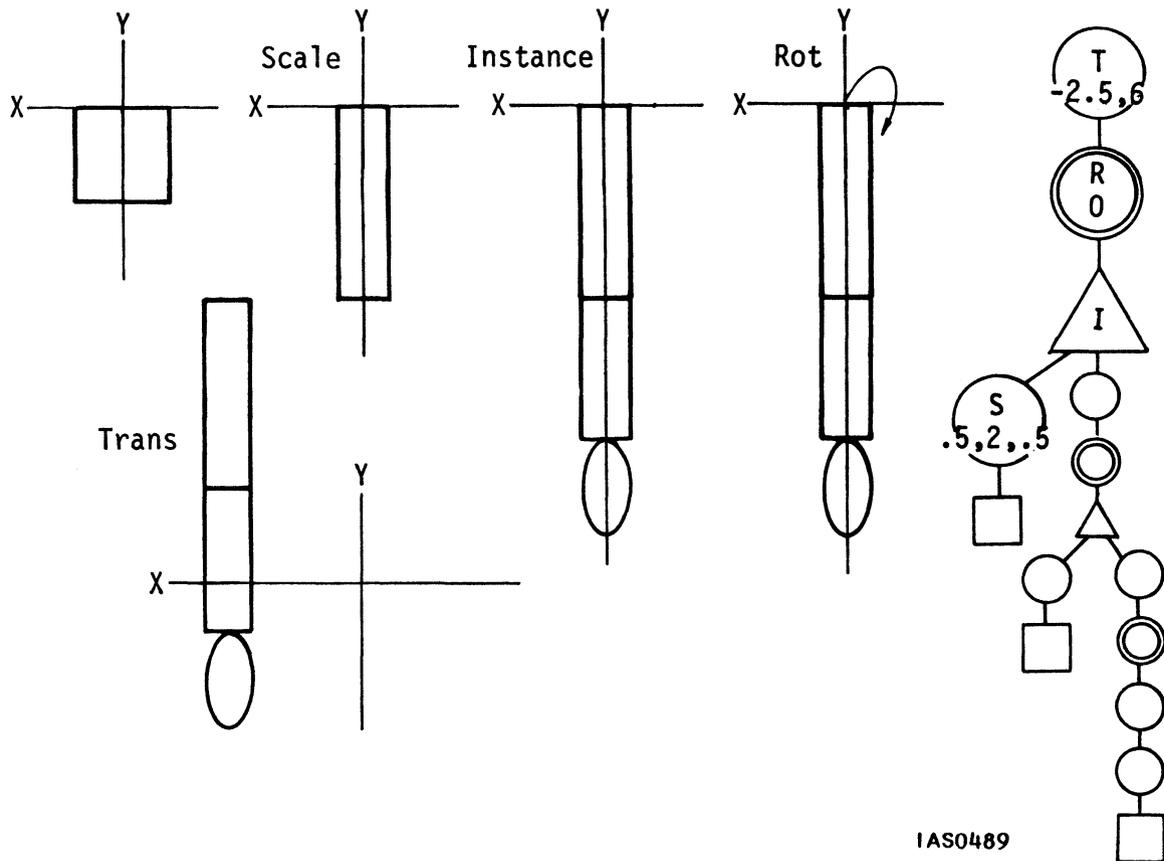


Figure 30. Robot--Right Arm Display Tree

Creating the Left Arm

Many of the modeling steps used to create the right arm are used to create the left arm. Rather than repeat these nodes in a second branch of the display tree, you can "share" nodes whenever possible, reducing the total number of nodes in the display tree.

For example, since a hand has already been modeled by scaling and translating a sphere, these nodes can be referenced in the other arm. However, the second hand requires a separate rotate node so the left hand can "wave" independently (Figure 31).

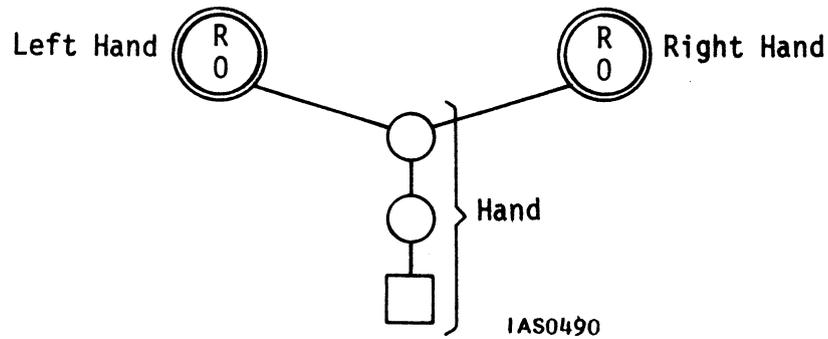


Figure 31. Robot--Shared Nodes for Hand

The left hand also requires its own translate node to move the hand from the origin down the Y axis the length of the forearm piece (0,-3) (because one transformation cannot point directly to two descendent nodes).

The forearm piece was already created in doing the right hand, so next create left forearm as an instance of the forearm piece and the hand (Figure 32).

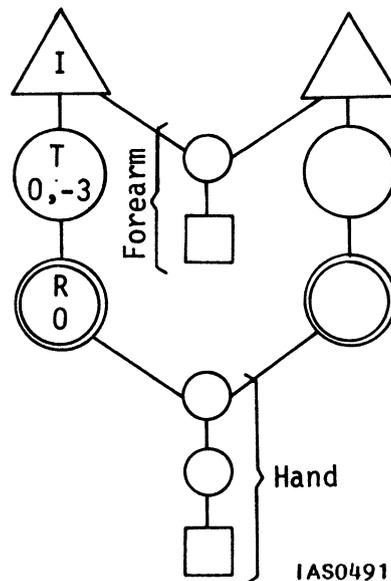


Figure 32. Robot--Left Forearm Display Tree

Then insert a rotate node so the lower arm moves at the elbow and translate this part of the arm down the Y axis the length of the upper arm piece (0,-4).

The upper arm piece is already built, so it can be joined to the rest of left lower arm with an instance node. A rotate node comes next to allow for left shoulder manipulation. Then the left arm can be translated out to its final place in Robot (2.5,6).

Figure 33 shows the display tree for the right and left arms.

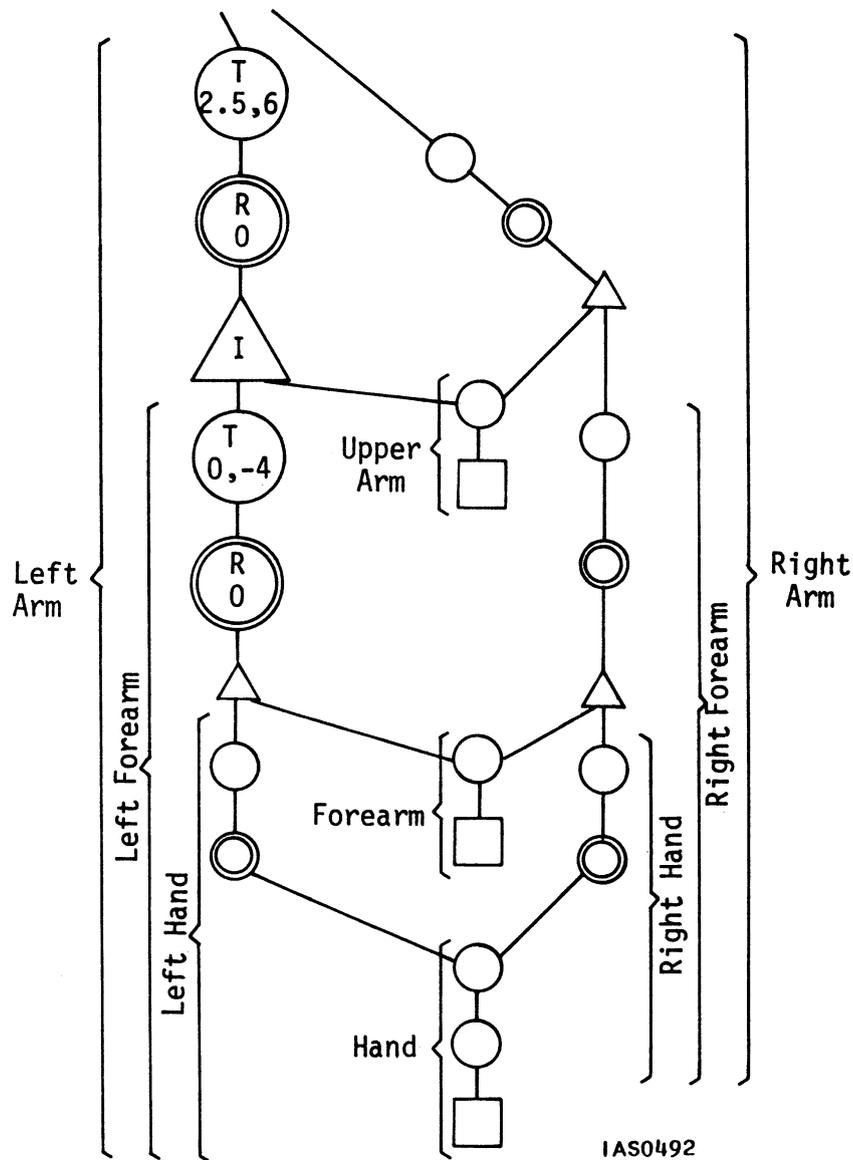


Figure 33. Robot--Display Tree for Two Arms

Besides the two arms, the upper body includes the head and trunk.

Creating the Head

Scale (1,2,1) the sphere primitive to create Robot's head. Then translate the head (0,1) so rotations (0,0) such as nodding and shaking the head, take place at the neck. (Notice that, in this case, translating before scaling would produce the same result.)

The head can then be translated to its final position (0,6). See Figure 34.

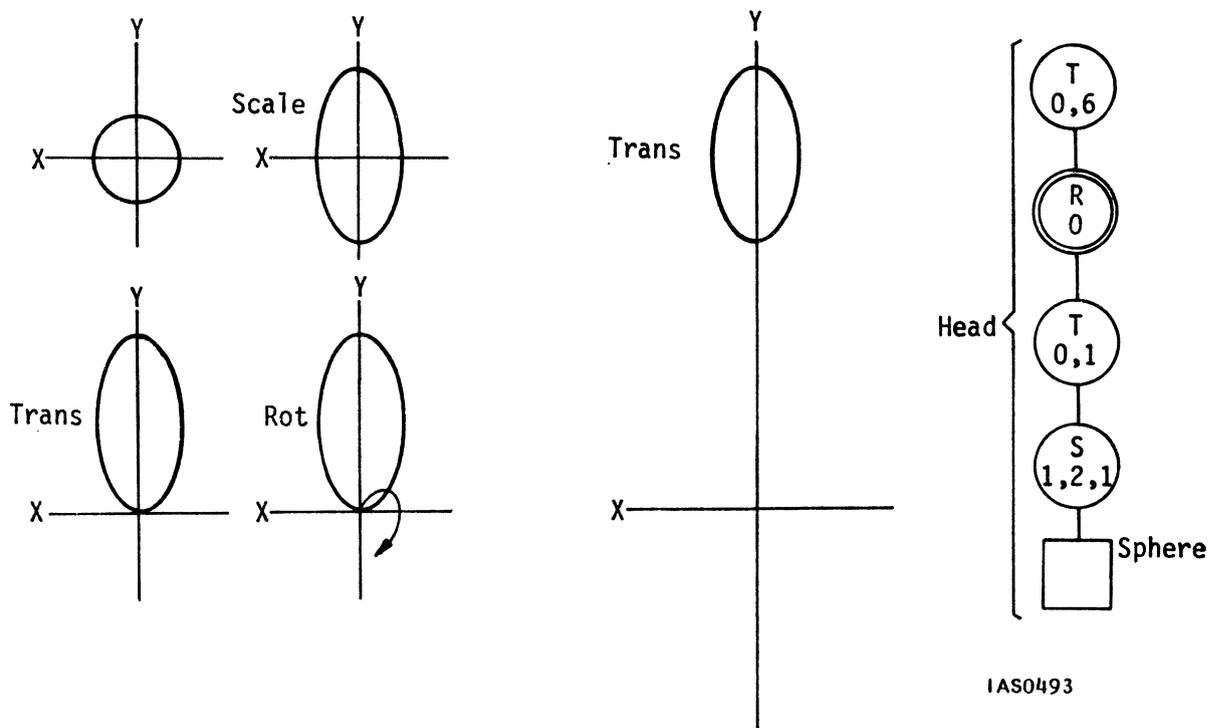


Figure 34. Robot--Head Display Tree

Creating the Trunk

With the arms and head built, you need only complete the trunk before linking all four together as Upperbody. Scale the cylinder for the trunk (2,3,1). Then translate the cylinder up the Y axis (0,6) to its final position. (This is the one time in the Robot model when it would have been better to have a cylinder primitive that rested "on top of" the X axis instead of "hanging down" from it.)

Then join the arms, head, and trunk to form the upper body with an instance node. Finally, insert a rotate node above the upper body to allow Robot to bow at the waist and turn from side to side.

Figure 35 shows the display tree for transformations that define the trunk and also shows the display tree for the upper body of the robot.

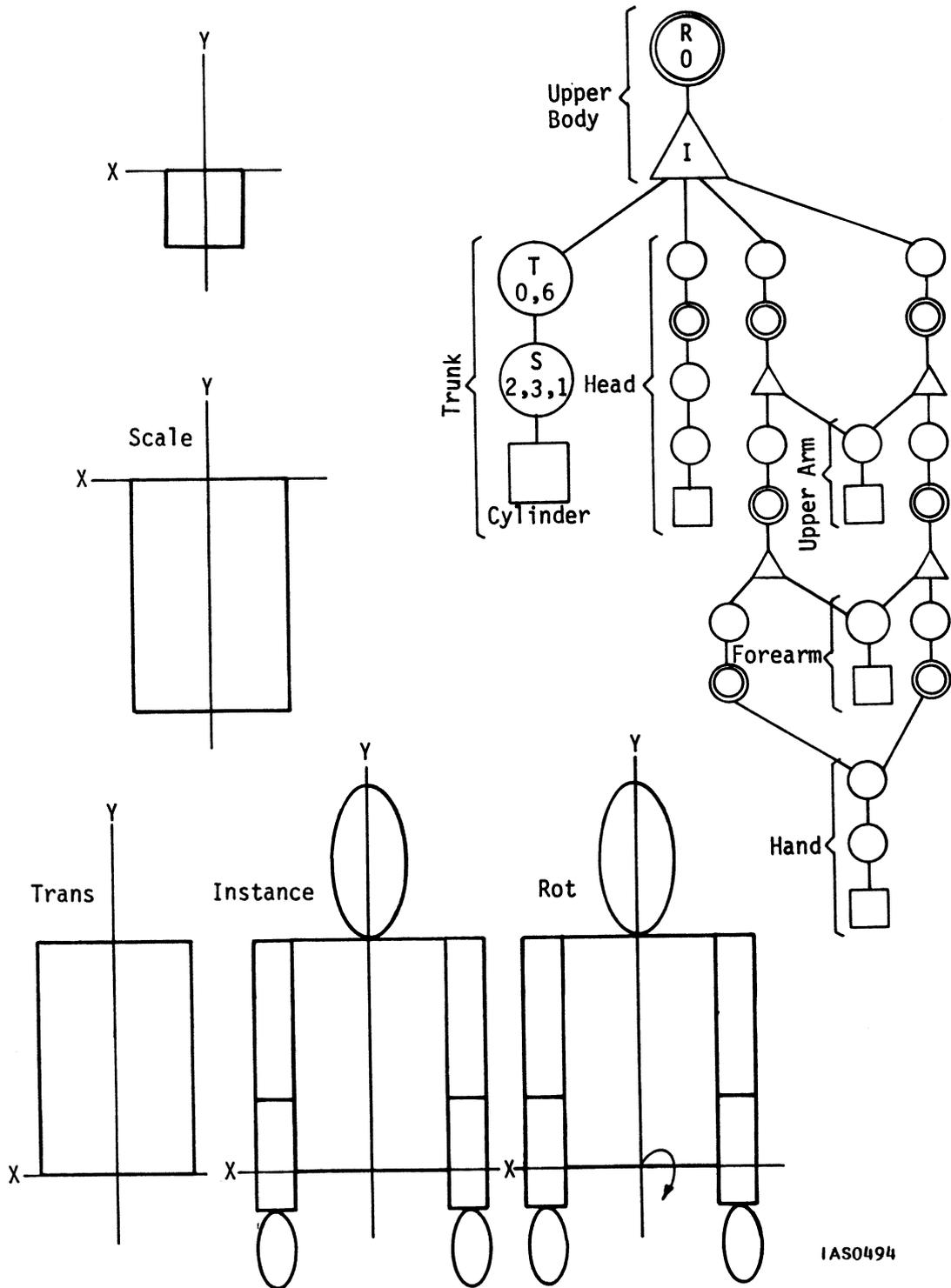


Figure 35. Robot--Upper Body Display Tree

No constraints have been placed on how much Robot can turn. He can rotate his head 360 degrees if desirable. If you are trying to model a human realistically, you must set limits. These are put in place using function networks (refer to Volume 2B, "Function Networks II" module for more information on this.)

The rotate node that allows Robot to bow cannot be put above the trunk alone. It must be above the instance node for Upperbody in the display structure to affect ALL the parts of Upperbody.

Now that the upper body is finished, the lower body needs to be built before they can be linked together as an instance of BODY. Begin at the bottom of the hierarchy with the foot and build up the display tree. In building the legs, proceed as you did with the arms, sharing nodes whenever possible.

Creating the Right Foot

Because the foot is positioned perpendicular to the leg, the primitive cylinder must first be rotated 90 degrees in Z. Then the cylinder can be scaled to its proper size (.75,.5,1). Finally, the scaled foot must be translated back in Z so the foot will be correctly placed on the leg (0,0,1). These transformations are shown in Figure 36.

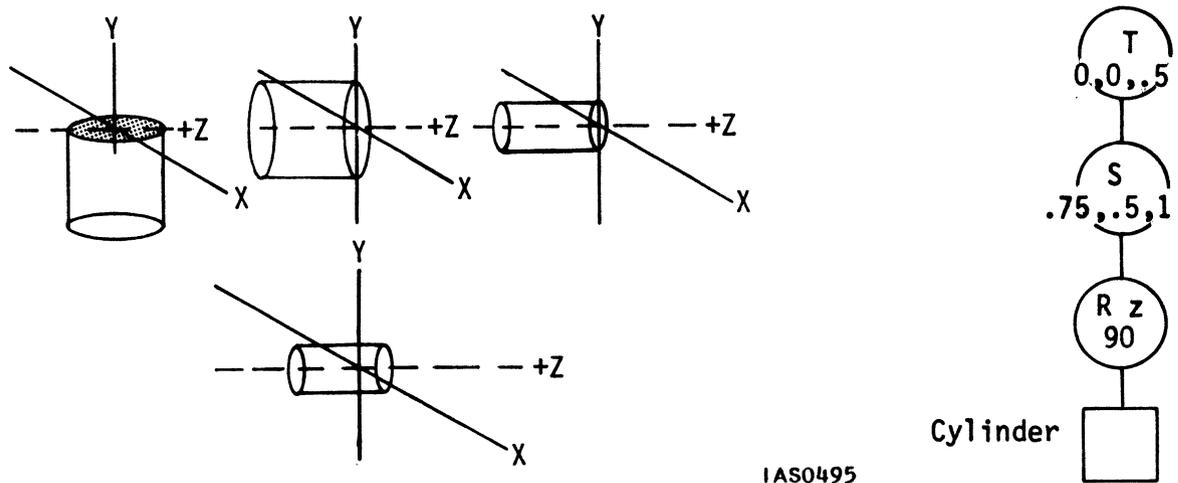


Figure 36. Robot--Foot Display Tree

Above the foot, place a rotate node to allow independent movement at the ankle. Then translate the foot down the Y axis so you can build the calf (0,-5.5) (Figure 37).

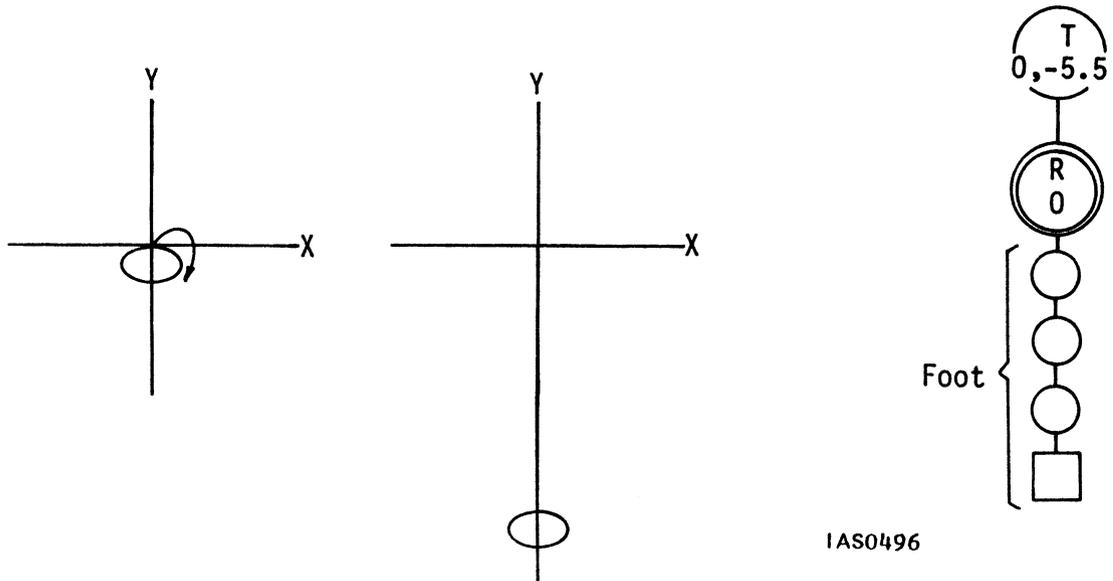


Figure 37. Robot--Rotate and Translate for Foot

Creating the Right Calf

Build the calf by scaling the cylinder (.65,2.5,.65). Link the calf and the foot together as an instance of lowerleg. Place a rotate node above that to allow the knee to bend, and then translate the right lower leg down the Y axis so you can build the thigh (0,-5). Figure 38 shows these transformations.

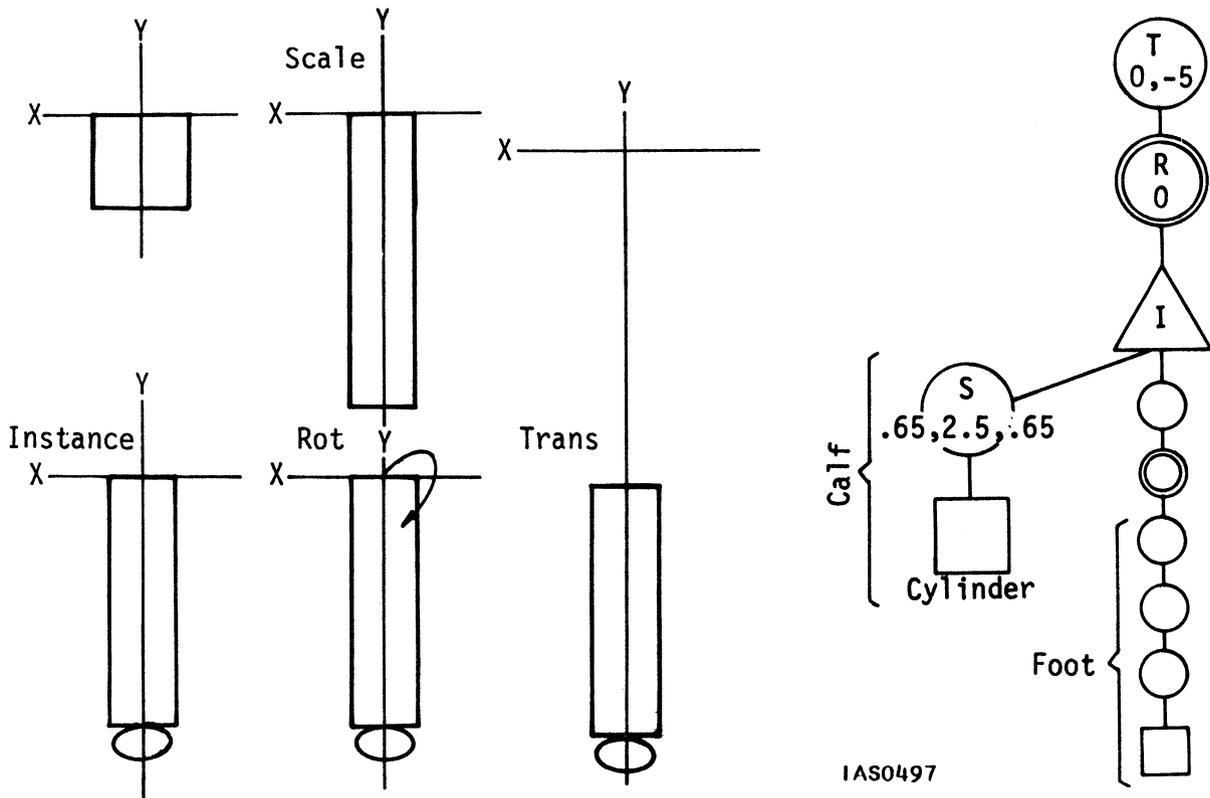


Figure 38. Robot--Right Calf Display Tree

Creating the Right Thigh

To create the thigh, scale the cylinder (.75,2.5,.75). Link the thigh and the lower leg together as an instance of right leg, and put a rotate node above that to allow the leg to kick. Then translate the right leg into its final position (-1,-2).

Figure 39 shows the transformations that create right thigh and the display tree for the right legs.

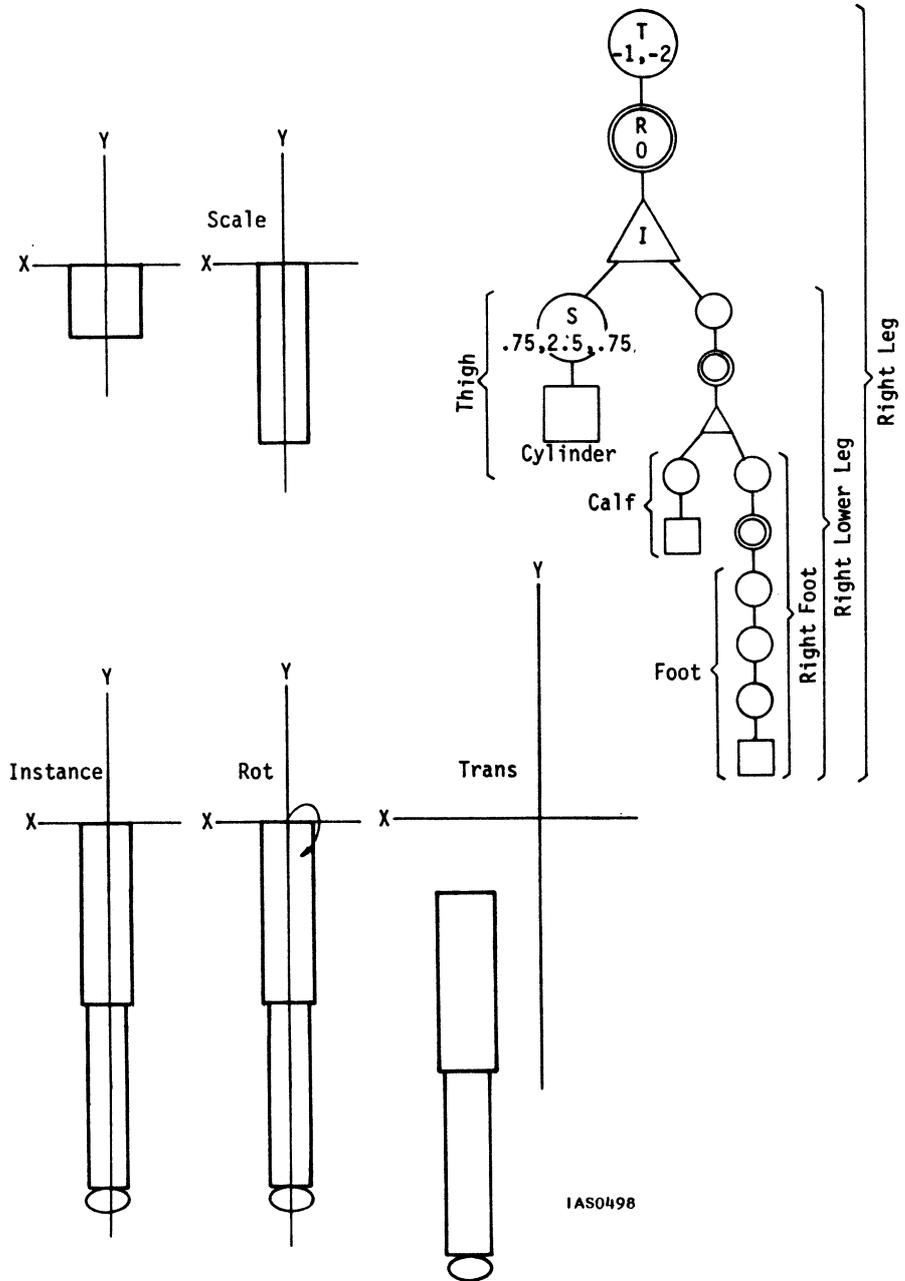


Figure 39. Robot--Right Thigh Display Tree

Creating the Left Leg

Many of the modeling steps used to create the right leg are used to create the left leg. Rather than repeat these nodes in a second branch of the display tree, share nodes whenever possible.

For example, since the scaled cylinders for the foot, calf, and thigh pieces are used in both legs, a single scaled primitive for each can be used by both legs.

Since the nodes to create a foot piece are already in place, the first new node needed to build the left leg will be a rotate node (so the left ankle can move independently) (Figure 40).

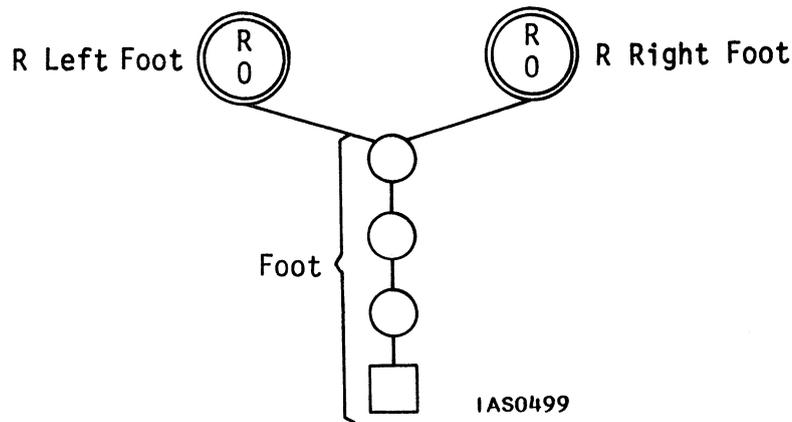


Figure 40. Robot--Shared Nodes for Foot

The left foot also requires its own translate node to move the foot from the origin down the Y axis the length of the calf piece (0,-5.5). (Though these translate values are the same as for the right foot, this node cannot be shared because a translate node can have only one direct descendent node.

The calf piece was already created in doing the right leg, so create left lower leg as an instance of the calf piece and the left foot as shown in Figure 41.

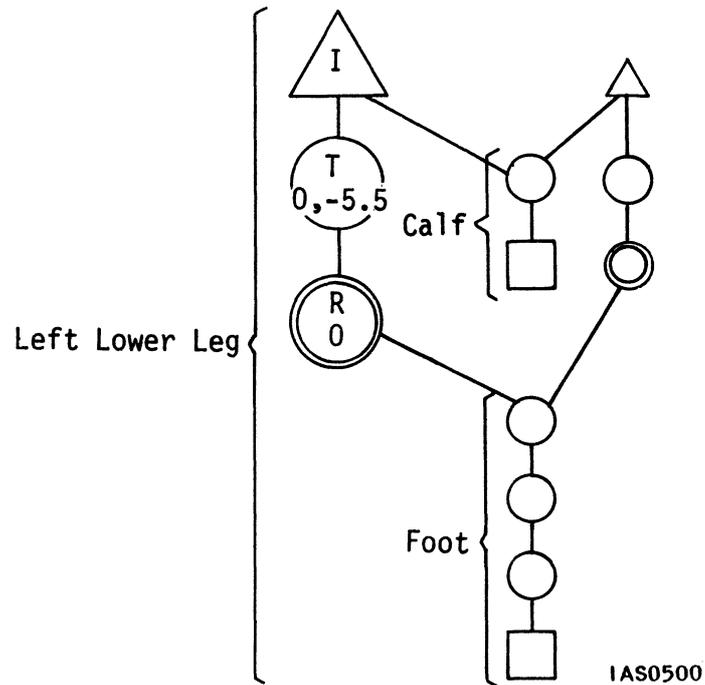


Figure 41. Robot--Left Lower Leg Display Tree

Then insert a rotate node so the lower leg moves at the knee and translate this part of the leg down the Y axis to make room for the thigh piece (0,-5).

The thigh piece is already built, so it can be joined to the left lower leg with an instance node. Then a rotate node can be added to allow for manipulation. Finally, the left leg can be translated out to its final place in Robot (1,-2).

Figure 42 shows the display tree for the left and right legs.

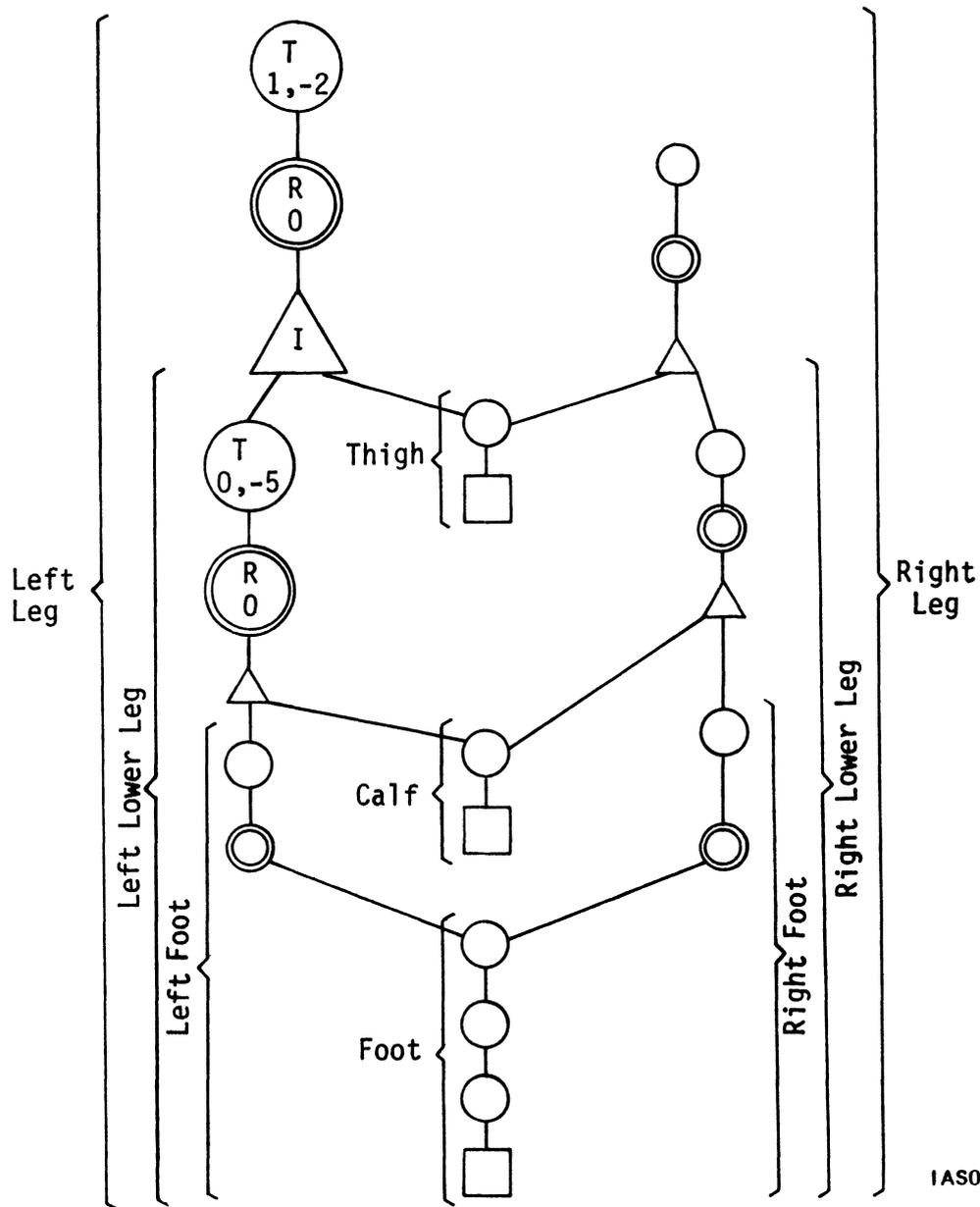


Figure 42. Robot--Left Leg Display Tree

Creating Lowerbody

Now build the last piece of Lowerbody, the pelvis, and then link that with the two legs.

Scale the cylinder (2,1,1). There is no need to translate this into position, since it is already in place below the X axis and the legs have been translated down to fit under the pelvis.

Link the pelvis and legs together as an instance of Lowerbody:

The transformations that create the pelvis, as well as the complete display tree⁷ for lower body, are shown in Figure 43.

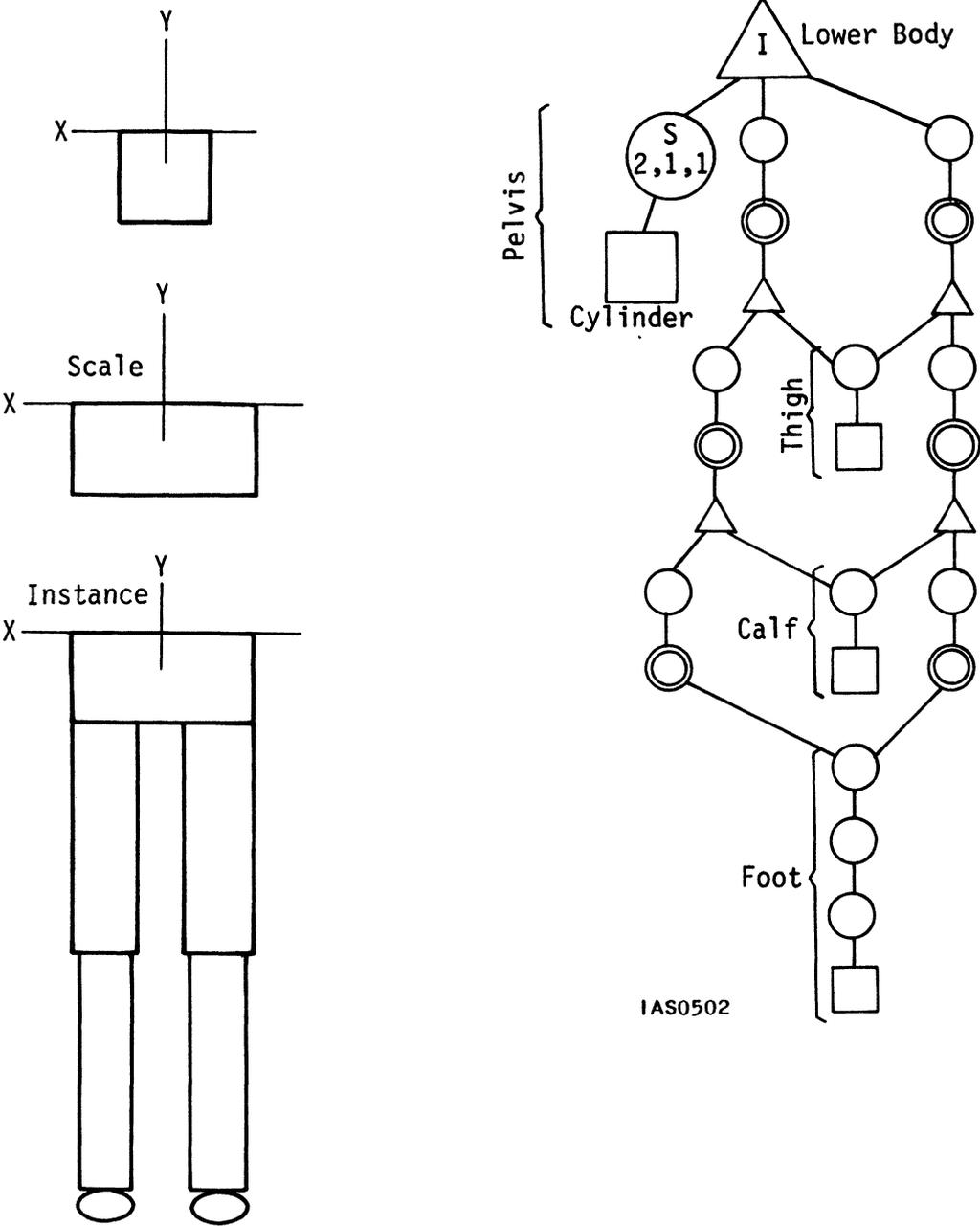


Figure 43. Robot--Lower Body Display Tree

Both major subparts of Robot (Upperbody and Lowerbody) are complete. To move Robot as a whole, link the two subparts together with an instance node called BODY. Then scale Robot (.075) to proportions visible on the screen (the screen coordinates are positive 1 to negative 1 on the X and Y planes). Finally, allow for rotation and translation of the whole robot with the top two nodes.

The completed display tree for Robot is shown in Figure 44.

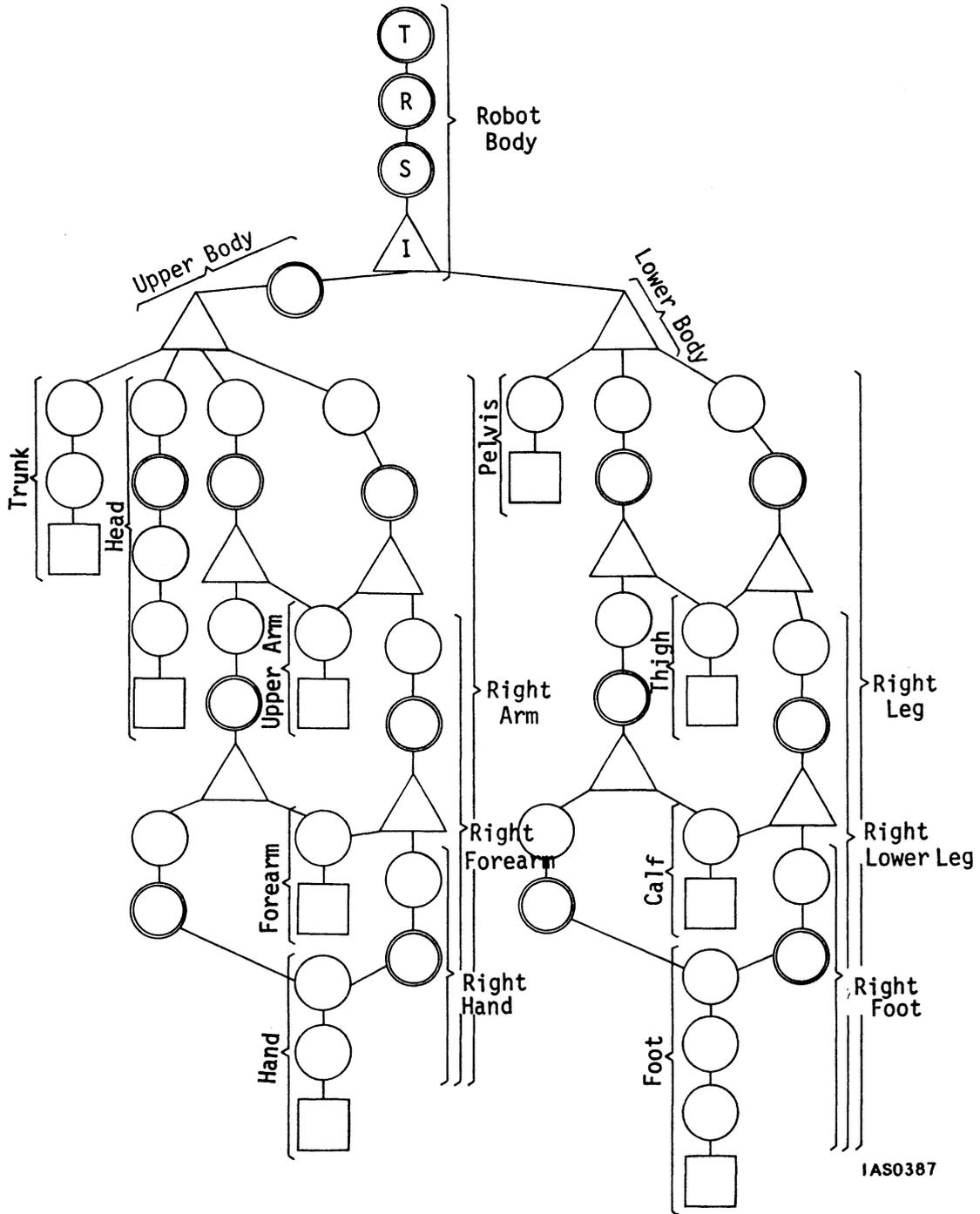


Figure 44. Robot--Completed Display Tree

SUMMARY

This module details the major steps in designing a conceptual data base:

- Designing a hierarchy
- Designing a display tree

In designing a display tree, you must first be aware of what the model looks like, what attributes you want associated with parts, where it's placed in the world coordinate system, what primitives it is made of, and how you want to interact with it.

You can then divide your model into pieces and group those pieces into a hierarchy which shows how they relate to each other.

Finally, you should design the display tree from the bottom up, using the hierarchy as your basis of organization. The design process is as much an art as it is a science, requiring attention to detail, synthesis of information, and a good knowledge of the rules governing display trees.

There are certain rules governing display trees which you should be aware of. For your convenience, these rules have been summarized in Table 1. A more lengthy summary on data, operation and instance nodes follows Table 1.

The next module, "PS 300 Command Language," explains how to code a display tree into the PS 300 using PS 300 commands. It is highly recommended that you read that module next.

Table 1. Rules for Display Trees

NODE TYPE	FUNCTION	POINTERS TO OTHER NODES	COMMENTS
DATA	Vector_Lists, characters, curves, polygons.	0	Data nodes are always terminal nodes in a data structure.
OPERATION	Operation to be performed on data further down the hierarchy. Examples include--Translate, Rotate, Character_Font, Set Level_Of_Detail, etc.	0 or 1	0 pointers makes this node and the path to it useless. All terminal nodes in a data structure should be data nodes.
INSTANCE	Point to other nodes. Save and restore the state of the machine between descendent branches. The state of the machine includes: A. The current transformation matrix (CTM) B. Current Level_Of_Detail C. Current Conditional_Bits values D. Pick id's active	0,1,2,...	Except for some debugging uses, 0 or 1 pointers from an INSTANCE node is an inefficient data structure.

The following sections detail important rules to follow when designing display trees.

Data Nodes

Data nodes represent the primitive shapes that compose a model. Data nodes are always the terminal (bottom) nodes in any display structure; they never "point to" other nodes.

The data base that defines primitives may originate from several sources. You may have been given the geometry already from another source. For example, if you are an architect, you may already have access to primitive shapes to define windows, doors, roofs, and buildings.

You can specify all the vectors in a primitive, or you might use commands to specify characters, curves, and polygonal primitives.

- Vector lists define an object in terms of its coordinate points and how to connect them. Points and line endpoints are expressed as world coordinate locations. The VECTOR_LIST command allows you to specify all the vectors that make up an object.
- You can use line patterns (dashes, center lines, etc.) to draw a vector list with the WITH_PATTERN command.
- The PS 300 allows you to generate vector lists that specify curves using POLYNOMIAL and BSPLINE commands.
- Characters are ASCII character codes that are displayed using a predefined character font. Both the CHARACTERS and LABELS commands define an object as a character string. (Refer to the "Text Modeling" module for details on characters.)
- Polygons can be created to define surfaces for advanced 3D visualization operations (PS 340). (Refer to the module "Using the PS 340--Rendering Operations for Surfaces and Solids" for more information on this.)
- Interactive devices are commonly connected to operation nodes in a display structure, but they can also be connected to other nodes. For example, you can use a dial or data tablet to generate points for a vector list.

Though a data node never points to another node, it can have multiple parents (more than one operation or instance node pointing to it). For example, to display a windmill with four identical blades, the display structure might reuse a single data node translated to distinct locations (Figure 45).

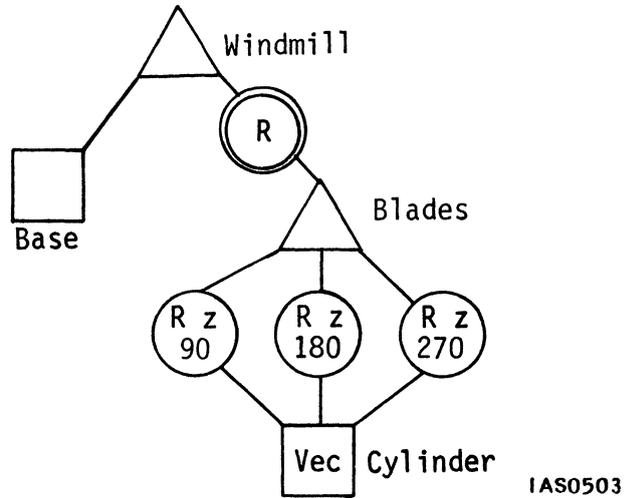


Figure 45. Windmill Display Tree #1

Another way to create a display tree for the same windmill is to define all four blades in a single vector list, as in Figure 46. Creating this display tree might be more programming work (specifying the points and lines which form each blade) but eliminates four nodes: three operate nodes and one instance node.

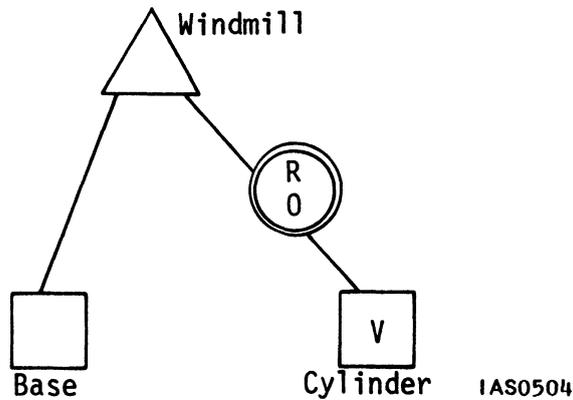


Figure 46. Windmill Display Tree #2

This model is simple. Four nodes is not a significant savings unless the blades are to be instanced many times. But in a complex model, you may save nodes by carefully analyzing the pieces of the model that are better defined with a single vector list.

The tradeoffs: A model made of numerous transformed primitives may be easier and quicker to code than a single vector list of plotted lines and points. However, it often takes a longer time for the Display Processor to traverse the extra transformation nodes than it does to traverse the single vector list. A longer vector list uses more available memory, and may take longer to program, but it reduces the time required for picture refresh.

To save you the trouble of plotting all the points and lines initially, the PS 300 has a group of commands and functions that convert a data base consisting of transformed data into a single vector list for you. Refer to the module on "Transformed Data" for more information on this.

Operation Nodes

An operation node represents an operation to be performed on data below it in the display tree. Operation nodes are used to represent all types of operations, including translations, rotations, and attributes such as set level of detail and color.

As part of a display tree, an operation node affects only what is below it on a hierarchical branch. Although an operation node can have multiple nodes pointing to it, it can point to no more than one node below it (Figure 47).

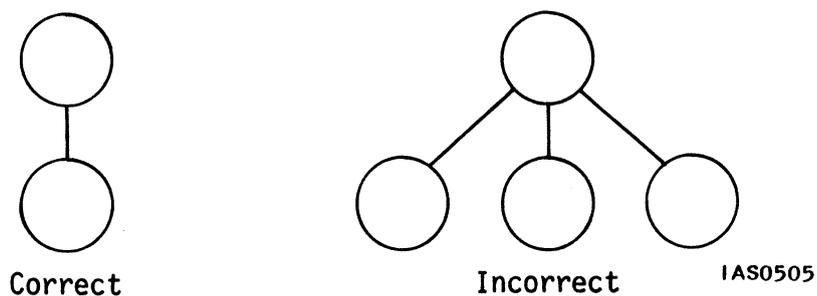


Figure 47. Correct and Incorrect Usage of Operate Nodes

Operation nodes can point to a data node, an instance node, or another operation node. However, if a display tree contains a series of operation nodes, the order of the operations is significant. Put the node for the first operation to be performed on the data closest to the data node. Place the second operation node above that, and so on.

Operation nodes are used in two ways: for modeling and for interaction. Modeling operations are done strictly to shape the "building blocks" of a model and move them into place. Interaction operations allow you to interact with a model. Any operation node can be one or the other, depending on how it is used.

On the display tree diagrams, an interaction node is differentiated from other operation nodes by a double circle.

Interaction nodes allow you to interactively translate an object in X, Y, or Z; rotate that object around any one axis; or scale the object. In addition, you can alter the typeface of displayed text by changing the character font, apply viewing transformations (to create a limitless number of different views of an object), or specify viewports (different areas on the screen where the object is displayed).

Interaction nodes receive their values either from interactive devices or from a host computer via a function network. (For more information on function networks, refer to the "Function Networks I" module.)

Instance Nodes

Instance nodes serve two purposes:

First, they group other nodes and branches in a display tree. Instance nodes are the only nodes that can point to more than one descendant node in the display tree. Consequently, they are found wherever a hierarchical display tree breaks into more than one descending branch.

Second, instance nodes save and restore the *state of the machine* between descendant branches. The state of the machine includes the current transformation matrix, the current level of detail, current color, and status of pick identifiers. (For more information on this, refer to the "Graphics Principles" and to specific tutorial modules in Volume 2B.)

Instance nodes save you from having to save and store data explicitly before a path is traversed, thus preserving the integrity of the state of the machine down any path. As a quick review, look at the way in which the display processor saves and restores the state of the machine in the display tree shown in Figure 62. It is an illustration of the principle of *sphere of influence*.

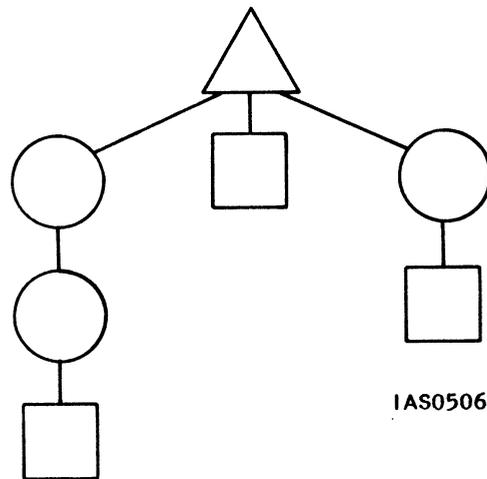


Figure 48. Sphere of Influence

When the Display Processor traverses this display tree, it saves the machine state when it reaches the instance node. It then continues down the leftmost branch until it reaches the data node. The data there are sent through the current transformation matrix and out to be displayed.

The Display Processor still has to travel down the other branches under the instance node. But the state of the machine is altered by the two operations in the first branch, so it must be restored to what it was when the Display Processor first transversed the instance node. Since the machine state is saved when it first reaches the instance node, it can be restored to that state.

The Display Processor returns to the instance node, restores the state that is saved there, and continues down the next branch to the right (the data node). The Display Processor sends that data through the matrix and out for display, returns to the instance node, restores the state, and travels down the last branch in the structure.

Saving the state of the machine ensures that operations in one branch can accumulate and affect everything below them in the display structure, and still not affect anything in other branches.

If an instance node points directly to data nodes, the state is not saved and restored because data nodes do not "operate" on or alter the state of the Display Processor (Figure 49).

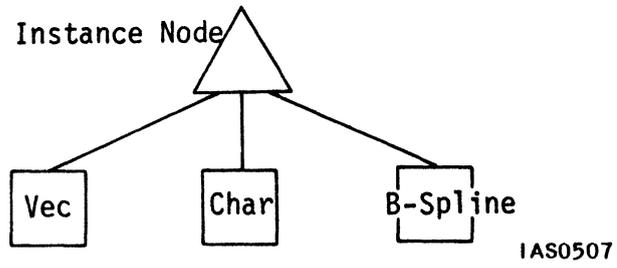


Figure 49. Instance Node Pointing to Three Data Nodes

PS 300 COMMAND LANGUAGE

COMMUNICATING WITH THE PS 300

CONTENTS

INTRODUCTION	1
OBJECTIVES	3
PREREQUISITES	3
USING EXPLICIT NAMING	5
Exercise	8
USING BEGIN_STRUCTURE...END_STRUCTURE	12
Exercise	19
USING IMMEDIATE ACTION COMMANDS	28
ENTERING CODE IN THE PS 300	31
SUMMARY	33



Once you have designed a model's display tree, you can code the display tree into the PS 300 using PS 300 commands. PS 300 commands:

- Create display trees.
- Modify display trees by sending new information to nodes.
- Create and modify function networks (refer to "Function Networks 1" in this volume).
- Instruct the Display Processor to display items or remove them from the display list.
- Query or reset the Command Interpreter (such as Command Status or !Reset).

PS 300 commands are NOT stored in memory. They are interpreted, and either execute immediately (e.g., DISPLAY OBJECT;) or create a data entity in Mass Memory.

Two kinds of PS 300 commands are detailed in this module: data structuring commands and immediate action commands.

Data structuring commands are the only commands that can be named, either directly or indirectly. Data structuring commands create the data structures in Mass Memory which correspond to a model's display tree or to a function network. (Data structuring commands that create function networks are dealt with in the "Function Networks 1" module.) The PS 300 associates the user-assigned name of the command with the Mass Memory location of the data structure the command creates.

Naming can be done explicitly by giving a unique name to a single node in a display tree, such as naming a rotation node. It can also be done via BEGIN_STRUCTURE...END_STRUCTURE, where a single name is assigned to a group of nodes.

Data structuring commands can be created in a file or an application program on the host system and then downloaded to the PS 300, or an application program can send these commands either directly in ASCII, or via the Graphics Support Routines (GSRs) in PS 300 binary format (preparsed).

The other type of command, the immediate action command, performs immediate operations, such as displaying an object on the screen. Because it does not create any autonomous structures in Mass Memory, an immediate action command cannot be named.

PS 300 commands are designed to be easy to use. Once you are familiar with how commands are used, you can refer to the *Command Summary* in Volume 3A, for quick explanations of all existing commands.

You will also want to become familiar with the Graphics Support Routines (GSRs). These are a collection of routines or procedures which allow faster, more efficient communication between the PS 300 and the host computer. Using a Graphics Support Routine causes a corresponding GSR command to be sent directly to the PS 300 without requiring further parsing. The GSRs are contained in Volume 3B.

All commands follow these conventions:

- Commands end with a semicolon.
- The name of each command is indicative of what the command does (for example, INITIALIZE, DISPLAY, and ROTATE).
- Commands have both a long and a short form. The short form is the shortest form of the word that uniquely identifies the command. For example, "DELETE" can be referenced by "DEL"; "APPLIED TO" can be referenced by "APPL". In this module, a PS 300 command will be written out fully in capital letters. For the short form of any command, consult the *Command Summary*.
- Commands may be entered in uppercase or lowercase letters or any combination of these. The PS 300 does not distinguish between upper and lower case.
- The PS 300 command language is free-formatted. Comments go wherever you can put a space and are enclosed in curly braces.

{This is a comment}

Comments, carriage returns, line feeds, spaces, and tabs are all treated as delimiters (white space). If a command extends beyond a single line, the PS 300 reads each line as part of the command until it reads a semi-colon. For example, the PS 300 interprets all of the following commands in the same way:

```
ROTATE IN X 0 {comment} THEN Object;
```

```
ROTATE IN X 0          THEN Object;
```

```
ROTATE IN X 0   <tab>THEN Object;
```

```
ROTATE IN X 0 <CR>  
THEN Object;
```

OBJECTIVES

In this module, you will use PS 300 commands to create a model in PS 300 Mass Memory and to display that model on the screen. To do this you will learn to:

- Use explicit naming.
- Use BEGIN_STRUCTURE...END_STRUCTURE commands.
- Use immediate action commands.
- Enter code into the PS 300.

PREREQUISITES

Before reading this module, you should be able to design a display tree (refer to the "Modeling" module).

You should also be able to perform the operations detailed in the *User Operation and Communication Guide* in Volume 1. This includes how to switch to TE mode, how to use the text editor on your host, and how to download a file from the host.



USING EXPLICIT NAMING

Once you have a model's display tree designed on paper, you can code the structure into Mass Memory by entering data structuring commands into a file on the host.

Data structuring commands create each node in a model's display tree. Each command **MUST BE NAMED** either directly or indirectly. The name provides the "address" in Mass Memory for locating the corresponding data structure.

Which commands to use are determined by the model's display tree, like using a flowchart to determine code in conventional programming.

One way to code is to have one named command correspond to each node in the display tree. This is called "explicit naming". Another way is to group numerous nodes together in one named command. This is done using the `BEGIN_STRUCTURE...END_STRUCTURE` command. Most often, a combination of the two methods is used.

With either method, use the following naming conventions:

- Names can be up to 240 characters long. The name can be any alphanumeric combination but it must begin with a letter. The PS 300 does not distinguish between uppercase and lowercase letters, so use these for clarification, such as in the example: RightLowerArm.
- Names can include the underscore character (`_`), a dollar sign (`$`), or a period (`.`). However, it is strongly recommended that you do not use periods when explicitly naming something. (The PS 300 automatically inserts periods in the name of nodes contained in a `BEGIN_STRUCTURE...END_STRUCTURE`. For example, a node named "ROT" within a `BEGIN_STRUCTURE...END_STRUCTURE` you named HAND will automatically be named "HAND.ROT" by the PS 300.)
- Names cannot contain a space or other "white space" (`<CR>`, tab, etc.). These signal the end of a name. It may be convenient to run words together in a name (RightArm) or to use underscores (Right_Arm).
- Names must be unique.

- The following PS 300 commands can be used in conjunction with the naming of data structures:

- To name a data structure, use the (Naming of Display Data Structures) Command:

Name := Display_data_structure_command;

- A null data structure can be named using:

Name := Nil;

The command can also be used to redefine a name. The command saves the name but redefines the associated structure.

- The FORGET command deletes the name assigned to a command but saves the associated structure.
- The DELETE command removes both the name and the associated structure.

Primitives (data nodes) are created by specifying a series of points and the lines or planes that connect them. Several PS 300 commands create primitive shapes:

VECTOR_LIST
POLYGON
BSPLINE and RATIONAL BSPLINE
POLYNOMIAL and RATIONAL POLYNOMIAL
CHARACTERS
LABELS

Text is also a graphical primitive. The CHARACTERS command and the LABELS command create data nodes for displaying text.

This module will not cover how to create data nodes. Rather than listing all the points and lines required to build any data primitive in this module, a VECTOR_LIST command will be represented by the following abbreviated notation:

Name := VECTOR_LIST ... ;

DO NOT enter this abbreviated notation into the PS 300. Any data nodes you will need for exercises on the PS 300 have already been provided for you on the tutorial tape. For more information on creating data nodes, refer to the *Command Summary* in Volume 3A.

Using explicit naming, code the right forearm for the Robot that was designed in the "Modeling" module. The display tree looked like this:

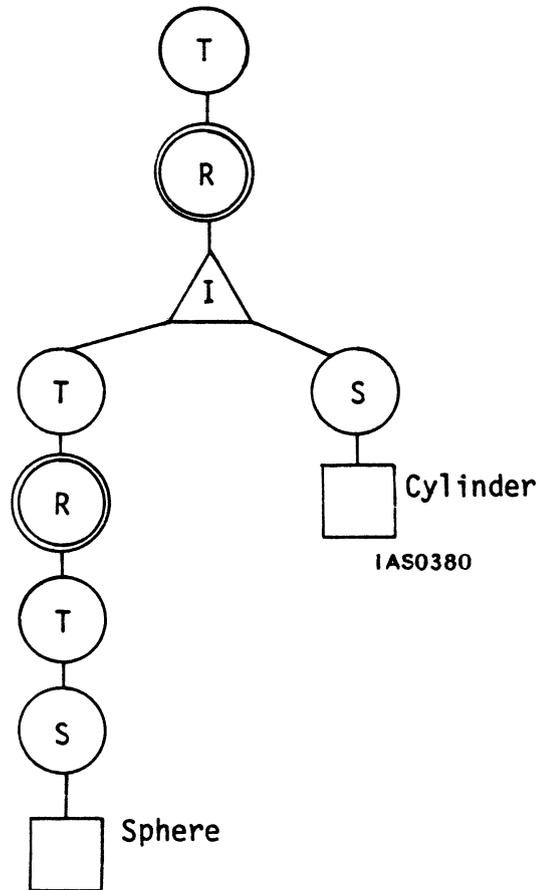


Figure 1. Display Tree for Robot's Forearm

Start from the top down.

```

TranRtLowerArm := TRANSLATE BY 0,-4 APPLIED TO RotRtLowerArm;
RotRtLowerArm := ROTATE 0 APPLIED TO RtLowerArm;
RtLowerArm := INSTANCE OF ForearmPiece, TranHand;
ForearmPiece := SCALE BY .5,1.5,.5 APPLIED TO Cylinder;
Cylinder := VECTOR_LIST ... ;
TranHand := TRANSLATE BY 0,-3 APPLIED TO RotHand;
RotHand := ROTATE 0 APPLIED TO TranSphere;
TranSphere := TRANSLATE BY 0,-1 APPLIED TO ScaleSphere;
ScaleSphere := SCALE BY .5,1,.5 APPLIED TO Sphere;
Sphere := VECTOR_LIST ... ;
  
```

Zeros are given as the values for interactive nodes such as RotRtLowerArm and RotHand because new values will be provided interactively from function networks.

With explicit naming a command can refer, or point, to a name that doesn't exist yet. The name exists once the PS 300 receives a command defining the data structure associated with that name.

Explicit naming has some disadvantages. First, it can be confusing with a complex display tree because you may have hundreds of command names to create and keep track of. Effective comment lines would be essential.

The major drawback of explicit naming is that it FORCES you to name every node in a display tree. This means unnecessary work. The only nodes requiring names are nodes you want to directly reference. There is no need to name nodes that you can safely predict will not receive new values, or be instanced or displayed directly.

Exercise

In the "Modeling" module, you designed a display tree for a car. That display tree is shown in Figure 2.

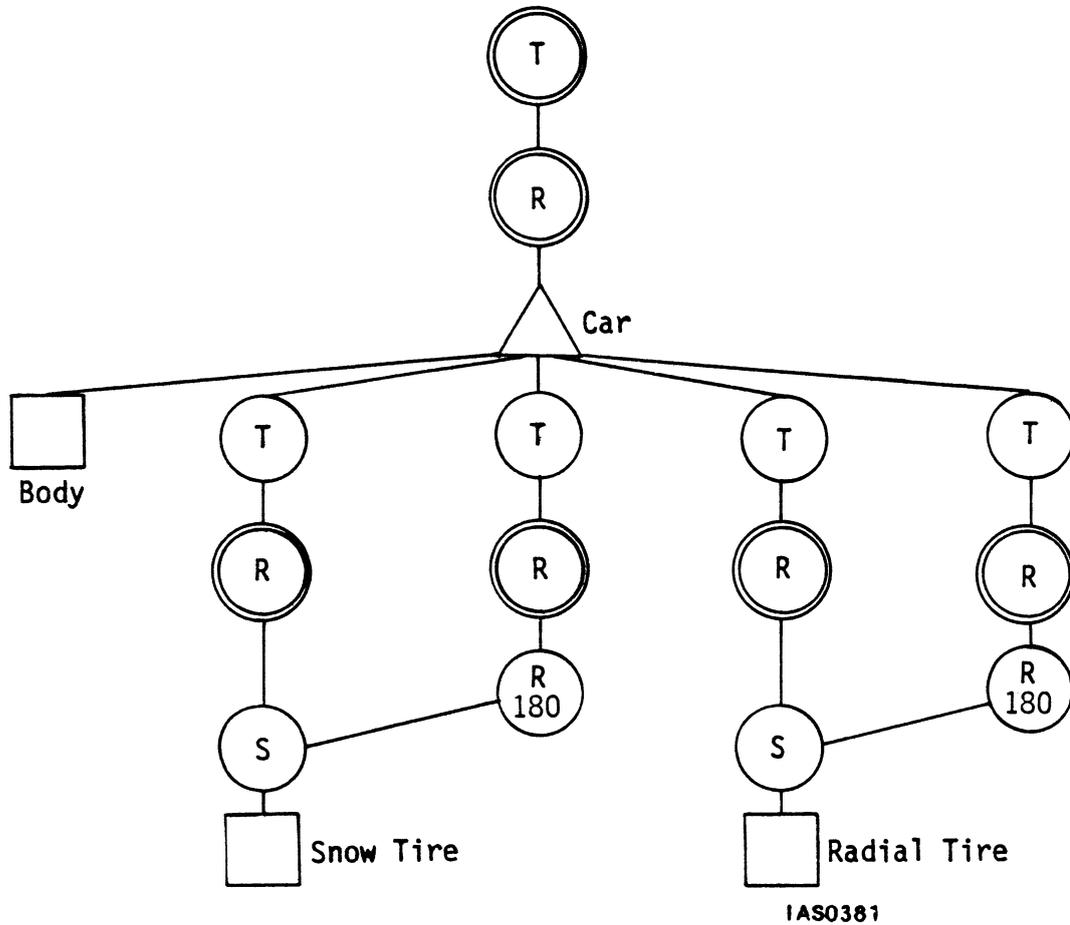


Figure 2. Car Display Tree

Code this display tree from the top down using explicit naming. The values provided below have been calculated to produce the desired transformations.

The first two nodes are interactive nodes which translate and rotate the whole car:

```
TranCar := TRANSLATE BY 0,0 APPLIED TO RotCar;
RotCar := ROTATE 0 APPLIED TO Sportscar;
```

The car is defined as an instance of the car body and the translations applied to each tire:

```
Sportscar := INSTANCE OF Car_Body, Tran_FR_Tire, Tran_FL_Tire,  
Tran_RR_Tire, Tran_RL_Tire;
```

Define the radial on the rear, right side of the car. A translation node and an interactive rotation node are needed:

```
Tran_RR_Tire := TRANSLATE BY -.5664,-.1598,-.3357 THEN Rot_RR_Tire;  
Rot_RR_Tire := ROTATE IN Z 0 THEN Scaled_Radial_Tire;
```

Since both radial tires are scaled, the scaled radial tire can be defined once and then referenced with each tire.

```
Scaled_Radial_Tire := SCALE BY .139 THEN Radial_Tire;
```

The display tree for the radial on the rear, left side of the car also includes a translation and rotation node:

```
Tran_RL_Tire := TRANSLATE BY -.5664,-.1598,.3357 THEN Rot_RL_Tire;  
Rot_RL_Tire := ROTATE IN Z 0 THEN Flip_RL_Tire;
```

Then the tire is rotated so the hubcap faces out:

```
Flip_RL_Tire := ROTATE IN Y 180 THEN Scaled_Radial_Tire;
```

The same process is used for the snow tires. Both snow tires consist of a translation and interactive rotation node. Then the snow tire on the front, left side of the car is rotated so the hubcap faces out.

The code for the snow tire on the front, right side of the car would be:

```
Tran_FR_Tire := TRANSLATE BY .5415,-.1598,-.3357 THEN Rot_FR_Tire;  
Rot_FR_Tire := ROTATE 0 THEN Scaled_Snow_Tire;
```

Since both snow tires are scaled, the scaled snow tire can be defined once and then referenced twice.

```
Scaled_Snow_Tire := SCALE BY .139 THEN Snow_Tire;
```

The code for the front left snow tire would be:

```
Tran_FL_Tire := TRANSLATE BY -.5415,-.1598,.3357 APPLIED TO  
Rot_FL_Tire;
```

```
Rot_FL_Tire := ROTATE 0 APPLIED TO Flip_FL_Tire;
```

```
Flip_FL_Tire := ROTATE IN Y 180 APPLIED TO Scaled_Snow_Tire;
```

The points and connecting lines for the three primitives also need to be defined, In this module, that code is represented by:

```
Snow_Tire := VECTOR_LIST ... ;  
Radial_Tire := VECTOR_LIST ... ;  
Car_Body := VECTOR_LIST ... ;
```

Since the data primitives for Snow_Tire, Radial_Tire, and Car_Body are already provided for you when you load the Tutorial Demonstration tape in the PS 300, display the car by typing:

```
DISPLAY Sportscar;
```

To prepare for the new definition of Sportscar you will be coding in the next section, enter:

```
INITIALIZE;
```

This removes the present definition of Sportscar you just coded in.

USING BEGIN_STRUCTURE...END_STRUCTURE

An alternative to naming every node is to group nodes inside a BEGIN_STRUCTURE...END_STRUCTURE. Though each node within a BEGIN_STRUCTURE...END_STRUCTURE is created in Mass Memory, you only have to name those nodes which will be accessed again. (The Display Processor "accesses" the node every time the model is displayed.)

Besides less naming, another advantage of BEGIN_STRUCTURE...END_STRUCTUREs is that they are an effective way to organize the commands in your file. In a complex display tree, nodes that directly affect each other can be grouped together in the same BEGIN_STRUCTURE...END_STRUCTURE.

BEGIN_STRUCTURE...END_STRUCTUREs are usually used in conjunction with explicit naming. To illustrate this, code the Sportscar using a combination of the two. There are many ways to code a model. The following is only one possible way of determining which type of code to use.

Before coding, identify all data nodes and any shared nodes in the car's display tree. In this case, there are three data nodes: the car body, the radial tire, and the snow tire. The scaled radial tire and scaled snow tire are shared nodes.

Now look at the branches above any shared nodes, data nodes, and instance nodes, for those which have two or more operation nodes. These nodes could be grouped into BEGIN_STRUCTURE...END_STRUCTUREs.

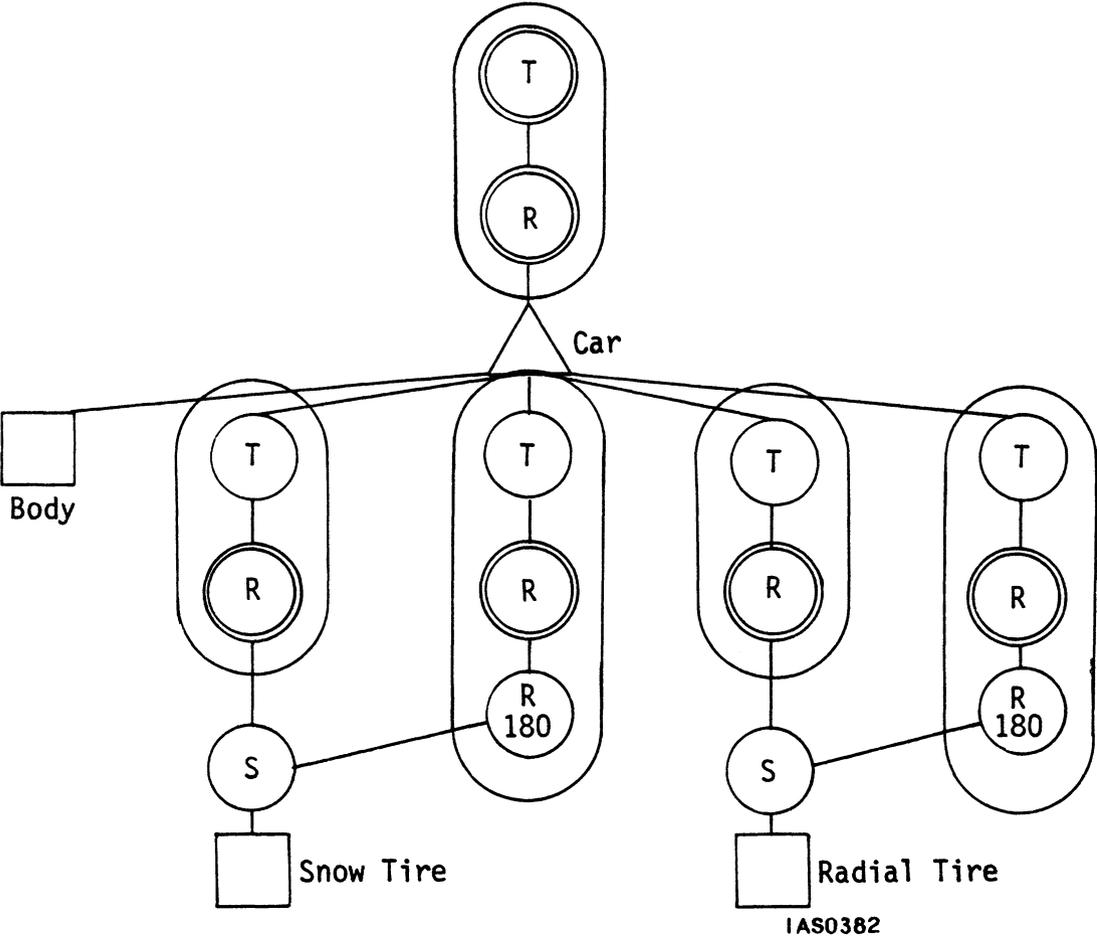


Figure 3. Possible BEGIN_STRUCTURE...END_STRUCTUREs for Car

Begin coding the display tree top down. The first two commands allow you to see the car immediately once it is defined.

```
INITIALIZE DISPLAY;  
DISPLAY Sportscar;
```

The display tree begins with a BEGIN_STRUCTURE...END_STRUCTURE:

```
SportsCar :=BEGIN STRUCTURE  
  Tran := TRANSLATE BY 0,0;  
  Rot := ROTATE 0;  
  INSTANCE OF Car_Body, RL_Tire, RR_Tire, FL_Tire, FR_Tire;  
END_STRUCTURE;
```

The interactive translation and rotation nodes must be explicitly named so they can be accessed to provide values to manipulate the car. Grouping these nodes in a BEGIN_STRUCTURE...END_STRUCTURE saves you from having to name the instance node. Naming the whole structure also allows you to reference a "Car" that can be rotated and translated.

Each tire could be defined within a BEGIN_STRUCTURE...END_STRUCTURE:

{Rear left wheel}

```
RL_Tire := BEGIN_STRUCTURE
           TRANSLATE BY -.5664,-.1598,.3357;
Rot :=    ROTATE 0;
           ROTATE IN y 180 APPLIED TO Scaled_Radial_Tire;
           END_STRUCTURE;
```

{Rear right wheel}

```
RR_Tire := BEGIN_STRUCTURE
           TRANSLATE BY -.5664,-.1598,-.3357;
Rot :=    ROTATE 0 APPLIED TO Scaled_Radial_Tire;
           END_STRUCTURE;
```

Scaled_Radial_Tire references a scale operation node and a data node. These two nodes can be shared by both the right and the left rear tires. Notice how the operations within the BEGIN_STRUCTURE...END_STRUCTURE reflect the order indicated in the display tree; i.e., translates precede rotates, which precede scaled data.

In much the same way, the last two branches define the snow tire:

{Front left wheel}

```
FL_Tire := BEGIN_STRUCTURE
           TRANSLATE BY -.5415,-.1598,.3357;
Rot :=    ROTATE 0;
           ROTATE IN Y 180 APPLIED TO Scaled_Snow_Tire;
           END_STRUCTURE;
```

{Front right wheel}

```
FR_Tire := BEGIN_STRUCTURE
          TRANSLATE BY .5415,-.1598,-.3357;
          Rot := ROTATE 0 APPLIED TO Scaled_Snow_Tire;
          END_STRUCTURE;
```

The shared nodes for each tire are coded as:

```
Scaled_Radial_Tire := SCALE BY .139 APPLIED TO Radial_Tire;
```

```
Scaled_Snow_Tire := SCALE BY .139 APPLIED TO Snow_Tire;
```

The three primitives also need to be defined. The actual vector lists for these have been provided for you on the Tutorial Demonstration tape, so that you may see the Sportscar. The following abbreviated code is just to remind you that primitives must always be defined:

```
Snow_Tire := VECTOR_LIST ... ;
Radial_Tire := VECTOR_LIST ... ;
Car_Body := VECTOR_LIST ... ;
```

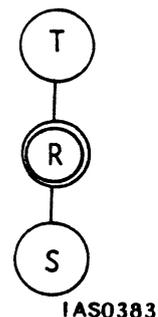
In general, then, BEGIN_STRUCTURE...END_STRUCTUREs:

- Group associated nodes together into identifiable parts of a model.
- Reflect the order of operations shown in the display tree.
- Eliminate the unnecessary naming of nodes. Nodes within a structure are only named if they need to be referenced; i.e., are interactive.

There is one possible disadvantage to using too many BEGIN_STRUCTURE...END_STRUCTURE commands. Each time the PS 300 parses a BEGIN_STRUCTURE...END_STRUCTURE command, it automatically creates another instance node in the display tree.

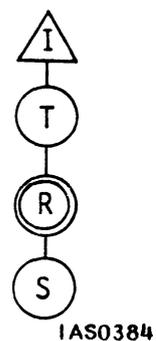
The created instance node is placed above all other nodes within that BEGIN_STRUCTURE...END_STRUCTURE command. The name of the BEGIN_STRUCTURE...END_STRUCTURE is actually the name of the created instance node. For example, if you put the following nodes into a BEGIN_STRUCTURE... END_STRUCTURE:

```
TransMolecule :=TRANSLATE BY 0,1 APPLIED TO RotMolecule;  
RotMolecule := ROTATE 0 APPLIED to ScaleMolecule;  
ScaleMolecule :=SCALE 2,2 APPLIED TO Molecule;
```



The PS 300 would create a display tree like this:

```
Molecule := BEGIN_STRUCTURE  
             TRANSLATE 0,1 APPLIED TO RotMolecule;  
             Rot := ROTATE 0 APPLIED TO ScaleMolecule;  
                   SCALE 2,2 APPLIED TO Molecule;  
             END_STRUCTURE;
```



Extraneous instance nodes are only costly if they are used so frequently that they begin to affect the Display Processor's traversal time. In evaluating when to use BEGIN_STRUCTURE...END_STRUCTURE command, then, you must weigh the advantage of grouping nodes together without having to name each node against the disadvantage of creating an extra instance node in the display tree.

Now that you are familiar with BEGIN_STRUCTURE...END_STRUCTURE commands, examine some rules regarding their use.

1. When using BEGIN_STRUCTURE...END_STRUCTUREs, note that an operation is applied to everything below it in the structure unless the operation is explicitly applied to another structure. If an operation is applied directly to a name, nothing else listed below it in the structure is affected by that operation.

So in the following example, the cylinder is both translated and scaled, but the piston is only translated:

```
Shaft_Housing := BEGIN_STRUCTURE
                  TRANSLATE BY 0,1,0;
                  SCALE 2,2 APPLIED TO Cylinder;
                  INSTANCE OF Piston;
                  END_STRUCTURE;
```

Look at another example. Each operation applies to everything following it in the BEGIN_STRUCTURE...END_STRUCTURE. So the Cube is rotated in X 30 degrees. The sphere is rotated in y 10 degrees and rotated in X 30 degrees. The pyramid is translated by 0,1,1; rotated in Y 10 degrees; and rotated in X 30 degrees.

```
Shapes := BEGIN_STRUCTURE
           XRot:= ROTATE IN X 30;
           Cube:= VECTOR_LIST ...;

           YRot:= ROTATE IN Y 10;
           Sphere:= VECTOR_LIST ...;

           Tran:= TRANSLATE BY 0,1,1;
           Pyramid:= VECTOR_LIST ...;
           END_STRUCTURE;
```

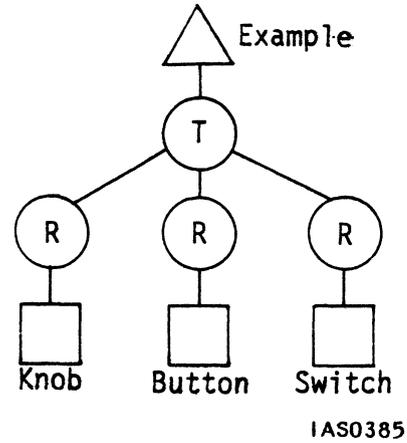
2. Remember from the "Modeling" module that instance nodes can point to any number of descending nodes, operation nodes can point to only one descending node, and data nodes are terminal nodes; that is, they have no descending nodes.

Inside a BEGIN_STRUCTURE...END_STRUCTURE, if an operation node points to more than one other node below it, the PS 300 automatically creates an instance node below the operation node. The operation node points to the instance node, which points to the descendant branches.

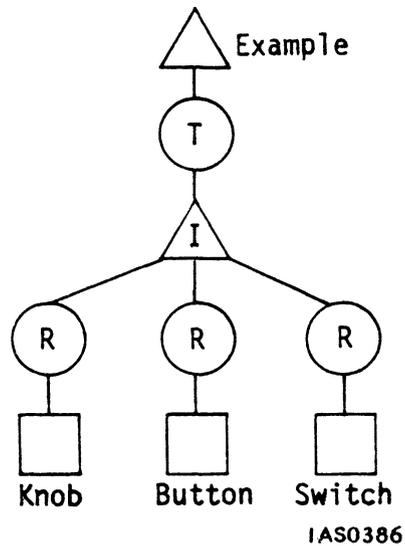
For example, the BEGIN_STRUCTURE...END_STRUCTURE code below is illegal because the translation applies to all three rotations below it.

```

Example := BEGIN_STRUCTURE
          TRANSLATE BY 0,2;
RotKnob := ROTATE IN X
          APPLIED TO Knob;
Rotbutton := ROTATE IN Y
          APPLIED TO Button;
Rotswitch := ROTATE IN Z
          APPLIED TO Switch;
END_STRUCTURE;
    
```



In this case, the PS 300 automatically places an instance node below the translation node:



You could create the instance node explicitly with the following code:

```

Tran := TRANSLATE BY 0,2 APPLIED TO Parts;
Parts := INSTANCE OF RotKnob, Rotbutton, Rotswitch;
    
```

3. When located inside a `BEGIN_STRUCTURE...END_STRUCTURE`, a "user-named" node is assigned a name by the system which consists of the `BEGIN_STRUCTURE...END_STRUCTURE` name, a period, and the "user-assigned" name of the node. So in the above example of the `BEGIN_STRUCTURE...END_STRUCTURE` named Shapes, the X Rotate node can be accessed as `SHAPES.XROT`. The period indicates that the name `XROT` is in the `BEGIN_STRUCTURE...END_STRUCTURE` named "Shapes".

The PS 300 assigns the name automatically, so you can keep your naming procedures simple, reusing descriptive names like `SCALE`, `ROTATE`, and `TRANSLATE` as long as they are not repeated in the same `BEGIN_STRUCTURE...END_STRUCTURE` (so all names remain unique).

If a node is not named, it is just part of the hierarchical structure and cannot be addressed explicitly. This is indirect "naming" of a node.

4. `BEGIN_STRUCTURE...END_STRUCTURE`s can be nested inside each other. No operations within a nested `BEGIN_STRUCTURE...END_STRUCTURE` apply to any nodes in the encompassing `BEGIN_STRUCTURE...END_STRUCTURE`. Think of the nested `BEGIN_STRUCTURE...END_STRUCTURE` as a terminal node in the display tree.
5. No immediate action commands should be placed in `BEGIN_STRUCTURE...END_STRUCTURE`s (with two debugging exceptions: `COMMAND_STATUS` and `!RESET`). Refer to the next section of this module on Immediate action commands.

Exercise

In this section you will use the following display tree, which was designed in the "Modeling" module, to code Robot into Mass Memory.

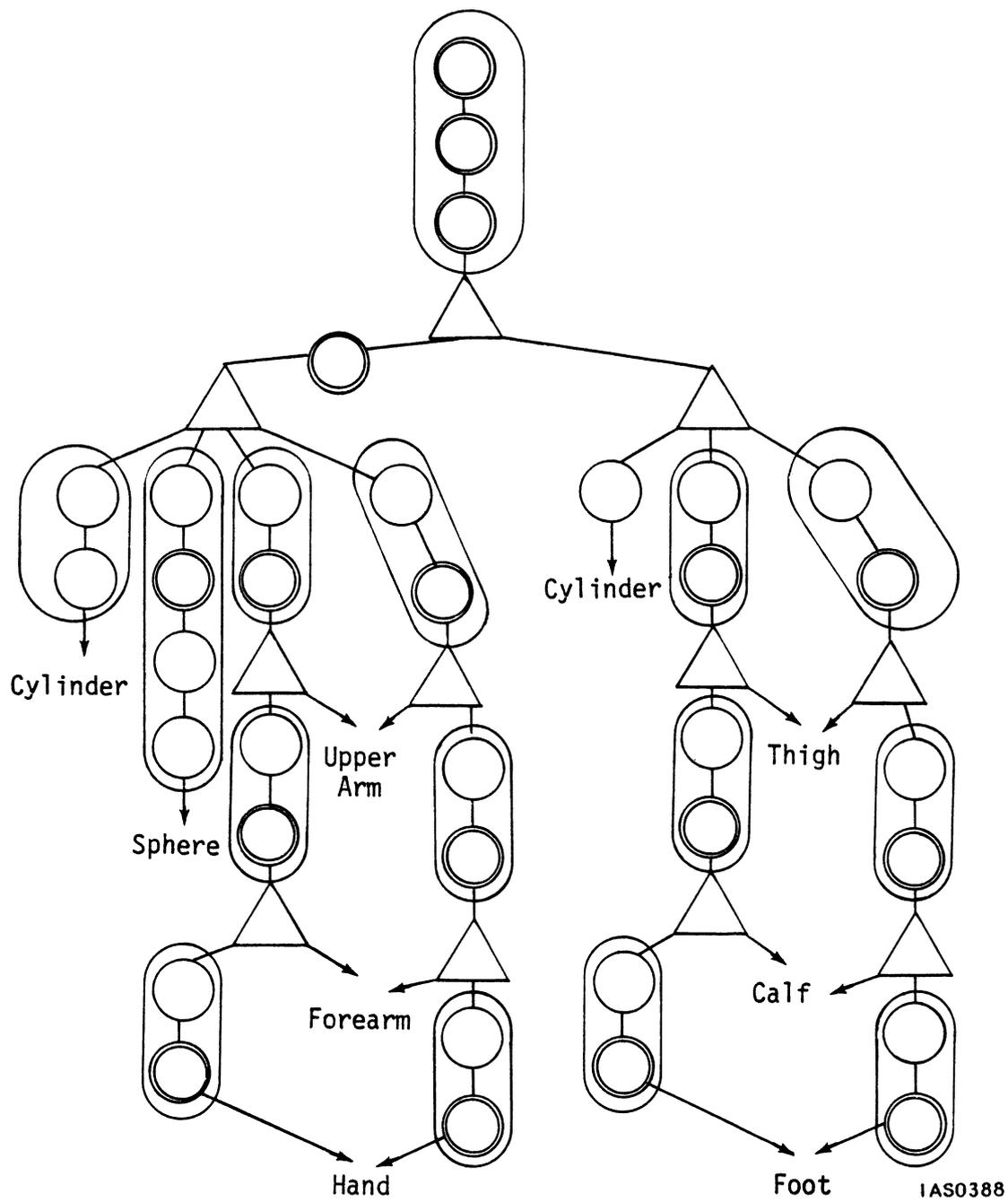


Figure 4. Robot Display Tree

Enter the commands in a host file as you proceed (otherwise, they cannot be saved).

Use explicit naming and BEGIN_STRUCTURE...END_STRUCTUREs to the best advantage. Keep in mind that you could use BEGIN_STRUCTURE...END_STRUCTUREs any number of ways to help you organize and economize on code. This example develops one possible way.

First, isolate and identify all data nodes and any shared nodes.

Remember that the same data node--CYLINDER--has been used for all the body pieces except the hands and head, which are SPHERES. Shared nodes have already been specified in the design of the display tree. These form the pieces labeled hand, forearm, upper arm, foot, calf, and thigh.

First code the primitives. The sphere and cylinder are vector lists that have been provided for you on the Tutorial Demonstration tape. The following serves only a reminder that you must always define primitives.

```
Sphere := VECTOR_LIST ... ;  
Cylinder := VECTOR_LIST ... ;
```

Code the shared body pieces using both explicit naming and using BEGIN_STRUCTURE...END_STRUCTUREs. Calf, thigh, forearm, and upper arm each consist of one node above a data node, so each can be coded explicitly:

```
Upper_Arm := SCALE BY .5,2,.5 APPLIED TO Cylinder;  
ForeArm := SCALE BY .5,1.5,.5 APPLIED TO Cylinder;  
Thigh := SCALE BY .75,2.5,.75 APPLIED TO Cylinder;  
Calf := SCALE BY .65,2.5,.65 APPLIED TO Cylinder;
```

The hand and foot have two or more operate nodes above data nodes and so code them using BEGIN_S...END_STRUCTUREs:

```
Hand := BEGIN_STRUCTURE  
        TRANSLATE BY 0,-1;  
        SCALE BY .5,1,.5 APPLIED TO Sphere;  
        END_STRUCTURE;
```

```
Foot := BEGIN_STRUCTURE  
        TRANSLATE BY 0,0,1;  
        SCALE BY .75, .5, 1;  
        ROTATE IN X 90 APPLIED TO Cylinder;  
        END_STRUCTURE;
```

The figure below illustrates what nodes in the display tree remain to be coded. The nodes already accounted for are represented by name. The branches containing two or more operation nodes above a name, a data node, or an instance node have been circled. These could be coded using BEGIN_STRUCTURE...END_STRUCTUREs. Explicit naming could be used elsewhere.

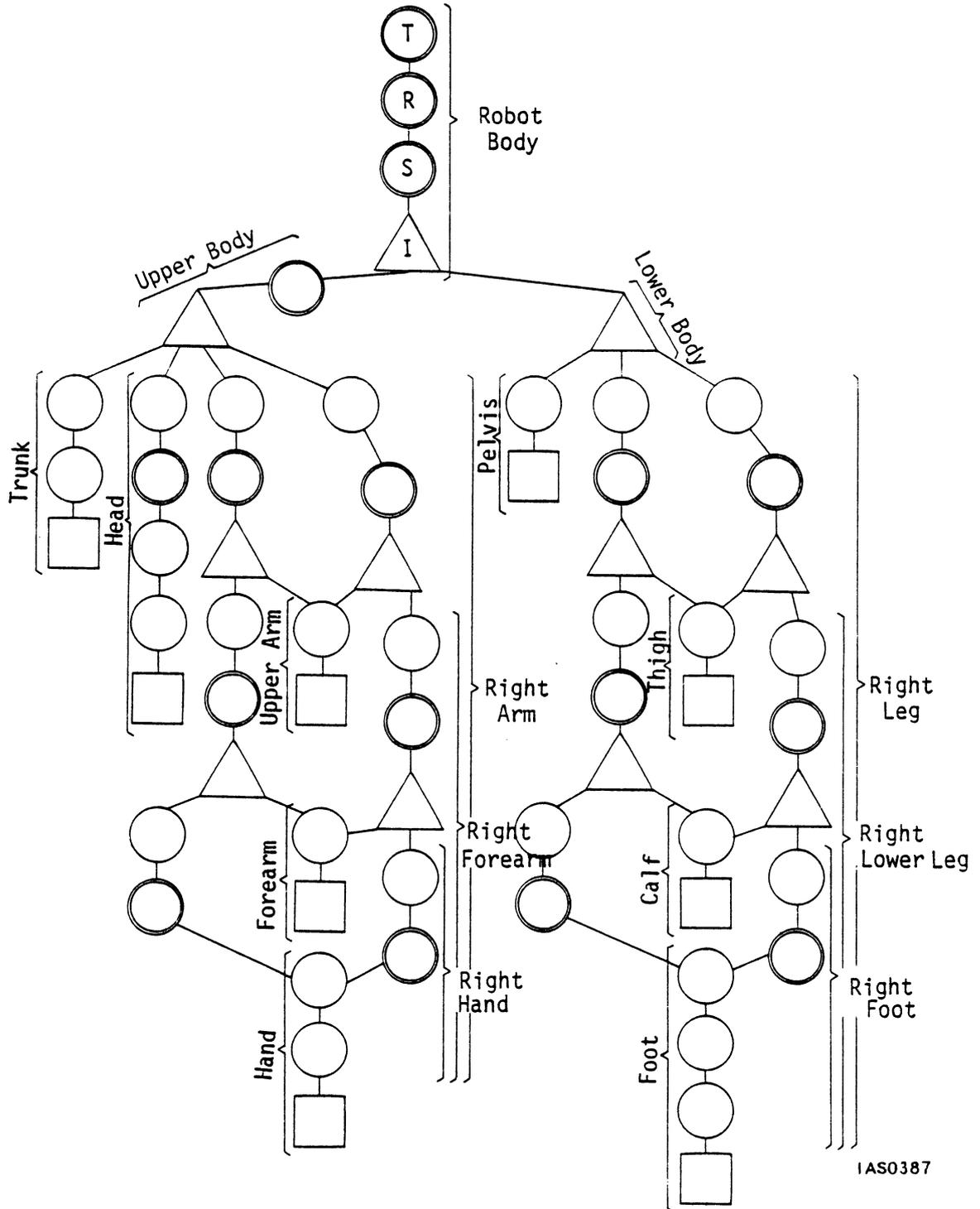


Figure 5. Possible BEGIN_STRUCTURE...END_STRUCTUREs for Robot

With this in mind, begin at the top branches of the display tree and code down.

The top of the display tree can be coded in one BEGIN_STRUCTURE...
END_STRUCTURE:

```
Robot := BEGIN_STRUCTURE
      Tran := TRANSLATE BY 0,0;
      Rot := ROTATE 0;
      Scale := SCALE BY .075;
             INSTANCE OF Upper_Body, Lower_Body;
      END_STRUCTURE;
```

Remember that PS 300 screen coordinates are +1 to -1 units, so the screen is 2 units across. Because of Robot's dimensions, he is scaled so he is viewable on the display screen.

Each of these top nodes could have been coded explicitly since three of the four nodes are interactive and must be named regardless:

```
TranRobot := TRANSLATE BY 0,0 APPLIED TO RotRobot;
RotRobot := ROTATE 0 APPLIED TO ScaleRobot;
ScaleRobot := SCALE BY .075 APPLIED TO Robot;
Robot := INSTANCE OF Upper_Body, Lower_Body;
```

However, the BEGIN_STRUCTURE...END_STRUCTURE saves naming one node, the instance node, within the structure and groups all the interactive nodes under a single name: Robot. The tradeoff, an additional instance node, is not prohibitive in this case.

Next, Upper_Body groups all the upper body parts into one entity which can be rotated interactively.

```
Upper_Body := BEGIN_STRUCTURE;
             Rot := ROTATE 0;
             INSTANCE OF Head, Right_Arm, Left_Arm, Trunk;
             END_STRUCTURE;
```

The above could be coded explicitly:

```
Rot_Upper_Body := ROTATE 0 APPLIED TO Upper_Body;
Upper_Body := INSTANCE OF Head, Right_Arm, Left_Arm, Trunk;
```

However, it may be more convenient to group them into a BEGIN_STRUCTURE... END_STRUCTURE under a single name. This way, "Robot" could subsequently be coded as an instance of Upper_Body and Lower_Body instead of an instance of Rot.Upper_Body and Lower_Body.

You could choose to code explicitly for efficiency. However, in this example, the convenience is worth the extra instance node.

Both the trunk and head could be defined in BEGIN_STRUCTURE... END_STRUCTUREs.

```
Trunk := BEGIN_STRUCTURE
        TRANSLATE BY 0,6;
        SCALE BY 2,3,1 APPLIED TO Cylinder;
        END_STRUCTURE;

Head := BEGIN_STRUCTURE
        TRANSLATE BY 0,6;
Rot :=  ROTATE 0;
        SCALE BY 1,2,1;
        TRANSLATE BY 0,1 APPLIED TO Sphere;
        END_STRUCTURE;
```

Notice that only the rotation node (Head.Rot) is named so values may be sent interactively to move the head.

Now work through the rest of the code. When you are finished, you can compare your code with the following:

Begin with Right_Arm and code top down. The circled display tree indicates the arm may be coded in three BEGIN_STRUCTURE...END_STRUCTUREs.

```
Right_Arm := BEGIN_STRUCTURE;
             TRANSLATE BY -2.5,6;
Rot:=      ROTATE 0;
             INSTANCE OF Upper_Arm, Right_Forearm;
             END_STRUCTURE;

Right_Forearm := BEGIN_STRUCTURE;
                 TRANSLATE BY 0,-4;
Rot :=        ROTATE 0;
                 INSTANCE OF Forearm, Right_Hand;
                 END_STRUCTURE;
```

Since Hand has already been coded, Right_Hand is defined as:

```
Right_Hand := BEGIN_STRUCTURE;
              TRANSLATE BY 0,-3;
  Rot :=      ROTATE 0 APPLIED TO HAND;
              END_STRUCTURE;
```

Left_Arm is coded like Right_Arm but with different transformation values and name changes:

```
Left_Arm := BEGIN_STRUCTURE;
             TRANSLATE BY 2.5,6;
  Rot :=      ROTATE 0;
             INSTANCE OF Upper_Arm, Left_Forearm;
             END_STRUCTURE;
```

The pattern is similar for Left_Forearm. Note that it references Forearm and Left_Hand, which have both been defined:

```
Left_Forearm := BEGIN_STRUCTURE;
                 TRANSLATE BY 0,-4;
  Rot :=        ROTATE 0;
                 INSTANCE OF Forearm, Left_Hand;
                 END_STRUCTURE;
```

Notice in these two structures, the instance nodes are defined on a separate line from the rotation node because there is more than one instance node being referenced.

Since Hand has already been coded, Left_Hand is defined as:

```
Left_Hand := BEGIN_STRUCTURE;
              TRANSLATE BY 0,-3;
  Rot :=      ROTATE 0 APPLIED TO HAND;
              END_STRUCTURE;
```

Now the lower half of Robot can be coded. First, group the pieces of Robot's lower body together in one line of code:

```
Lower_Body := INSTANCE OF Pelvis, Right_Leg, Left_Leg;
```

Then define each piece. Pelvis can be coded explicitly:

```
Pelvis := SCALE BY 2,1,1 APPLIED TO Cylinder;
```

The legs, like the arms, consist of BEGIN_STRUCTURE...END_STRUCTUREs:

```
Right_Leg := BEGIN_STRUCTURE;
             TRANSLATE BY -1,-2;
  Rot :=     ROTATE 0;
             INSTANCE OF Thigh, Right_Lower_Leg;
             END_STRUCTURE;

Right_Lower_Leg := BEGIN_STRUCTURE;
                  TRANSLATE BY 0,-5;
  Rot :=         ROTATE 0;
                  INSTANCE OF Calf, Right_Foot;
                  END_STRUCTURE;

Right_Foot :=  BEGIN_STRUCTURE;
               TRANSLATE BY 0,-5.5;
  Rot :=      ROTATE 0;
               TRANSLATE BY 0,0,-.5 APPLIED TO Foot;
               END_STRUCTURE;

Left_Leg :=   BEGIN_STRUCTURE;
              TRANSLATE BY 1,-2;
  Rot :=     ROTATE 0;
              INSTANCE OF Thigh, Left_Lower_Leg;
              END_STRUCTURE;

Left_Lower_Leg := BEGIN_STRUCTURE;
                  TRANSLATE BY 0,-5;
  Rot :=         ROTATE 0;
                  INSTANCE OF Calf, Left_Foot;
                  END_STRUCTURE;

Left_Foot :=  BEGIN_STRUCTURE;
               TRANSLATE BY 0,-5.5;
  Rot :=      ROTATE 0;
               TRANSLATE BY 0,0,-.5 APPLIED TO Foot;
               END_STRUCTURE;
```

USING IMMEDIATE ACTION COMMANDS

Immediate action commands are executed immediately when they are received by the system. This kind of command cannot be named. For example:

```
DISPLAY Smallstar;
```

is a command that causes the model (data structure) named Smallstar to be displayed. The data structure Smallstar already exists in Mass Memory; the DISPLAY command creates no additional data structure. Naming the display command:

```
Name := DISPLAY Smallstar;
```

would cause an error message to appear.

Immediate action commands can be divided into three types:

1. Those used with function networks. (For more information on these, refer to the "Function Networks I" module.)

```
CONNECT  
DISCONNECT  
SEND  
STORE
```

2. General commands

```
ALLOCATE PLOTTER  
DEALLOCATE PLOTTER
```

Allow you to specify up to four Versatec plotters to allocate for hardcopy or deallocate after hardcopy.

```
BEGIN...END
```

Defines a batch of commands so they appear to execute simultaneously.

```
COMMAND STATUS
```

Reports current level to which BEGIN...END and BEGIN_STRUCTURE...END_STRUCTURE commands are nested.

DEFINE

Sets up internal conversions when using different data base units together (such as inches and centimeters) or angular measurements other than degrees (such as radians).

DISPLAY

REMOVE

Cause objects either to appear on or disappear from the screen.

FORGET (data structures)

Removes a name from use while leaving the data structure associated with the name in place.

DELETE

Removes a name and its associated data structure from use.

INITIALIZE

Clears all user-defined structures from Mass Memory.

OPTIMIZE STRUCTURE...END OPTIMIZE

Places the PS 300 in, and removes it from, "optimization mode," which minimizes Display Processor traversal time for structures created in this mode.

!RESET

Equivalent to entering enough "END;" commands to terminate any unENDED BEGIN_STRUCTURE...END_STRUCTUREs (i.e., more BEGIN_STRUCTURE statements than END_STRUCTURE statements, creating an "unended" structure).

3. Structuring commands

These "hybrid" commands have characteristics of both immediate action commands and data structuring commands. Like data structuring commands, they create data tree nodes in Mass Memory by inserting nodes into an already-named display tree. (Note that the SEND command sends new values to existing display tree nodes).

Unlike data structuring commands, these commands cannot be named (the nodes they create derive their names or traversal paths from existing nodes), so they are considered immediate-action commands.

FOLLOW WITH
REMOVE FOLLOWER

The first command follows a named operation node with another operation node. The second command removes this added operation node.

INCLUDE
REMOVE FROM

The first command modifies an existing INSTANCE display tree node by including one named display tree in a named INSTANCE display tree. The second command removes this added structure.

PREFIX WITH
REMOVE PREFIX

The first command prefixes a named display tree with an operation node. The second command removes the prefixed node.

SEND

This command sends a value to a display tree node, as well as to a function instance or variable.

Immediate action commands are not only used for interactions such as displaying or deleting an object on the screen. They are also useful for making experimental or temporary changes to a model's display structure in Mass Memory. For example, if you wanted to change the view of a given model, you could add operation nodes to the model's display structure (in Command Mode) by using the FOLLOW WITH command. Should the view prove undesirable, you could remove these nodes with the REMOVE FOLLOWER command.

If you liked the effect and wanted the nodes permanently, you could edit the model's display tree file (in Terminal Emulator Mode) adding data structuring commands.

ENTERING CODE IN THE PS 300

When you have determined the commands needed to create the display tree, enter them into the PS 300. You could enter them directly, as you did in the "Hands-On Experience" module, but this means they will not be saved.

To retain a copy of the code for further use, enter the commands into a text file on your host computer. The procedure for this varies according to the host. Refer to your host documentation for details.

The file can then be downloaded to the PS 300. For details on how to do this, refer to the section on PS 300 and Host Communication in the *User Operation and Communication Guide* in Volume 1.

When the transfer is completed, you can display the model using the DISPLAY command.

For example, transfer the Robot file to the PS 300 and display the Robot model by entering:

```
CTRL LINE/LOCAL  
@@Display Robot;
```

Command Summary

Now that you are familiar with how the commands work, the *Command Summary*, Volume 3A should be the only reference you need. The Command Summary serves as a quick, complete reference on all available commands. Commands are listed alphabetically. Acceptable abbreviations, formal command syntax, and information on data types for parameters are supplied.

Graphics Support Routines

The Graphics Support Routines (GSRs) are a collection of procedures which are used to improve speed and efficiency in communications between the host computer and the PS 300. They reside in the host computer.

Most of the PS 300 commands have a corresponding Graphics Support Routine. When you call one of these routines, the corresponding PS 300 command is sent in BINARY directly to the PS 300 to be executed.

This improves efficiency in that data is received in the format required by the PS 300--ready for direct interpretation. Because the data requires no further parsing, less time is required by the PS 300 to interpret commands.

Keep in mind that the GSR software does not replace the PS 300 Command Language. It is an alternative way to invoke it. For details on the Graphics Support Routines, refer to the *Programmer Reference Manual*, Volume 3B.

SUMMARY

All PS 300 commands:

- End with a semi-colon.
- Are free-formatted.
- Have a long and short form.
- Can consist of upper and/or lower case letters.

Two kinds of PS 300 commands are needed to create a model in PS 300 Mass Memory and display it on the screen: data structuring commands and immediate action commands.

Immediate action commands are executed immediately, and cannot be named by the user. These are used in function networks, for general system operations such as displaying or removing an object from the PS 300 screen, and for temporary modifications to existing display trees in Mass Memory.

Display trees are initially created in Mass Memory using data structuring commands. All data structuring commands must be named, either explicitly or using BEGIN_STRUCTURE...END_STRUCTUREs.

Whichever method you use, the following naming conventions are used.

- A name can be up to 240 characters and consist of any alphanumeric combination as long as it begins with a letter.
- A name can include underscore characters (_), a dollar sign (\$), or a period (.). However, using periods may be confusing because the PS 300 automatically inserts periods in the name of nodes contained in a BEGIN_STRUCTURE... END_STRUCTURE.
- A name cannot contain a space or other delimiter (<or>, tab, etc.).
- A name is followed by a colon, an equal sign, and the command to be associated with the name.
- Other PS 300 commands can be used in conjunction with the naming of data structures:
 - (Naming of Display Data Structures) Command
 - The (name := nil) Command
 - The FORget command
 - The DELete command

Most display trees are coded using a combination of explicit naming and BEGIN_STRUCTURE...END_STRUCTUREs. In general, BEGIN_STRUCTURE...END_STRUCTUREs:

- Group associated nodes together into identifiable parts of a model.
- Reflect the order of operations shown in the display tree.
- Eliminate the unnecessary naming of nodes: nodes within a structure are only named if they need to be referenced, i.e., are interactive.

BEGIN_STRUCTURE...END_STRUCTURE commands follow certain conventions:

- BEGIN_STRUCTURE...END_STRUCTURE always creates an INSTANCE node.
- OPERATION commands inside a BEGIN_STRUCTURE...END_STRUCTURE apply to everything below in the structure unless they are explicitly APPLIED TO other structures.
- If an OPERATION node in a BEGIN_STRUCTURE...END_STRUCTURE is to be applied to more than one branch of the hierarchy, the PS 300 creates an INSTANCE node below the OPERATION node, and the INSTANCE node points to the appropriate branches.
- In a BEGIN_STRUCTURE...END_STRUCTURE, the PS 300 automatically prefixes the name of a "user-named" node with the name of the BEGIN_STRUCTURE...END_STRUCTURE. The prefixed (hierarchical) name is first, followed by a period (.) and the "user" name.
- END_STRUCTURE does not create any data node, but merely indicates to the PS 300 that the BEGIN_STRUCTURE...END_STRUCTURE is finished.
- Nested BEGIN_STRUCTURE...END_STRUCTUREs can be considered as terminal nodes in the encompassing BEGIN_STRUCTURE...END_STRUCTURE. In other words, nothing in the nested structure is applied to anything in the rest of the encompassing BEGIN_STRUCTURE...END_STRUCTURE.

A model's display tree is coded in a text file or an application program on the host computer and then downloaded to the PS 300. For quicker communications between host and PS 300, use the PS 300 Graphic Support Routines. Use the *Command Summary* as an easy reference manual of all available commands.

For information on how to connect function networks into a model's display tree, to manipulate the model interactively, refer to the "Function Networks I" module.

FUNCTION NETWORKS I

INTERACTION WITH A MODEL

CONTENTS

INTRODUCTION	1
OBJECTIVES	2
PREREQUISITES	2
CONVERTING INPUT DEVICE VALUES TO UPDATE AN INTERACTION NODE	3
Exercise	16
ADDING FURTHER INTERACTION: ROTATION IN OTHER DIMENSIONS	19
Exercise	21
EXPAND THE NETWORK FOR OTHER KINDS OF INTERACTION	23
Exercise	26
Exercise	28
A CLOCK FUNCTION AS AN ALTERNATE SOURCE OF INPUT FOR THE NETWORK	29
Exercise	31
SUMMARY	34

FUNCTION NETWORKS I

ILLUSTRATIONS

Figure 1.	Sample Function Network	1
Figure 2.	The "Black Box"	3
Figure 3.	Robot Display Tree	4
Figure 4.	F:YROTATE	7
Figure 5.	Possible Y Rotation Network	7
Figure 6.	Y Rotation Network With Multiplier	8
Figure 7.	Adding an Accumulator	9
Figure 8.	Tracing Dial Values (Part 1)	9
Figure 9.	Tracing Dial Values (Part 2)	10
Figure 10.	Tracing Dial Values (Part 3)	10
Figure 11.	F:MUL	13
Figure 12.	F:DYROTATE	14
Figure 13.	F:DYROTATE Function Network	15
Figure 14.	"Spinner" Function Diagram	16
Figure 15.	Completed Function Network for X, Y and Z Rotation	18
Figure 16.	Modified Display Tree With Three Rotate Nodes	19
Figure 17.	F:YROTATE Network	20
Figure 18.	Common Accumulator for Rotate Functions	20
Figure 19.	Suggested Configuration for Dials	24
Figure 20.	Translate Network	25
Figure 21.	Network for Uniform Scaling	27
Figure 22.	F:CLFRAMES	29
Figure 23.	F:CLFRAMES as Input Source for Y Rotation	31
Figure 24.	A Network That Toggles	32
Figure 25.	A More Efficient Toggle Switch	33
Figure 26.	The Completed Network	34

This module illustrates how to construct simple function networks. Function networks allow you to interact with a model you have created for display.

The first steps to using function networks were detailed in the "Modeling" and "PS 300" modules. There, you analyzed a model (the robot) for movement, allowed for that movement by including interaction nodes in the display tree, and named those nodes when you coded the model so they could be accessed.

Interactive nodes can be accessed using function networks. For example, in the "Hands-On Experience" module, you used a simple function network to rotate a star on the screen (Figure 1).

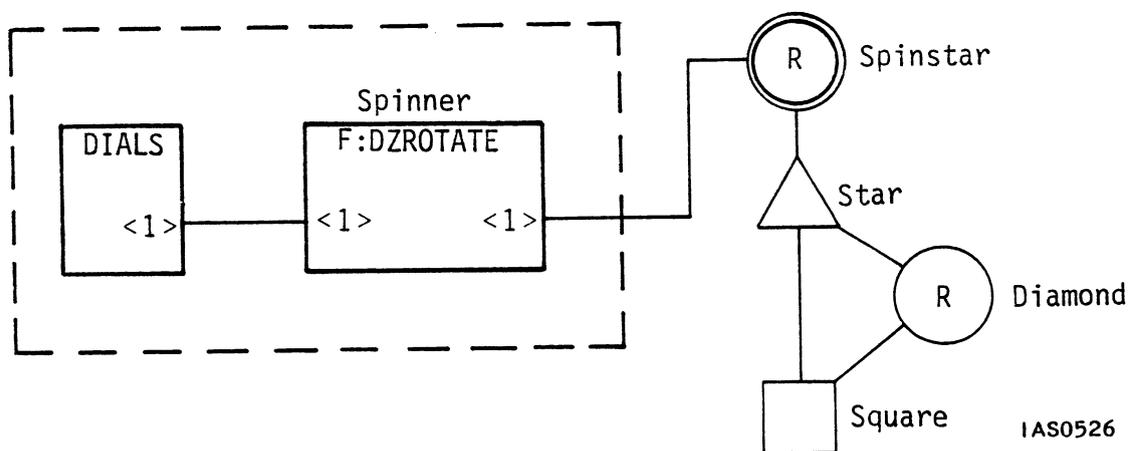


Figure 1. Sample Function Network

This module will illustrate in greater detail how this kind of function network operates.

You will also learn how to use PS 300 interactive devices to move a more complex model--the robot. When the robot display tree was initially coded, zero values were assigned to the interaction nodes. In the case of rotating the model, this meant the model would rotate zero degrees from initial position when first displayed. To rotate the model, you can create a function network to send a non-zero value to the rotate node. Specifically, this function network takes values from input devices and transforms them into values the interaction nodes can accept.

2 - FUNCTION NETWORKS I

Part of this process entails selecting the appropriate PS 300 functions and linking them together into a function network. This function network may contain additional functions which perform other kinds of tasks, such as accumulating values which are not large enough.

As you did when you created a display tree for your model, you will first create a diagram of your function network and then code from that. Creating a diagram of the network first allows you to modify it easily and detect errors before they become bugs in the code.

OBJECTIVES

In this module you will learn how to:

- Select functions which will convert input device values into values that can update an interaction node
- Add functions to the network for rotation in all three dimensions
- Expand the network for other kinds of interaction (scaling and translating)
- Use a CLOCK function as an alternate source of input for the network.

PREREQUISITES

Before beginning this module, you should be familiar with the concepts presented in the "Modeling" and "PS 300 Command Language" modules.

CONVERTING INPUT DEVICE VALUES TO UPDATE AN INTERACTION NODE

The first step to selecting the appropriate function to convert input values into values that can update an interaction node is to identify the type of values needed by the node. To understand this, look at the the most common graphics transformations--rotation, scaling, and translation.

Rotations and scales are done with 3×3 matrices; translations are specified with a 2- or 3-dimensional vector. It makes sense, then, that the type of data used by a rotation or scale node is a 3×3 matrix, and the data type for a translation node is a vector.

Your task, if you are trying to rotate part of a model, is to find a way to make an input device, such as a dial, send the correct 3×3 matrices to a rotate node. In this module, this process will be represented by a "black box" (Figure 2) that takes one kind of value and changes it into another kind.

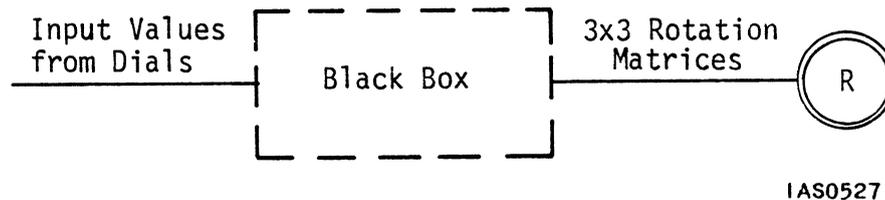


Figure 2. The "Black Box"

In the "Hands-On Experience" module, you created Diamond by specifying a 45 degree rotation of Square. You did not need to work out what the 3×3 matrix for 45 degrees was. Whenever you use a command to create a rotate or scale node (such as Diamond), you only have to specify an angle using a real number value and the PS 300 automatically creates the associated 3×3 matrix.

Once the node is created, however, you can only update it with the type of data it accepts--in this case, a 3×3 matrix. For example, look at the robot display tree again (Figure 3) the names for the nodes are supplied so you can refer to them.

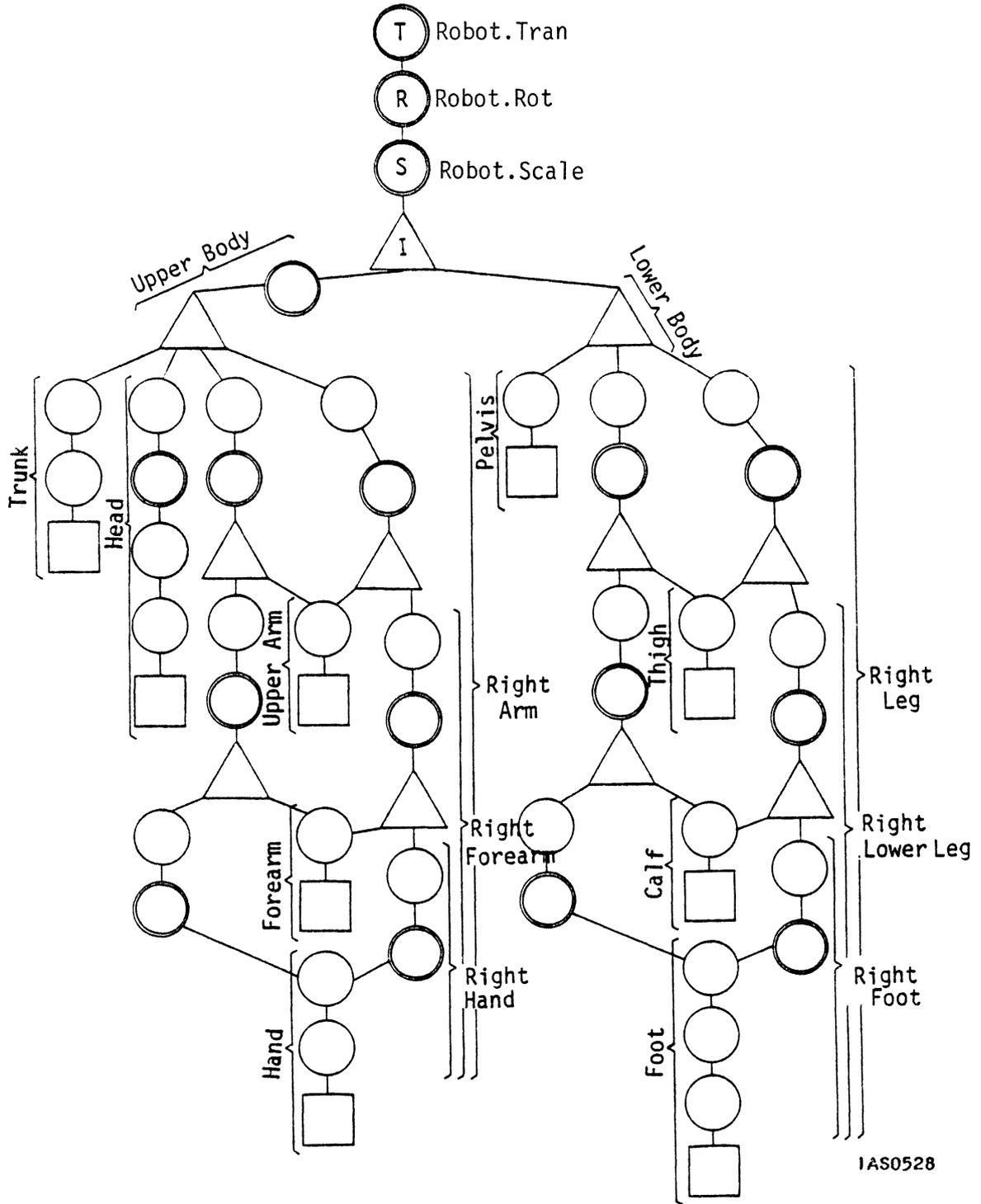


Figure 3. Robot Display Tree

To move the left arm, you must send a rotation matrix to the node named LeftArm.Rot.

Look at the other interaction nodes. Almost all of them are rotations except for the translate node above robot (Robot.Tran). All of these rotate nodes accept 3×3 rotation matrices, and Robot.Tran requires vectors.

One other node in the robot display tree is interactive: Robot.Scale, the scale node above Robot. Unlike the rotation nodes, it currently contains a non-zero value (.05). Since it has been named, you can reference it and connect functions to it. This will allow you to change the values it contains, interactively making the robot larger or smaller.

You can control whether or not any operation node in the display tree will be processed. Sending a Boolean value FALSE to input <-1> of any operation node will turn off the action specified in that node. Sending a TRUE to input <-1> will turn it back on. The default is ON.

Once you know what values you need to produce for interaction nodes (output from the black box), you should identify the type of values produced from the input device (input to the black box).

The twelve function keys across the top of the PS 300 keyboard produce discrete integer values from 1 to 36. Consequently, they are a useful source of input for discrete tasks such as changing states or "modes".

The data tablet is commonly used for digitizing (sketching or tracing from a drawing), making menu selections, and picking. The values it produces are XY vectors with fractional values between plus and minus 1.

This module focuses on the dials. Dials produce a stream of small, incremental real numbers. This means the dials are well-suited for producing smooth motion, so they are often used in controlling the three common transformations: translations, rotations, and scaling.

When you turn a dial clockwise, it sends out a stream of fractional values (called *delta values*) that sum to 1 after a complete rotation of the dial. If you turn it a full turn counterclockwise, the values sum to -1.

If the delta value is .1, every time you turn the dial one-tenth of the way around, it produces a .1. If you turned the dial one-twentieth of a turn, it will not produce a value. By default, a dial is set to produce a delta of about .001. This makes the dial extremely responsive. Only a slight movement of the dial will generate a value.

The delta values produced by the dials do not accumulate. In other words, the dial does not send out .001 the first click, .002 the next, and so on, ending with 1.0 after one complete turn. After one complete turn, the dial still sends out the same delta value it did when it was turned enough to produce a value. This fact is extremely important to remember when you are designing function networks.

Once you have identified the input source and the type of values it generates, you can use the *Command Summary* in Volume 3A to identify the appropriate function(s) to convert incoming values from an input device into appropriate output. For rotations, the function(s) must convert real numbers sent out by the dials into 3×3 matrices needed for the interaction nodes.

The *Command Summary* contains a description of each command associated with a node (INSTANCE OF, ROTATE, VECTOR LIST, and so on). In each description is a list of associated functions that produce values the node can accept. For example, if you look up ROTATE, you will find these associated functions:

F:MATRIX3, F:XROTATE, F:YROTATE, F:ZROTATE, F:DXROTATE,
F:DYROTATE, F:DZROTATE, F:SCALE, F:DSCALE

All of these functions produce 3×3 matrices, so, in theory, all of them can be connected to a rotate node. However, those most closely associated with rotate nodes are the ones with ROTATE in their names. These are the candidate functions for the black box.

The *Command Summary* lists the following associated functions in its description of the SCALE and TRANSLATE operation nodes:

SCALE: F:MATRIX3, F:XROTATE, F:YROTATE, F:ZROTATE,
F:DXROTATE, F:DYROTATE, F:DZROTATE, F:SCALE, F:DSCALE

TRANSLATE: F:XVECTOR, F:YVECTOR, F:ZVECTOR

As with the ROTATE command, where a number of associated functions produce the type of output needed, the name indicates the most likely functions to use. So if you wanted to send values to a scale node, you should use F:SCALE and F:DSCALE (for the differences between F:SCALE and F:DSCALE, refer to the *Function Summary*).

Note that the associated functions for TRANSLATE do not have "TRAN" in their names. Since TRANSLATE nodes accept 3D vectors as input, the associated functions are ones that generate 3D vectors--F:XVECTOR, F:YVECTOR, and F:ZVECTOR.

To evaluate the functions themselves, look them up in the *Function Summary* in Volume 3A. The following example illustrates how functions work, and how the *Function Summary* presents the information.

Assume you want to rotate Robot in Y, that is, to make it spin around around its vertical axis. The node to interact with is the rotate node named Robot.Rot in the display tree. Y rotations are associated with the functions F:YROTATE and F:DYROTATE. Under F:YROTATE in the *Function Summary* you will find a diagram like the one in Figure 4.

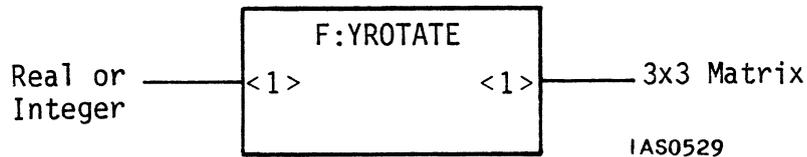


Figure 4. F:YROTATE

The diagram indicates that this function has one input on the left and one output on the right. It can accept real values which represent degrees of rotation on its input and send out 3x3 rotation matrices as output values.

Like all PS 300 functions, it waits until an input value has arrived, performs computations, and outputs the value. It is capable of consuming a steady stream of input values and producing a steady stream of outputs.

F:YROTATE seems to be the likely candidate for the black box. It accepts real numbers so it can be connected to the dials, and it produces 3x3 matrices to send to the rotation node. Look at example shown in Figure 5.

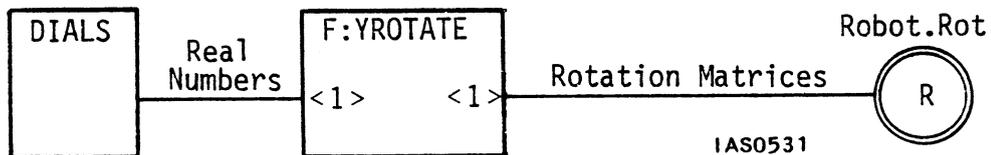


Figure 5. Possible Y Rotation Network

If you were to test this function with a stream of values from the dials, you would discover several facts. The first is that the values from the dial are very tiny. Each one supplies only a fraction of a degree of rotation to F:YROTATE, so the corresponding matrices that F:YROTATE sends out specify almost insignificant amounts of rotation in the model on the screen. You need a way to multiply the effect the dial values have. Adding a new function can do that (see Figure 6).

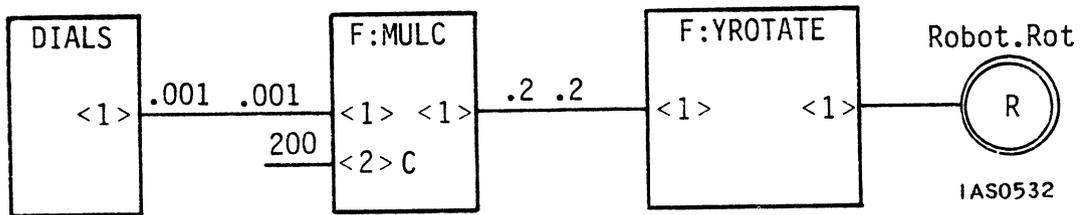


Figure 6. Y Rotation Network With Multiplier

F:MULC is a multiplying function that takes any value it receives on input <1> and multiplies it by the *constant value* on input <2>. Many PS 300 functions have constant inputs. Unlike regular inputs, called *active inputs*, constant inputs never consume the values on them. If you place a large number on input <2> (200 is the value shown in the diagram), then each incoming dial value will be converted to a value 200 times greater. That will specify noticeable amounts of rotation for F:YROTATE. F:MULC converts a .001 from the dials to 0.2.

Continue to trace successive values through this modified network. When F:YROTATE receives the 0.2 from F:MULC, it will immediately send out a matrix to Robot.Rot that will rotate Robot 0.2 degrees.

The dial produces only a stream of incremental delta values--each one is about 0.2. As F:YROTATE receives these, it produces a stream of matrices, each corresponding to about .2 degrees of rotation. But nothing greater than about a fifth of a degree of rotation ever occurs. The effect of this is that the robot rotates only a small amount and stays there. It may even look like the robot is not responding to the dials at all.

What is needed is a way to accumulate values, so the first delta value causes .02 degree of rotation, the second value .04 degrees, and so on. This calls for another modification of the network. Figure 7 shows one method of adding an accumulator.

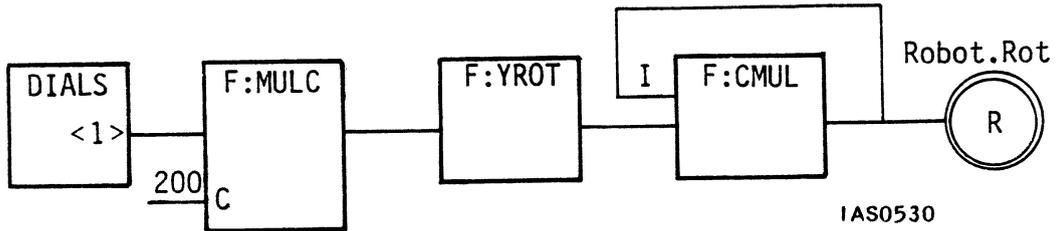


Figure 7. Adding an Accumulator

Values from F:YROTATE can accumulate using a multiply function (F:CMUL) as the accumulator. F:CMUL is the same as F:MULC, except its first input is constant. To fire, this function needs to be "primed" the same way F:MULC did. Place an identity matrix on input <1>. This ensures that F:YROTATE will produce a product. When the first incoming value from F:YROTATE arrives at input <2> F:CMUL, it will be multiplied by the matrix waiting on input <1>.

The product of these two matrices goes to update the rotation node. It also goes back to F:CMUL input <1> to replace the identity matrix that was there. So the next rotation matrix to arrive on F:CMUL input <2> gets multiplied by the accumulated matrix, not by the identity matrix first placed there.

This whole process repeats each time F:CMUL fires. A new matrix, containing the accumulated rotations is continually being sent back to input <1> as each new matrix is output from the function.

Figures 8, 9 and 10 trace a stream of three or four values from the dial through the network to see if the modifications that are added produce the desired matrices.

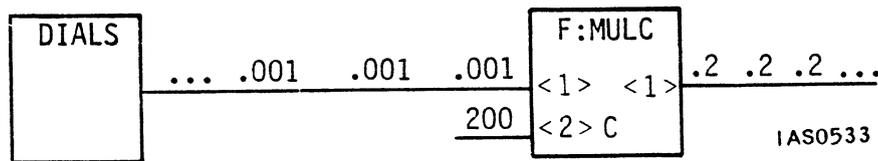


Figure 8. Tracing Dial Values (Part 1)

As shown in Figure 8, F:MULC multiplies the first dial value by 200 to produce a 0.2. This value triggers F:YROTATE to produce a rotation matrix for one-fifth of a degree of rotation. That travels to F:CMUL, where it will be multiplied by an identity matrix (I) on input <1> (see Figure 9).

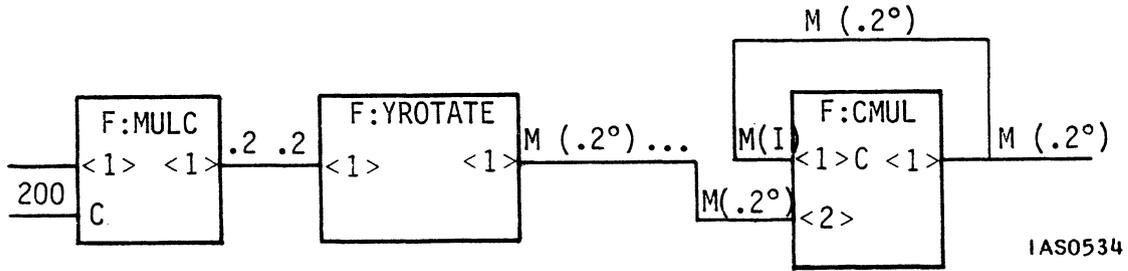


Figure 9. Tracing Dial Values (Part 2)

Multiplying a matrix by an identity matrix has the same effect that multiplying by 1 was on numbers. The matrix that first arrives on F:CMUL input <2> from F:YROTATE is multiplied by the identity matrix on input <1> and output unchanged from the function to the rotate node. This matrix also travels back to F:CMUL constant input <1> and replaces the identity matrix that was there.

The second dial value goes through the exact same process, causing F:MULC to send out a 0.2, which causes F:YROTATE to send out another matrix for 0.2 degree of rotation. But this second matrix gets multiplied, not by the identity matrix as the first matrix did, but by the most recent value sitting on input <1>. In this case, that value is a matrix for .2 degrees of rotation (Figure 10).

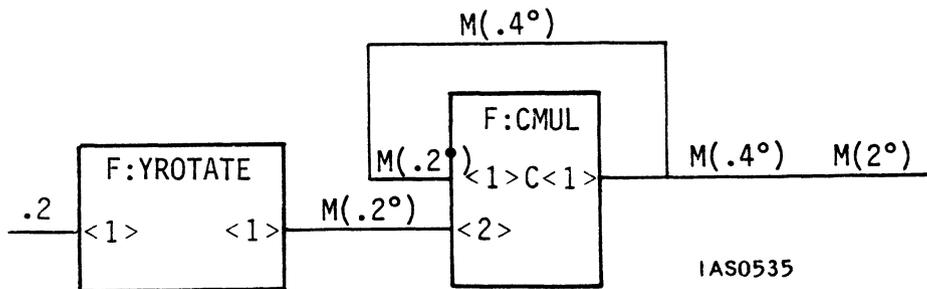


Figure 10. Tracing Dial Values (Part 3)

The second product from F:CMUL is a new accumulated matrix for .4 degrees of rotation that updates the rotate node and goes back to replace the .2-degree matrix on the F:CMUL constant input. Each time a new 0.2-degree rotation matrix comes from F:YROTATE, this process repeats, and a new cumulative rotation matrix goes around to the F:CMUL constant input. So each time F:CMUL fires, the matrix sent to Robot.Rot will specify a little more rotation than the one before it.

Once this network is implemented, it handles input values from the dials so quickly that the model will appear to rotate in real time. It also provides the expected results when the dial is turned the other way: it generates small, negative values which cause F:YROTATE to output rotation matrices for negative rotation. The result is that the model rotates in the opposite direction.

Examine the diagram you have so far. It illustrates several important facts about functions. One of the functions, F:YROTATE, is a *data conversion* function. It takes one type of input and produces a different type. Other functions do not do this. For example, F:ADD is an *arithmetic operation* function. It adds two incoming values and produces the same type of data it receives as input. F:MULC and F:CMUL, used in this network, are also examples of functions that do arithmetic operations. Other functions perform *logical operations* or select and route data. The classes of functions are outlined below:

- Data Conversion

Data conversion functions combine vectors into matrices, extract vectors from matrices; form vectors from real numbers, round or truncate real numbers to integers, float integers to equivalent real numbers, make printable characters and convert character strings to a string of integers.

- Arithmetic and Logical

These functions perform all arithmetic operations (add, divide, subtract, multiply, square root, sine, and cosine) and logical operations (and, or, exclusive-or, and complement).

- Comparison

Comparison functions test whether values are greater than, less than, equal to, not equal to, greater than or equal to, and less than or equal to other values.

- Data Selection and Manipulation

These functions are used to selectively switch functions, choose outputs, and route data.

- Viewing Transformation

Viewing transformation functions connect to viewing operation nodes in display trees to change line-of-sight, window size, and viewing angle, interactively.

- Object Transformation

Object transformation functions connect to modeling operation nodes in display trees to interactively rotate, translate, and scale objects.

- Character Transformation

These functions are used to interactively position, rotate, and scale text.

- Data Input and Output

These functions set up and control the interactive devices dials, function keys, function buttons, data tablet, and keyboard, and output values to the optional LED labels that several of the devices have.

- Miscellaneous

Other functions set up and control picking, clocking, timing, and synchronizing operations.

Notice from the function network diagrams that values flow from left to right, with the input device usually situated "upstream" at the extreme left and the destination for the values, the nodes, "downstream" on the extreme right.

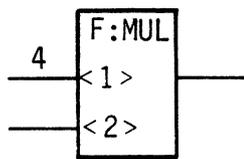
Despite this general direction of flow, values can be routed virtually anywhere in the network. A function's output can be connected to the input of any other function, including itself, as F:CMUL demonstrates.

In addition, an output can be connected to more than one destination. Again, F:CMUL illustrates this. Its output goes to a rotate node and also back to its own input. Similarly, an input can be fed by more than one source.

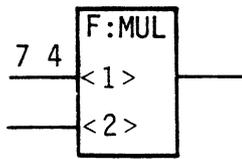
As mentioned, functions have two kinds of inputs: active and constant. Inputs in the *Function Summary* diagrams are active unless they are marked with a C. Both F:MULC and F:CMUL have constant inputs. Values coming in on an active input are consumed as the function executes, clearing the input for the arrival of a new value. If the function cannot execute yet because values for other inputs have not arrived, active input values will queue up, waiting their turn to be consumed.

Values on constant inputs stay on the input; they are not consumed when the function executes. If another value arrives, it does not queue up; it replaces the value that was there. In effect, then, there can only be one value at a time on a constant input. Constant inputs are useful in a situation like the previous network, where F:MULC is used to multiply a stream of values coming in on one input by the same constant factor.

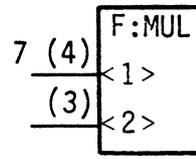
Contrast this with the multiplying function F:MUL, which has two active inputs (Figure 11).



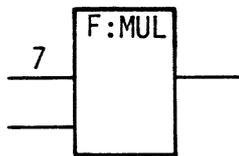
Ⓐ Value Arrives on Input 1.



Ⓑ Second Value Arrives on Input 1 and Queues Up.



Ⓒ Value Arrives on Input 2 Multiplies the First Value Received on Input 1 - Product is Fired Out.



Ⓓ After Firing, Value on Input 1 Waits for a Value to Appear on Input 2.

IAS0550

Figure 11. F:MUL

If you sent 200 to input <2> of F:MUL, it would remain there until the first value from the dial arrived on input <1>. Then the function would send out the product and the two input values would disappear. A second value from the dial would arrive on input <1> and wait, because there would be no value on input <2> to multiply it by. The third, fourth, and all succeeding values from the dial would queue up behind it on input <1>, all waiting for their turn to be multiplied. To keep F:MUL working, you would need to supply it with a steady stream of fresh 200s for input <2>. Obviously, it is easier to use a constant input for this, as with F:MULC.

This module employs PS 300 functions with fixed constant or active inputs. At times, however, it will be useful to specify whether an input to a function is active or constant. Refer to the *Command Summary* for information on the SETUP CNESS command, which allows you to determine whether or not an input is constant or active.

As a rule, PS 300 functions execute only when all the inputs have values. Some functions like F:ZVECTOR have only one input, so they fire whenever the correct value arrives on it. Others, such as F:MUL, require a value on each of two inputs. F:MULC and F:CMUL are this way, except that you can place a single value on the constant input and then control the function's firing by sending or not sending values to the active input.

Many PS 300 functions have two or more inputs which must be accounted for. Some function inputs have default values that do not need to be primed before using the function. The descriptions in the *Function Summary* detail which inputs are constant and which have default values provided for them.

So far you have a function network to serve as the black box for rotating the robot. there is still one other function to evaluate, F:DYROTATE. F:DYROTATE is shown in Figure 12.

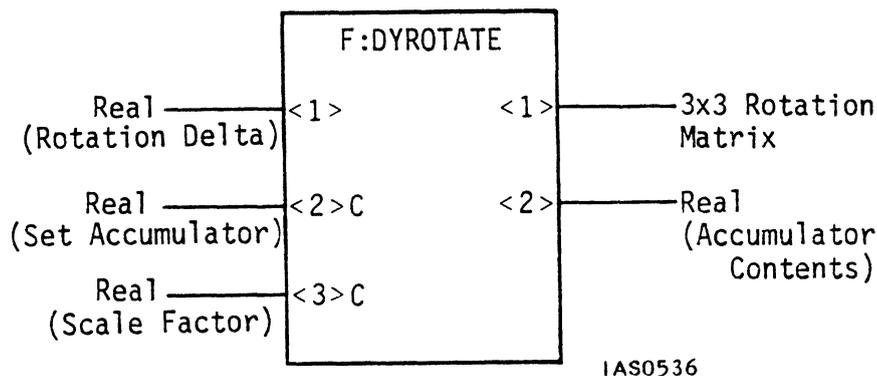


Figure 12. F:DYROTATE

This function has three inputs. Input <1> takes values directly from the dials. Input <3>, a constant input, holds a magnifying value and does the same thing F:MULC does in the F:YROTATE network; the values coming in on input <1> get multiplied by the value on F:DYROTATE's input <3>.

Input <2> is an accumulator value. It performs the same function F:CMUL does in the other network. This input requires an initial or *reset value* (in this case 0) the same way F:CMUL needs an identity matrix at input <1>. (You can send a zero to input <2> and reset the accumulator whenever desired.) Output <2> is not used here but consists of the constant accumulator content on input <2>.

In short, F:DYROTATE does everything the F:YROTATE network does with one function instead of three. Since there is no reason to use three functions where one will do, the next task is to use PS 300 commands to implement a function network using F:DYROTATE (Figure 13).

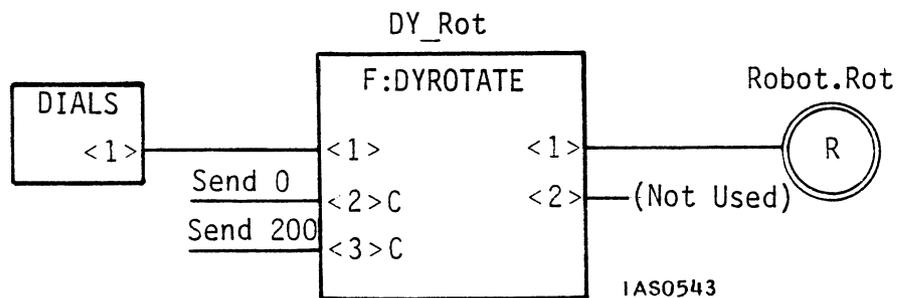


Figure 13. F:DYROTATE Function Network

This network consists of two functions and a named node. The first function, DIALS, is an Initial Function Instance. It has eight outputs, one for each dial. You do not have to instance it; the PS 300 does that automatically when you turn it on.

The second function in the network, F:DYROTATE, must be instanced and assigned a name.

This network is almost identical to the one used in the "Hands-On Experience" module. There you rotated the star-shaped object named Spinstar by connecting it to a F:DZROTATE function (Figure 14).

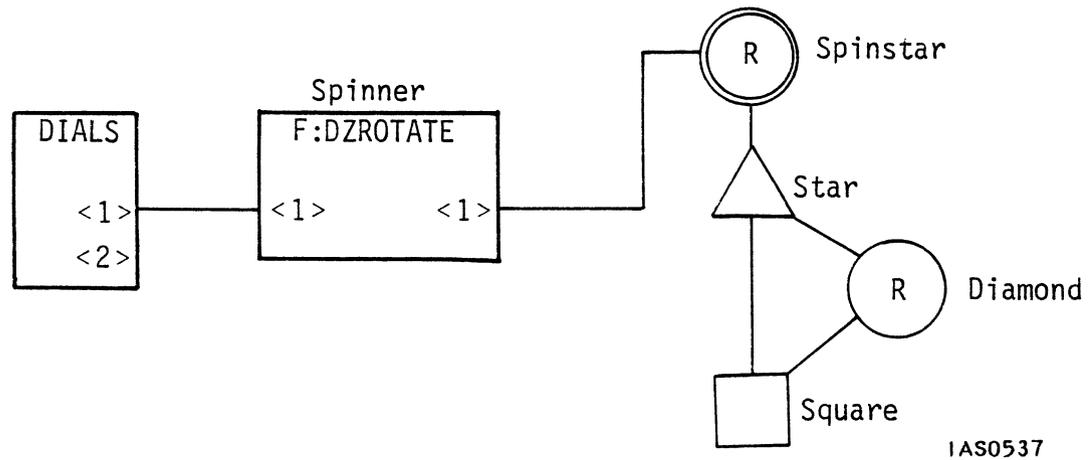


Figure 14. "Spinner" Function Diagram

To use that function, you had to instance it; i.e., assign it a unique name (Spinner). Then, using the CONNECT command, you connected DIALS<1> to Spinner's input and its output to the interaction node named Spinstar. Spinner was primed by sending initial values to its two constant inputs. Then it was ready to use. Turning the dial activated the network, causing the star to spin on the screen. Do the same thing in the following exercise.

Exercise

Define DY_Rot to be an instance of F:DYROTATE using this command

```
DY_Rot := F:DYROTATE;
```

Now connect outputs to inputs as shown in the diagram. Connect DIALS<1> (the top left dial) to DY_Rot's input and DY_Rot's output to Robot.Rot with the CONNECT command:

```
CONNECT DIALS<1>:<1>DY_Rot;
```

```
CONNECT DY_Rot<1>:<1>Robot.Rot;
```

In the CONNECT command, the input number of a function always precedes its name and the output number always follows it.

Send initial values where they are indicated. There are only two in this network: DY_Rot's two constant inputs.

SEND 0 to <2>DY_Rot;

SEND 100 to <3>DY_Rot;

Everything in the function network diagram you drew is now accounted for and implemented in the PS 300. Turn dial 1 and the model should rotate on the screen.

Then try to enlarge the network so dials 2 and 3 control the other two rotations in X and Z, using F:DXROTATE and F:DZROTATE in exactly the same way. When the network is coded, move the dials and watch what happens to the model on the screen.

Whenever you construct a function network, use the following good programming practices:

- Always design your network before you try to code it. If you work from a diagram, you will not forget to instance a needed function or to make a required connection.
- Instance all required functions, and check them off in the diagram as you instance them. If you try to connect a function that has not first been instanced, you will create an error.
- Next, make connections from left to right in the diagram, and check them off as you make them. Starting with the "most upstream" functions, make all connections until you reach the outputs of the network.
- Last, prime the network with initial values. Make sure that you send a number to a constant input whenever you need to.

Figure 15 is a diagram of a larger network that includes the other two rotations.

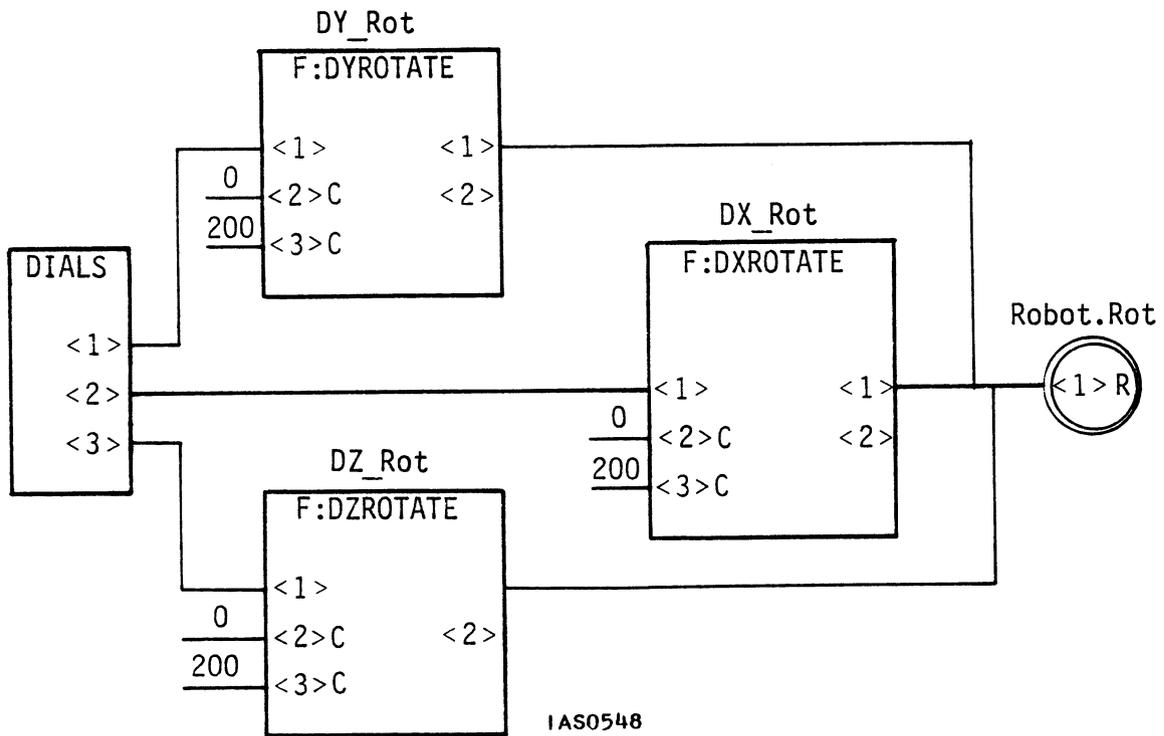


Figure 15. Completed Function Network for X, Y and Z Rotation

Probably the only differences between your network and this one will be the names used to instance functions. If you diagrammed your network and entered the commands correctly, you should see some unexpected jerking around in the model if you turn one dial after turning another. The next section explains why.

ADDING FURTHER INTERACTION: ROTATION IN OTHER DIMENSIONS

The first attempt to expand the number of rotations for Robot using F:DXROTATE and F:DZROTATE produced some jerkiness. The jerkiness occurs because each rotation function in this network has its own built-in accumulator (input <2>). If you rotate the robot in Y 90 degrees, you have an accumulated 90-degree rotation value in DYROTATE. Turning the X dial generates a matrix that specifies X rotations from the initial position in X, Y, and Z. In other words, the matrix that DXROTATE produces overrides the accumulations already in DYROTATE. The X rotation applies as if no other rotation has occurred. So the model appears to jump back to its initial position before it starts rotating in X.

It was not wrong to pick F:DYROTATE instead of the F:YROTATE network if you only want to rotate a model around one axis. In that case, a DROTATE function is simpler to use. But to add rotations in other dimensions, you need to account for all the rotations. You could add two more rotate nodes to the display tree for X and Z rotations as shown in Figure 16.

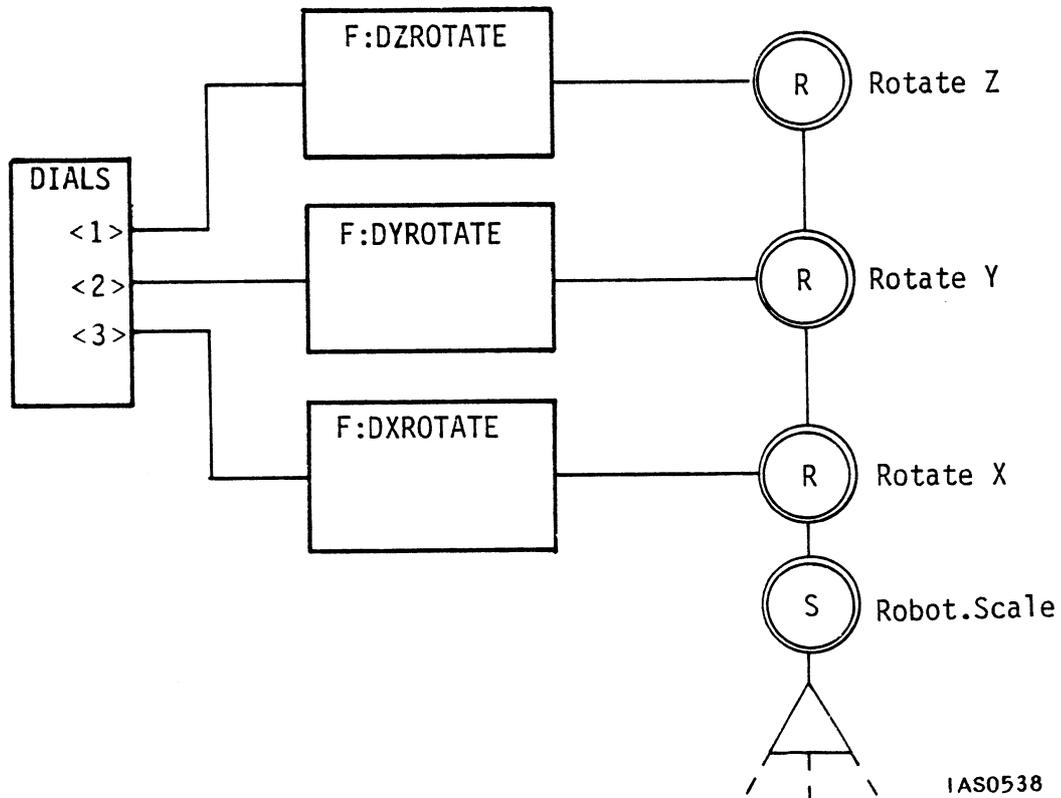


Figure 16. Modified Display Tree With Three Rotate Nodes

Another method is to provide a common accumulator for the whole group. Look at the network with F:YROTATE (Figure 17).

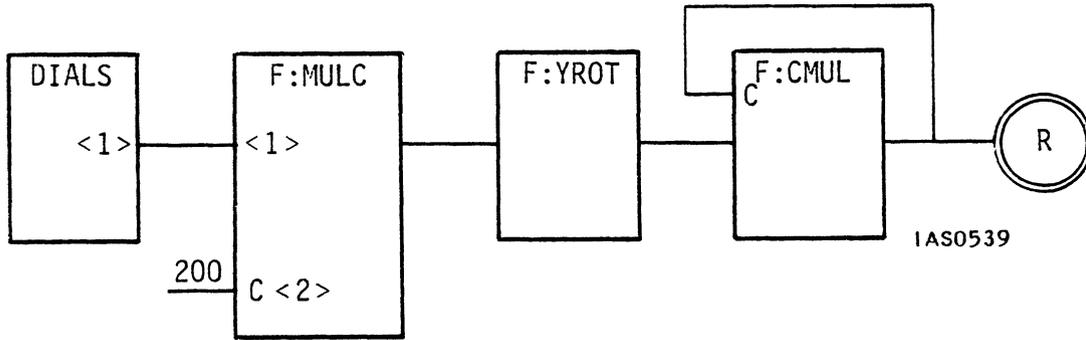


Figure 17. F:YROTATE Network

This network has a separate accumulator, an instance of F:CMUL. It can serve as the sole accumulator for all of Robot.Rot's rotations, if all three ROTATE functions are connected to it.

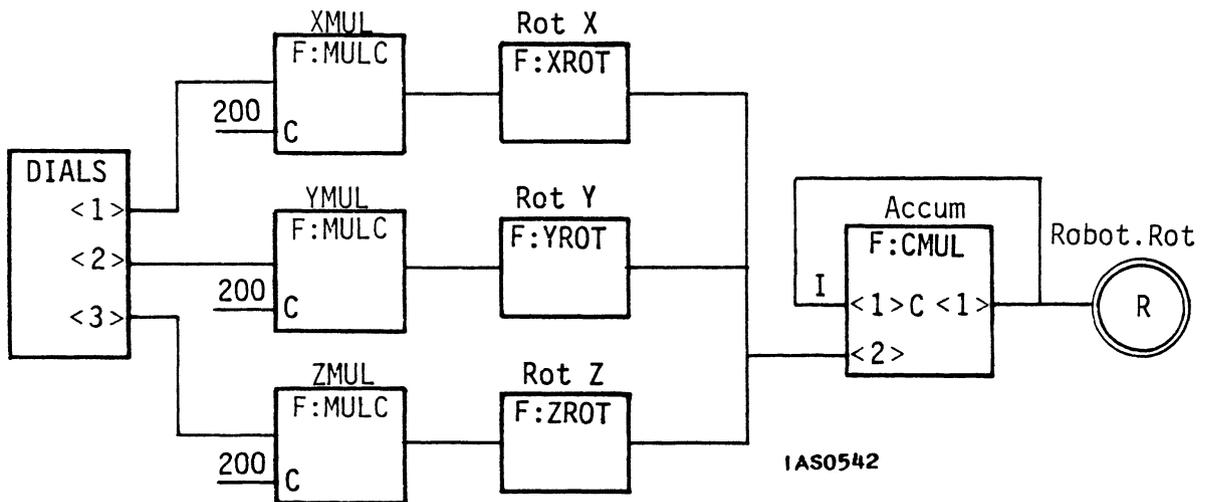


Figure 18. Common Accumulator for Rotate Functions

This example (Figure 18) shows how a single input of a function (<2>F:CMUL) can receive values from more than one source (F:XROTATE<1>, F:YROTATE<1>, F:ZROTATE<1>).

Exercise

Use the DISCONNECT command to break the connections for the network you have programmed into the PS 300. Enter:

```
DISCONNECT DIALS<1>:<1>DY_Rot;  
DISCONNECT DIALS<2>:<1>DX_Rot;  
DISCONNECT DIALS<3>:<1>DZ_Rot;
```

Now program the network shown in Figure 18. Then turn all three dials and pay close attention to how the model moves in Y after you have moved it in Z.

First, the functions must be instanced:

```
Xmul := F:MULC;  
Ymul := F:MULC;  
Zmul := F:MULC;  
Rotx := F:XROT;  
Roty := F:YROT;  
Rotz := F:ZROT;  
Accum := F:CMUL;
```

Second, connections must be made between the functions:

```
CONNECT Dials<1>:<1>Xmul;  
CONNECT Dials<2>:<1>Ymul;  
CONNECT Dials<3>:<1>Zmul;  
  
CONNECT Xmul<1>:<1>Rotx;  
CONNECT Ymul<1>:<1>Roty;  
CONNECT Zmul<1>:<1>Rotz;  
  
CONNECT Rotx<1>:<2>Accum;  
CONNECT Roty<1>:<2>Accum;  
CONNECT Rotz<1>:<2>Accum;  
  
CONNECT Accum<1>:<1>Accum;  
CONNECT Accum<1>:<1>Robot.Rot;
```

Finally, the functions must be primed by sending initial values to their constant inputs. This includes sending an identity matrix to initialize input <1> of the accumulator.

```
SEND 200 to <2>Xmul;  
SEND 200 to <2>Ymul;  
SEND 200 to <2>Zmul;  
SEND M3D(1,0,0 0,1,0 0,0,1) to <1>Accum;
```

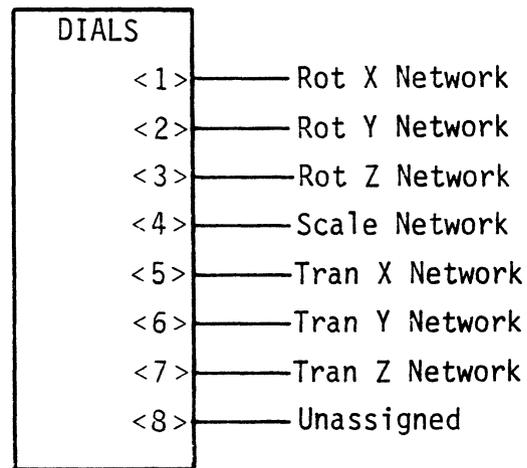
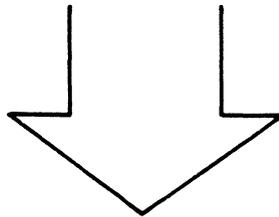
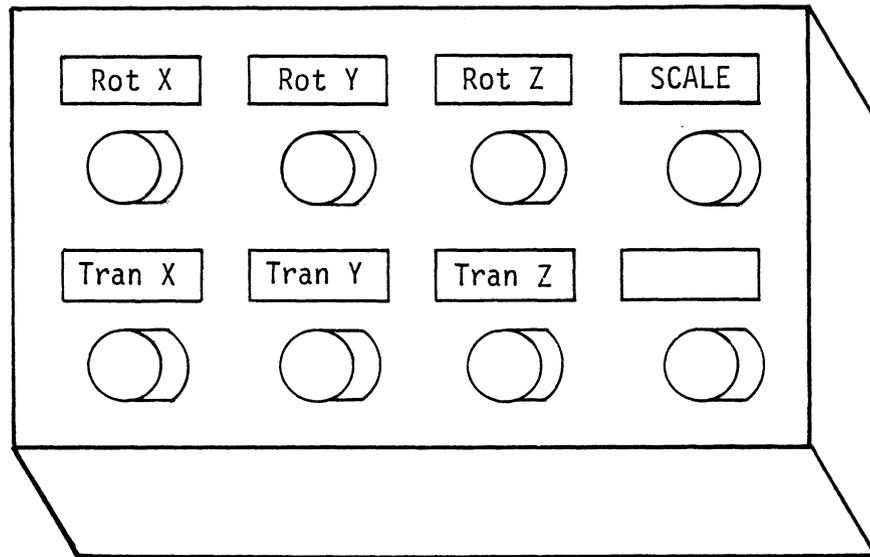
These rotations are called *world-space rotations*; they take place around the world's axes and not the model's axes. Once you rotate Robot in Z, if you rotate him in Y he will spin around an axis running through him that is parallel to the Y axis of the coordinate system. When a model rotates around its own axes, that is called *object-space rotation*. For a further discussion of object-space and world-space rotations, refer to the *PS 300 Application Notes* in Volume 4.

EXPAND THE NETWORK FOR OTHER KINDS OF INTERACTION

(Scaling and Translating)

Two other transformations are possible for the robot: scaling and translations. You have already laid the groundwork by including a scaling node and a translation node at the top of the robot display tree.

Dials 1, 2 and 3 now control the model's rotations. Determine which of the remaining dials will control scaling and translation. Figure 19 shows a suggested configuration for using the dials to control rotations, translation, and scaling for an object. One dial controls scaling, and three each are assigned for rotation and translation. One dial is unassigned.



IAS0540

Figure 19. Suggested Configuration for Dials

Translations take place in X, Y, and Z and need three dials: 5, 6, and 7. The type of scaling here is *uniform scaling*, so one value will scale in all dimensions equally. Only one dial needs to be used: dial 4.

Often, all the rotations you will need to make a model fully interactive will require more than eight dials. In the proposed network you have so far, for instance, only three nodes from the model's display tree use up seven dials. The remaining interaction nodes in the display tree require up to three dials each. This means about fifty dials are necessary to handle all those rotations. A way to reuse a single set of eight dials to solve this problem is discussed in "Function Networks II".

Now enlarge the network to translate the robot. This network will closely resemble the one just finished for rotations. First, determine what data type the node requires.

Translate nodes accept vectors. The TRANSLATE command's associated functions in the *Command Summary* are F:XVECTOR, F:YVECTOR, and F:ZVECTOR. These all take real numbers as their input and produce 3D vectors. The input value is in the X (or Y or Z) position in the vector. F:XVECTOR, for example, would take the real number 4.5 and send out the vector (4.5, 0, 0). F:YVECTOR would take the same input and send out (0, 4.5, 0).

Figure 20 shows a network with VECTOR functions for translating in three dimensions.

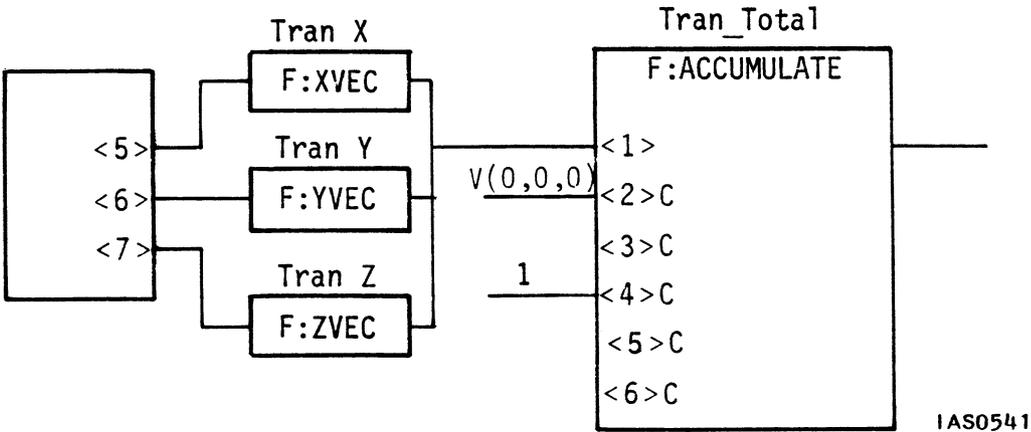


Figure 20. Translate Network

The accumulator in this network is not the multiplying function F:CMUL but a new one, F:ACCUMULATE. F:ACCUMULATE does the job of several functions. There is no need, for example, to put a magnifying function like F:MULC in this network. To enlarge dial values, send a magnifying factor to F:ACCUMULATE's input <4>. In the case of this translate network, a suggested factor is 10 (that corresponds to the 200 magnifying factor for rotations). The reset value for the accumulator goes on input <2>.

Three inputs are not used in this application. Input <3> lets you control the smoothness of translation by setting the minimum change in position per output. And the last two inputs control limits. If you do not want an object to move more than a specified amount (to keep it within the limits of the screen, for example), you can set limits on its movement with inputs <5> and <6>.

The accumulator is shared by all three VECTOR functions just as the three ROTATE functions share a common accumulator.

Exercise

Instance the functions in the translate network using the names suggested in Figure 20, connect them to dials 6, and 7, and prime constant queues <2> and <4> of F:ACCUMULATE. Then use what you know about building networks to diagram one for scaling the robot.

To create the translate network, first instance the functions:

```
Tranx := F:XVEC;  
Trany := F:YVEC;  
Tranz := F:ZVEC;  
Tran_total := F:ACCUMULATE;
```

Then connect the functions to the dials:

```
CONNECT Dials<5>:<1>Tranx;  
CONNECT Dials<6>:<1>Trany;  
CONNECT Dials<7>:<1>Tranz;  
  
CONNECT Tranx<1>:<1>Tran_Total;  
CONNECT Trany<1>:<1>Tran_Total;  
CONNECT Tranz<1>:<1>Tran_Total;  
CONNECT Tran_Total<1>:<1>Robot.Tran;
```

Prime the functions by sending initial values to any constant inputs.

```
SEND V(0,0,0) TO <2>Tran_Total;
SEND 1 TO <4>Tran_Total; {Note that 1 is the default for input <4>}
```

To construct a network for scaling, first use the *Command Summary* to determine what values the model's scale node uses--3x3 matrices--and what its associated functions are. You will find the same functions that are listed under the ROTATE command, but the applicable ones here are F:SCALE and F:DSCALE. To use F:SCALE, you need a separate magnifying function and a separate accumulator.

F:DSCALE is more of a "3-in-1" function like F:ACCUMULATE and the DROTATE functions. It combines a matrix-producing function, an accumulator, and a magnifier all in one. Since you only have one scaling factor, F:DSCALE will be safe to use here (you do not have to worry about separate X, Y, and Z scaling factors).

The values from the dial come in on input <1>. Input <3> is the magnifying factor for dial values. Rather than using 100, as you did in the rotation network, use 0.1. This smaller value is used because robot is initially scaled by .075. (See Figure 21.)

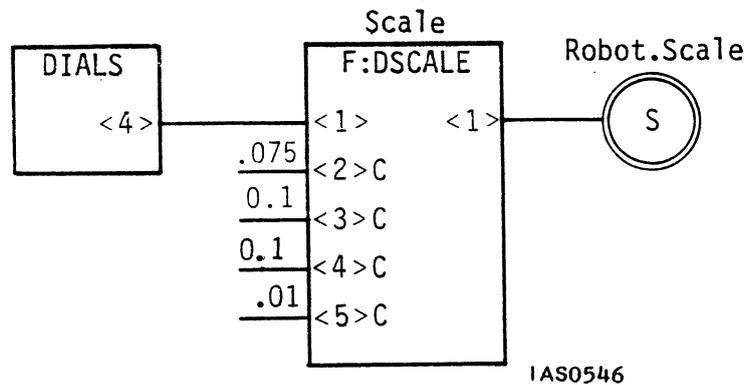


Figure 21. Network for Uniform Scaling

F:DSCALE requires an accumulator reset value for input <2>. This should correspond to the initial value the object is scaled to. In most cases, that is 1, but remember the robot is already scaled to .075 so he will be small enough to appear on the screen. Be sure to send this initial scale value (.075) to input <2>.

F:DSCALE, like F:ACCUMULATE in the translation network, also lets you set upper and lower limits so the object being scaled does not become too large or small. If you sent 0.1 to input <4> and .01 to input <5>, for example, the robot would never become more than twice or less than one-fifth his initial size (.075) on the screen. If you do not send limits to these two inputs, no limit is set.

Exercise

Tracing two or three of the fractional values from the dial shows that F:DSCALE accumulates scaling values as you expect. Now, use the names shown in Figure 21 to instance, connect, and prime the functions.

To create the scale network, first instance the function:

```
Scale := F:DSCALE;
```

Then connect the function to the dial and the interactive scale node in the robot display tree:

```
CONNECT Dials<4>:<1>Scale;  
CONNECT Scale<1>:<1>Robot.Scale;
```

Finally, prime the function by sending initial values to the constant inputs.

```
SEND .075 TO <2>Scale;  
SEND 0.1 TO <3>Scale;  
SEND 0.1 TO <4>Scale;  
SEND .01 TO <5>Scale;
```

A CLOCK FUNCTION AS AN ALTERNATE SOURCE OF INPUT FOR THE NETWORK

It is not always necessary to use dials, function keys, or data tablets to provide input for a function network. You may want some action to happen automatically or to cycle through and repeat. This section discusses how to use a clock function to do that for Y rotations. When the network is connected to the robot, it will automatically rotate.

Whenever you SEND an integer to a function, use FIX(number). FIX indicates the value is an integer and not a real number. If you do not use FIX, the function will still operate, but it requires more computation time.

Use F:CLFRAMES, shown in Figure 22.

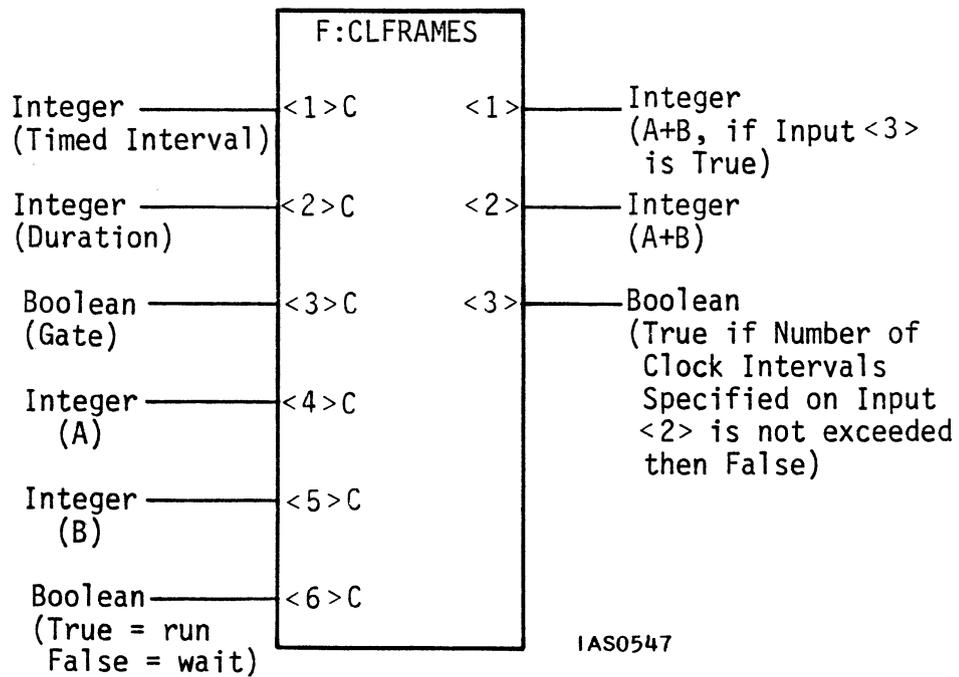


Figure 22. F:CLFRAMES

The "CL" in the name indicates it is a clock function; "FRAMES" means that its "ticking" depends on the rate at which refresh frames are displayed. The standard refresh rate for the PS 300 is about 60 frames each second. The refresh rate can vary, however, so it is not linked to time. So F:CLFRAMES sends out a value only when so many frames have been displayed, independent of time. The integer you place on input <1> specifies how many frames you want to elapse before F:CLFRAMES "ticks".

In all, there are six inputs and three outputs for this function. These allow you to use F:CLFRAMES as more than just a simple counter, which is all it will be used for in this example.

Input <1> is the interval measured in frames. It requires an integer, so send it a fix(2). This will result in about 30 degrees rotation per second.

Input <2> affects output <3> but has no effect on what you are doing right now. It requires an integer, so send it a fix(1).

Input <3> shuts down or opens output <1>. Since you will not use output <1> here, send a FALSE to input <3>.

Inputs <4> and <5> are constant, and contain integer values whose sum is generated when F:CLFRAMES ticks. You can accumulate the sum by connecting output <2> back around to input <5> and then sending fix(1) to input <4> and 0 to input <5>. However, since F:CLFRAMES is to be used in a network that is already set up to accumulate values from the dials (ACCUM), values should not be accumulated in F:CLFRAMES. F:CLFRAMES output <2> is connected to the rotation network you already have, as shown in Figure 23.

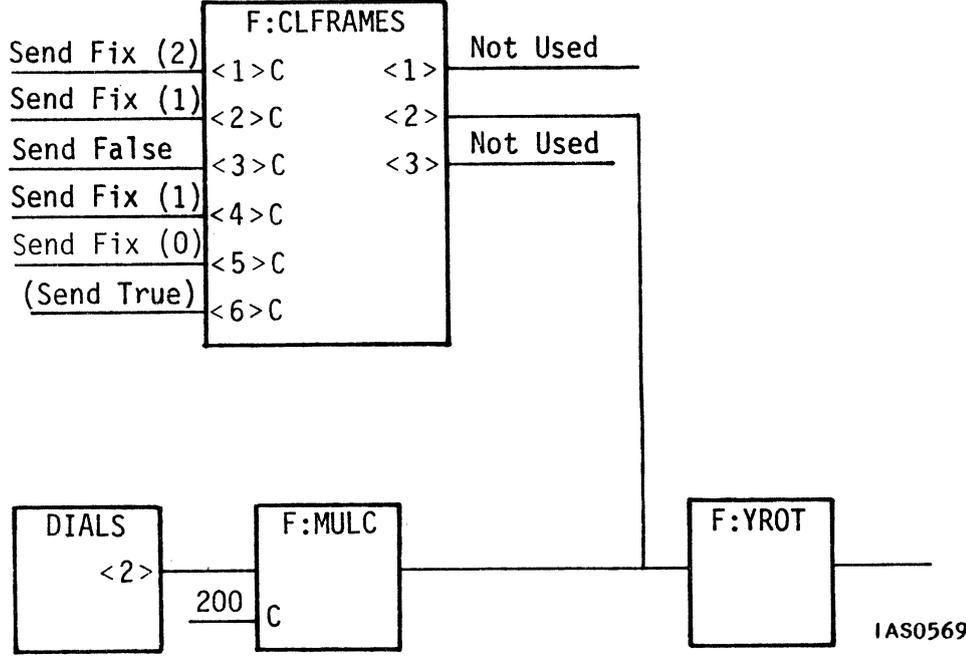


Figure 23. F:CLFRAMES as Input Source for Y Rotation

The diagram shows initial values for inputs <2> and <3> of F:CLFRAMES. Though they are not used here, they must be supplied for F:CLFRAMES to function. In the same way that F:ADD will not fire until it has two inputs, F:CLFRAMES requires that some value must be placed on all its inputs in order to run. The diagram reminds you of all the values you need to send to prime the network.

Input <6> provides a switch to operate the clock. It requires a Boolean value value, TRUE to run the clock or FALSE to stop it.

Exercise

Instance F:CLFRAMES as "Timer" and connect the function into the network as shown in Figure 23. Be sure to send initial values to all of the first five inputs. Last, send a TRUE to input <6> to make the model begin spinning. Here is a list of the commands needed to implement the network shown in Figure 23.

```
Timer := F:CLFRAMES;

CONNECT Timer<2>:<1>Roty;

SEND FIX(2) TO <1>Roty;
SEND FIX(1) TO <2>Timer;
SEND FALSE TO <3>Timer;
SEND FIX(1) TO <4>Timer;
SEND FIX(0) TO <5>Timer;
SEND TRUE TO <6>Timer;
```

To stop the robot from twirling, send a FALSE to input <6>.

Remember that the dials are still connected to the robot. You now have TWO sources of input for a function (Roty receives from DIALS through the instance of F:MULC and from the CLOCK function). To be sure the two sources of input do not compete, you can shut F:CLFRAMES off when you use the dials by sending FALSE to input <6>.

You can also "unplug" it entirely by using the DISCONNECT command. Use this command exactly as you do CONNECT:

```
DISCONNECT Timer<2>:<1>Roty;
```

Of course, to ensure that the dial does not interfere with the clock, you could break the connections between the dial and the instance of F:MULC that leads to Roty.

When you do turn the clock off by sending FALSE to input <6>, it would be convenient to do so by simply pushing a button instead of typing in SEND commands repetitively.

Since function networks are so flexible, there are dozens of ways to accomplish something, as you have already seen with the two or three ways to control rotations in a model. Designing a switch is just as open-ended. You could arrange to have F:CLFRAMES start firing when you push function key 1 and stop when you push function key 2, for instance. Or it could start if you sent it any value larger than five, and stop if it received a value less than five. The network shown in Figure 24, however, toggles. If you press any function key, it turns F:CLFRAMES on; if you press it again, F:CLFRAMES turns off.

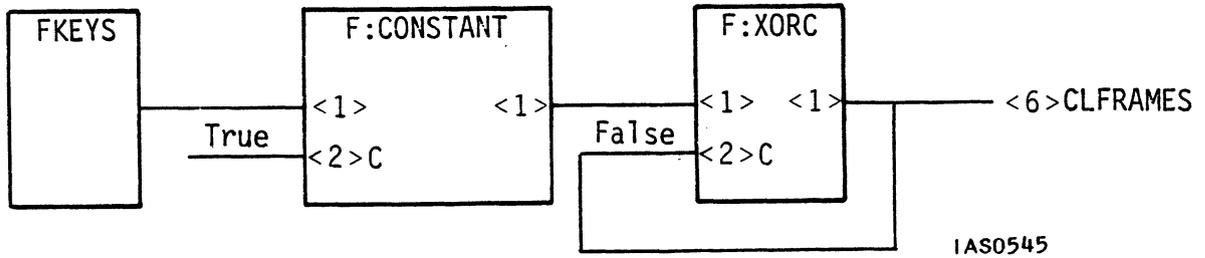


Figure 24. A Network That Toggles

In this network, F:CONSTANT can hold any value--a number, a matrix, a Boolean value value--on its constant input <2> and will send out that value when any value arrives on input <1>.

Place a TRUE on input <2> and connect the output of F:CONSTANT to another function, F:XORC. This function performs a logical operation, exclusive OR-ing. It compares the Boolean value it receives on input <1> to the Boolean value on its constant input <2> and produces a TRUE if they are different and a FALSE if they are the same. F:XORC's output is then sent back to its own constant input and also on to the place you need to toggle (input <6> of F:CLFRAMES).

Trace a couple of values from the function keys function FKEYS through this network to confirm that you get alternating TRUE and FALSE values as output.

To emphasize that there are many ways to do something with function networks, Figure 25 shows a more efficient network for a switch.

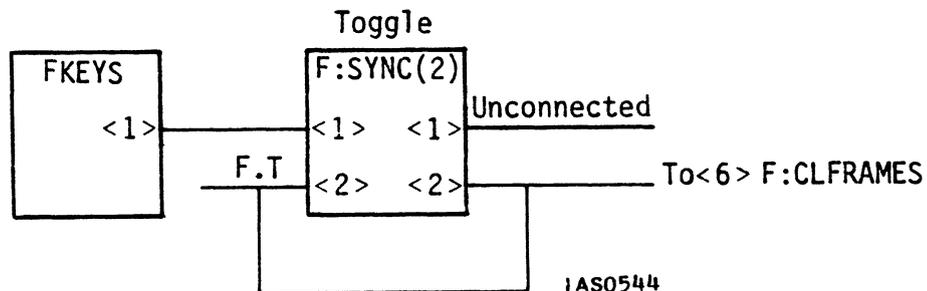


Figure 25. A More Efficient Toggle Switch

Here the network is composed of only one function. The stackable nature of active inputs is used to queue a FALSE and a TRUE. Then F:SYNC's output is connected back to its own input 2 so the two Boolean values can alternate as output values.

SUMMARY

If you added to your function network throughout this module, the final network diagram should look like the one in Figure 26.

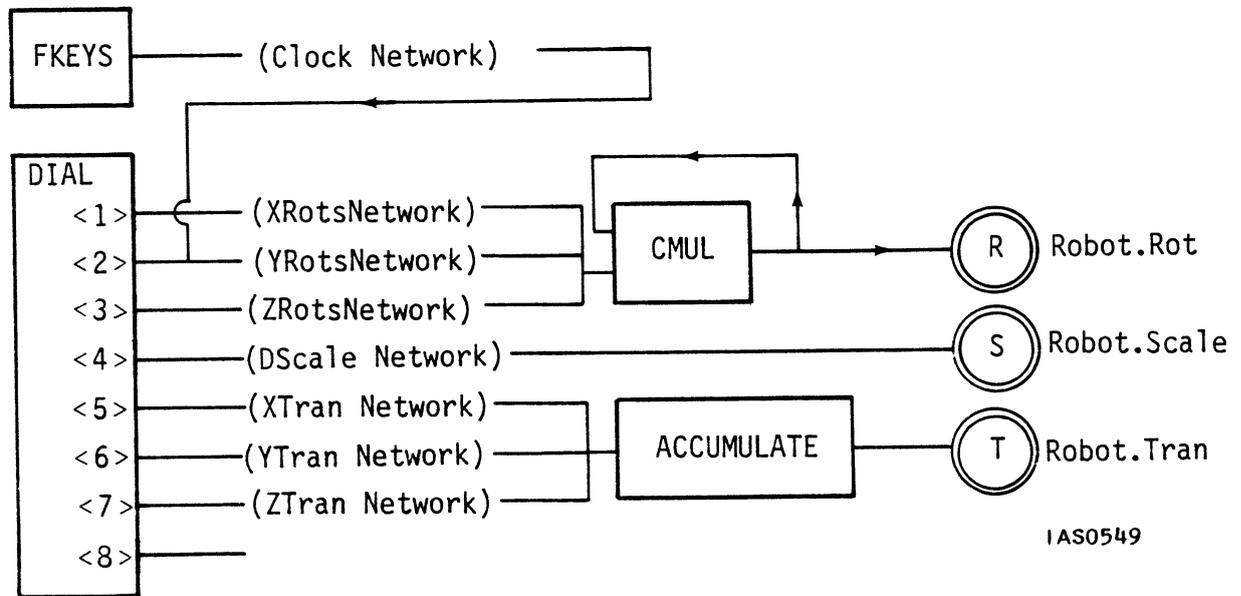


Figure 26. The Completed Network

When the diagram contains all function instance names and initial values to be sent, it looks complicated, but its operations are fairly simple. It controls interactions for only three display tree nodes.

Review of Major Points

To build a function network, you must find candidate functions or function networks (represented as a black box) which convert input device values into values that can update interaction nodes in a display tree. To do this:

- Identify the type of output needed by interaction nodes

- Identify sources of input and what type of values they generate
- Use the *Command Summary* to find related functions for interaction nodes
- Use the *Function Summary* to evaluate candidate functions or networks and modify network as needed with additional functions
- Implement the network using good programming practices:

Always diagram the network first
Instance functions first
Make connections from left to right in the diagram
SEND any initial values to prime the network.

Once the basic network is built, you can expand it. In this module's network, you added:

- Rotations in other dimensions. Some way of accumulating rotations is usually needed.
- Other kinds of interaction--scaling and translating.
- An alternate source of input for the network--a CLOCK function. This can be toggled on and off with a switch network connected to a function key.

Important Facts About PS 300 Functions

- When a complete set of input values arrives, the function executes and sends out values on its outputs.
- Functions can have constant or active 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.

Constant inputs hold only one value at a time--there is no queuing. A value on a constant queue is not consumed when the function fires. It will remain until it is overwritten by another value.

- Functions perform arithmetic, logical, routing, or data conversion operations.
- In a function network, values flow from left (upstream) to right (downstream).
- Functions that are directly associated with an input device, such as DIALS and FKEYS, do not need to be instanced. These are examples of initial function instances; they are instanced by the system.

PS 300 Commands Discussed in This Module

- Immediate action commands: CONNECT, DISCONNECT, SEND.
- Function-instancing commands: name := F:function name.



VIEWING OPERATIONS

LOOKING AT THE MODEL

CONTENTS

INTRODUCTION	1
OBJECTIVES	2
PREREQUISITES	2
DEFINING A LINE OF SIGHT	3
Looking Straight Up or Straight Down	7
Exercise	10
Using a 4x4 Matrix to Specify a Line of Sight	10
DEFINING AN ORTHOGRAPHIC WINDOW	11
Altering the Size of a Window	13
Exercise	14
Moving the Window	15
Exercise	17
Specifying Window Depth: Depth Clipping	19
Optimizing Depth Cueing	20
Using a 4x4 Matrix to Specify an Orthographic Window	22

VIEWING

DEFINING PERSPECTIVE WINDOWS	23
Using FIELD_OF_VIEW	25
Exercise	27
Exercise	28
Using the EYE Command	29
Exercise	33
Exercise	34
Using a 4x4 Matrix to Specify a Perspective Window	37
Exercises	39
SPECIFYING A VIEWPORT	38
Displaying Multiple Viewports	42
Exercise	42
Using Non-Square Viewports	43
Exercise	44
Exercise	45
Setting an Intensity Range for a Window in the Viewport	46
VIEWING ATTRIBUTES	48
Setting Intensity	48
Setting Displays ON and OFF	50
Exercise	51
Setting Color	51
Exercise	53
VIEWING SUMMARY	54

VIEWING

ILLUSTRATIONS

Figure 1.	LOOK Node	1
Figure 2.	Default LOOK	4
Figure 3.	Car	5
Figure 4.	Car From Left Side	6
Figure 5.	LOOK Transformation Sequence	7
Figure 6.	Line of Sight Colinear with UP Direction	8
Figure 7.	LOOKing Down	9
Figure 8.	WINDOW Node	11
Figure 9.	Default WINDOW	12
Figure 10.	Clipped View of Car	13
Figure 11.	Another View of Car	14
Figure 12.	Display Tree for Large Window	15
Figure 13.	Relocated Window	16
Figure 14.	Interrelation of LOOK and WINDOW Transformations	18
Figure 15.	Set Depth Clipping Display Tree	19
Figure 16.	Intensity as a Function of Z Location	21
Figure 17.	Orthographic Window Compared to Perspective Window	23
Figure 18.	Angles Between Opposing Sides of the Pyramid	24
Figure 19.	Display Tree with Field-Of-View Node	25
Figure 20.	Setting Z Boundaries for Maximum Depth Cueing	26
Figure 21.	Using FIELD_OF_VIEW with LOOK	27
Figure 22.	Setting Front and Back Boundaries	28
Figure 23.	Relative Room Coordinates	30
Figure 24.	Line of Sight for LOOK and EYE	31
Figure 25.	Specifying the Viewing Angle	32
Figure 26.	Moving Eyepoint Back and Left	32
Figure 27.	Boundaries Using the EYE Command	33
Figure 28.	EYE View of Cars	35
Figure 29.	EYE View of Car2	36
Figure 30.	Current Viewport Dimensions	38
Figure 31.	Port2 - Upper Right Quadrant	40
Figure 32.	Display Tree for Port2	40
Figure 33.	Port 3 and Associated Display Tree	41
Figure 34.	Dimensions of a Non-Square Current Viewport	43
Figure 35.	Square Window Mapped to Non-Square Viewport	45
Figure 36.	Non-Square Window Mapped to Non-Square Viewport	46
Figure 37.	Color Wheel	52



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 300 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 300 Display in which the window is displayed. Viewports can be full-screen or a smaller portion of the screen. The PS 300 lets you display multiple viewports simultaneously, so it is possible to have different views of the same model or view different models simultaneously.

The last set of viewing operations you can specify is called *viewing attributes*. These allow you to set an intensity range for displayed data, set any display on and off interactively, and set color for displayed objects (for viewing on a calligraphic color display).

When you turn on the PS 300, 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 viewport (which is full screen).

All of the PS 300 viewing operations--viewing transformations, viewports, and viewing attributes--are represented in a model's display tree structure by operation nodes.

Windowing transformations are 4x4 matrix operations that override the current transformation matrix. Because of this, a windowing transformation should be the topmost operation in a display tree branch. If it is not, any operations above it in the branch will have no effect.

OBJECTIVES

In this module 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 viewport.
- Set an intensity range.
- Set displays on and off.
- Set color.

PREREQUISITES

Before reading this module, you need to know basic graphics concepts, how data structuring is done in the PS 300, and how modeling transformations work on data. (Refer to the "Graphics Principles," "Modeling," and "PS 300 Command Language" modules.)

This module makes use of tutorial demonstrations. (Refer to "Tutorial Demonstration Package.")

To do the exercises in this module, put the PS 300 in command mode (<Control> LINE LOCAL).

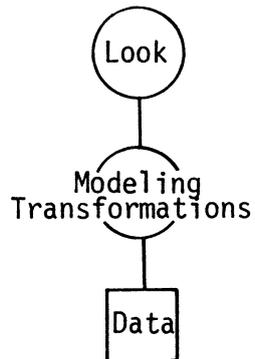
DEFINING A LINE OF SIGHT

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 300 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 tree. For a LOOK transformation to work correctly, it should be placed above all modeling transformations (ROTATE, TRANSLATE, SCALE) in the tree (Figure 1).



IAS0419

Figure 1. LOOK Node

4 - VIEWING

Note that the operate 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 2).

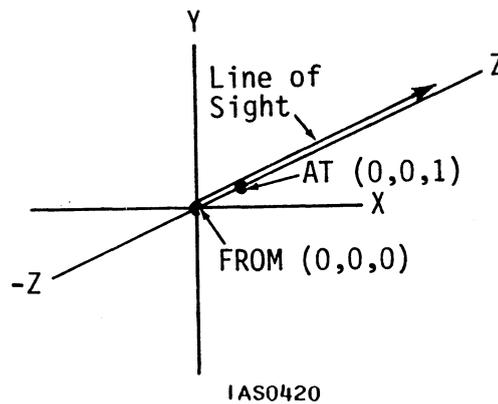


Figure 2. Default LOOK

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

Enter:

```
@@DISPLAY Car;
```

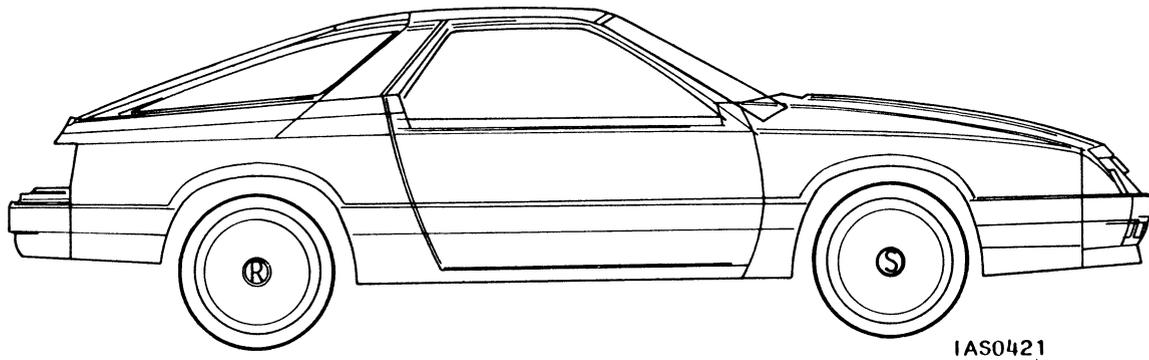


Figure 3. Car

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 4.

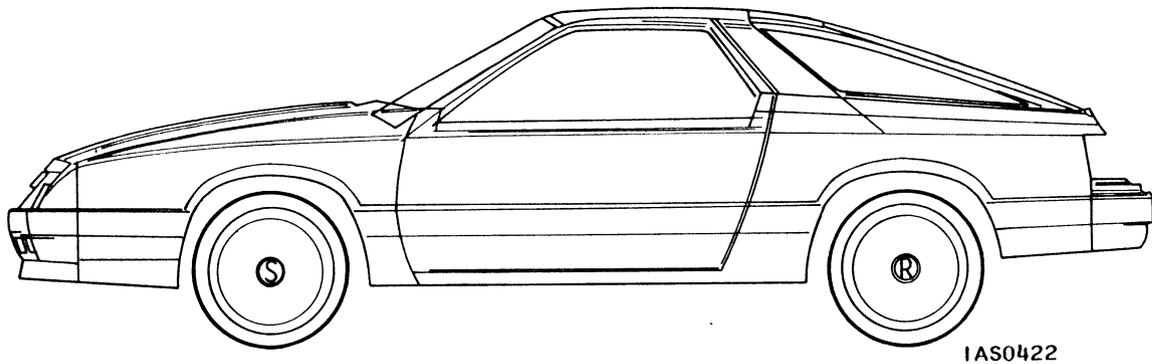


Figure 4. Car From Left Side

To create Left_View, the PS 300 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 5. (Note that the translation shown in Figure 5 is exaggerated for clarity.)

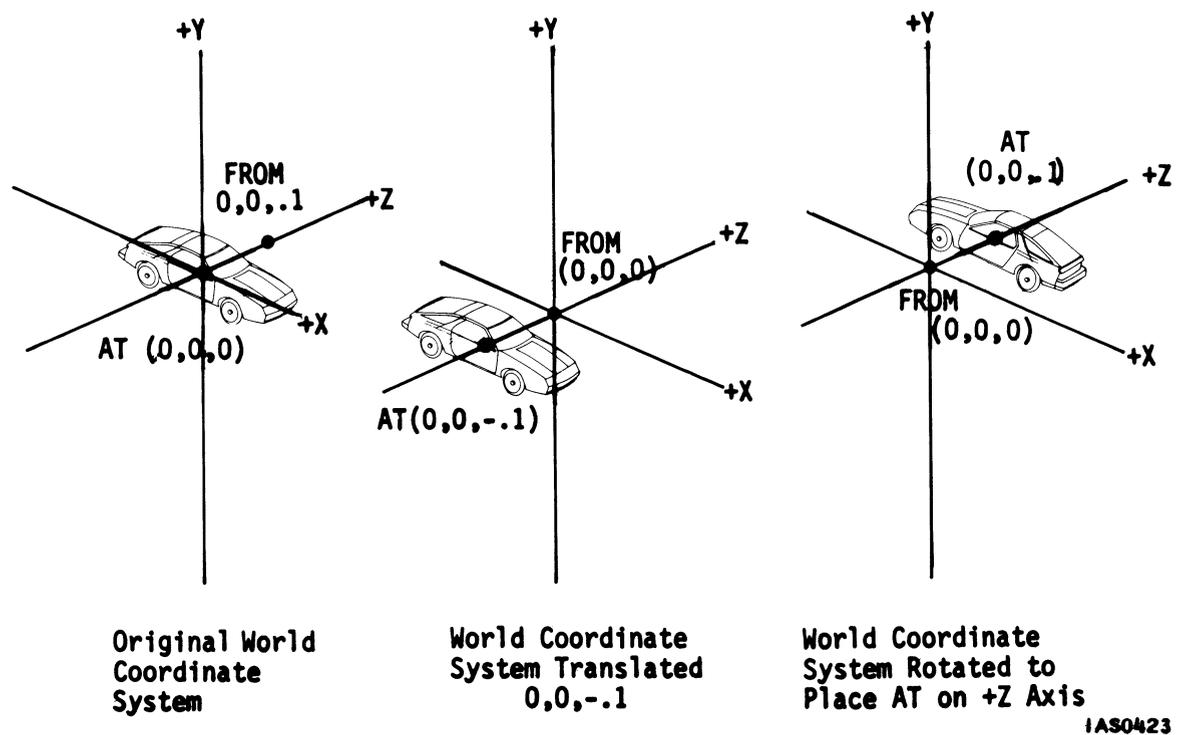


Figure 5. LOOK Transformation Sequence

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 colinear with the line of sight (Figure 6).

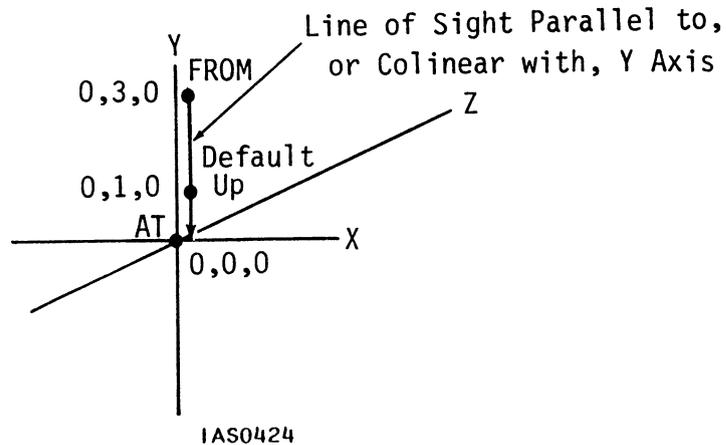


Figure 6. Line of Sight Colinear With UP Direction

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 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 7. (Note that in Figure 7 the distance from the FROM point to the AT point is exaggerated for clarity.)

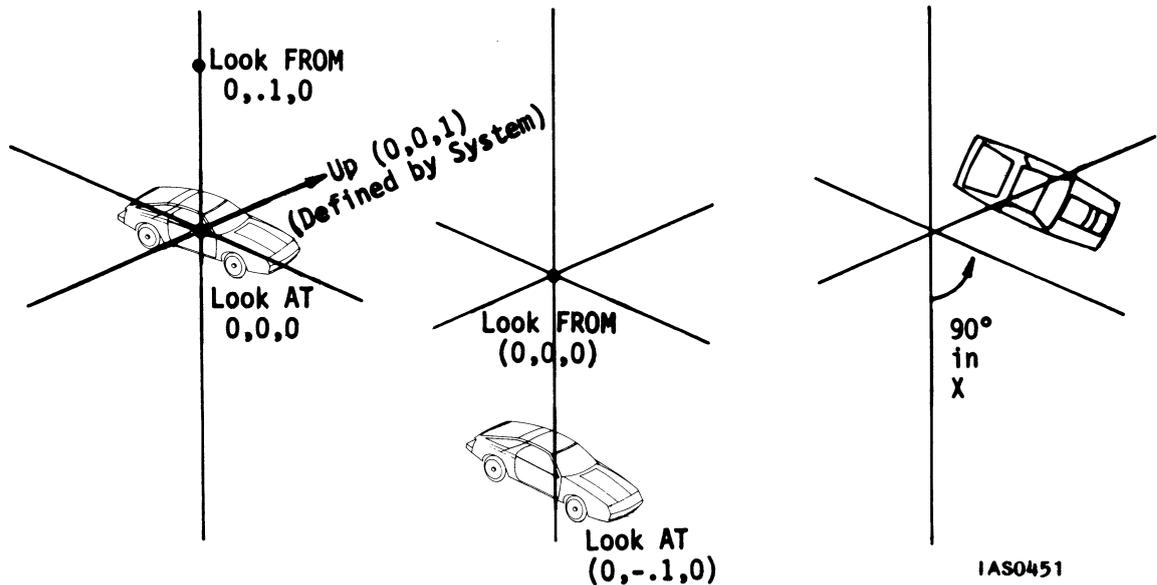


Figure 7. LOOKing Down

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;
```

Exercise

Refer to "Tutorial Demonstration Package" and run the LOOK demonstration program.

Using a 4x4 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;
```

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) parallel to the line of sight, with its front face in the X/Y plane. Once a window transformation is applied, everything 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 tree. In the PS 300, 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 tree. Figure 8 illustrates this rule.

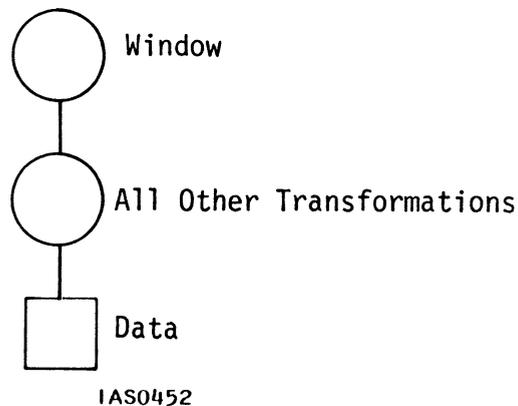


Figure 8. WINDOW Node

Just as there is a default LOOK imposed by the PS 300, 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 9) will appear on the screen. Objects outside the window in Z will be displayed unless depth clipping is enabled. Refer to Specifying Window Depth: Depth Clipping, a following section in this module.

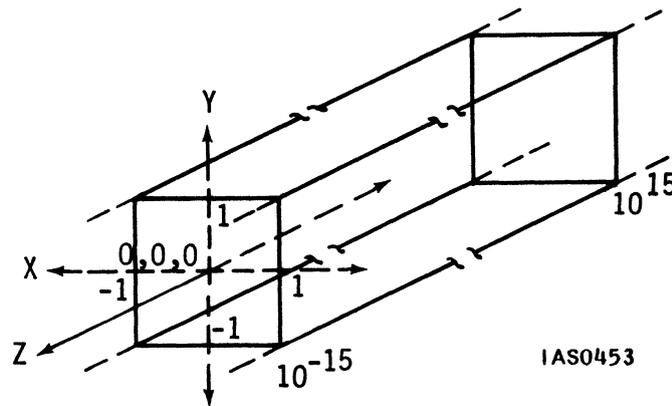


Figure 9. Default WINDOW

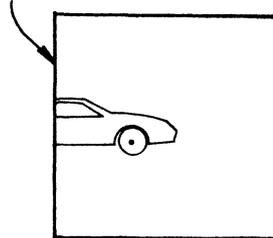
To see an object, it must be located within the X and Y boundaries of the viewing window. Any object outside these boundaries is removed from view via *clipping*.

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 10).

X and Y Window Boundaries



IAS0454

Figure 10. Clipped View of Car

Exercise

Define `Another_View` of `Car` as shown in the previous example and display `Another_View` to see the effect.

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.

Exercise

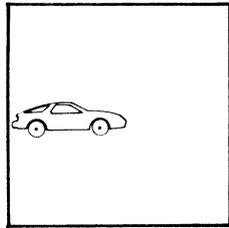
Enlarge the window and apply the new window specification to the LOOK called Another_View (Figure 11).

Enter:

```
@@Large_window := WINDOW X=-2:2 Y=-2:2
                    APPLIED TO Another_View;

@@DISPLAY Large_Window;

@@REMOVE Another_View;
```



IAS0455

Figure 11. Car in Large Window

All of the car appears in Large_Window. 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 tree created by the above sequence of commands is shown in Figure 12.

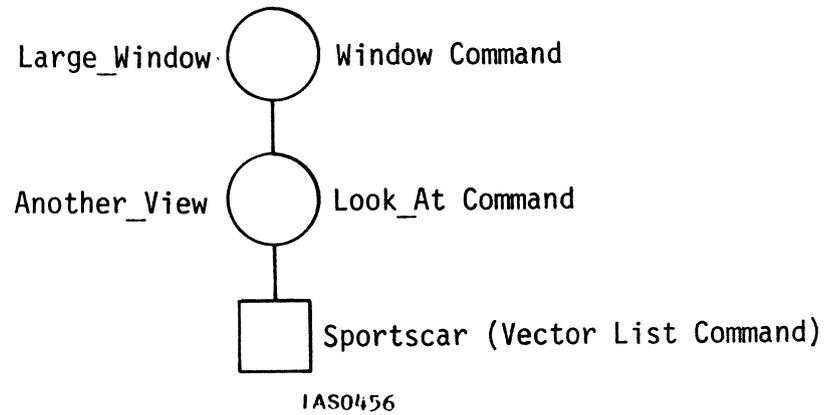


Figure 12. Display Tree for Large Window

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.

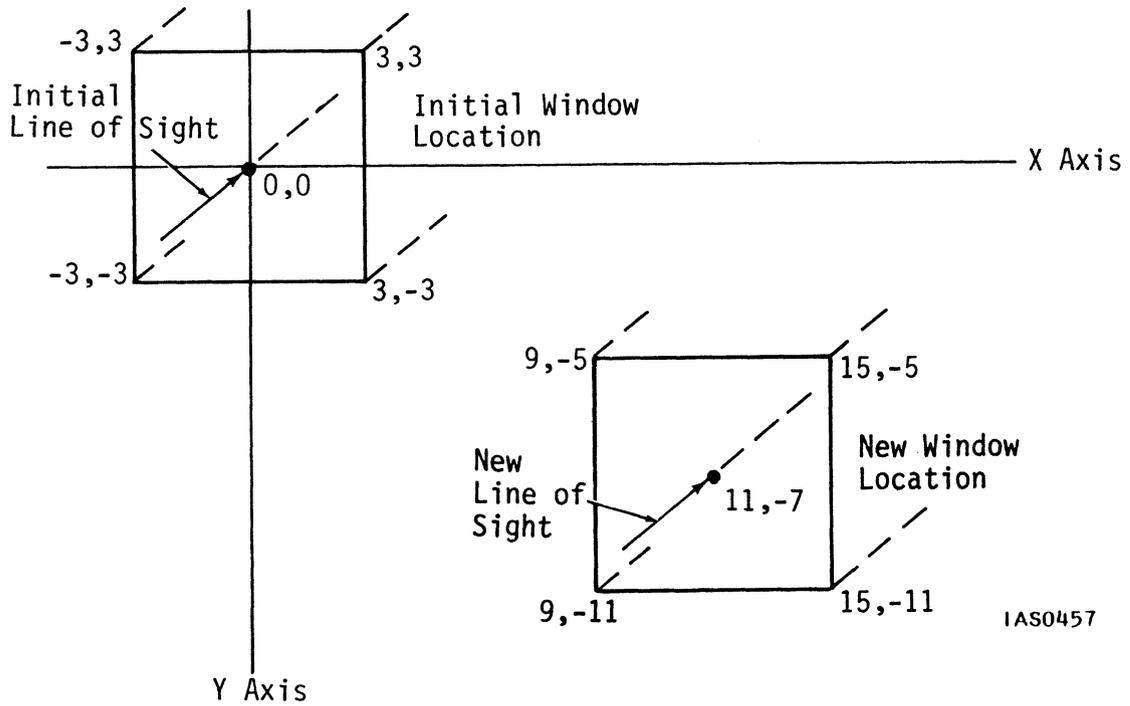


Figure 13. Relocated Window

If an orthographic window is defined as in the above diagram so that its center is not coincident with the Z axis, the PS 300 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.

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;

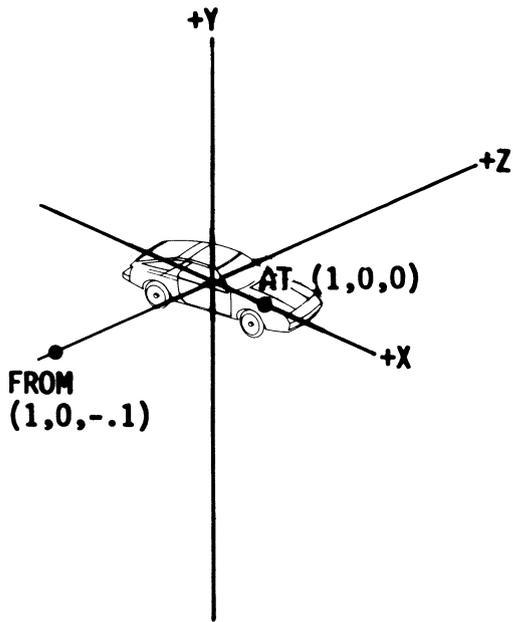
@@DISPLAY Move_Window;

@@REMOVE Another_View;
```

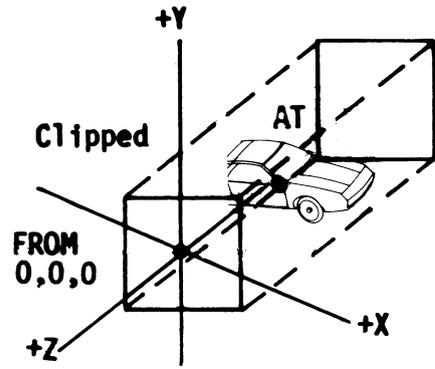
`Move_Window` clips no part of the car.

```
@@REMOVE Move_Window;
```

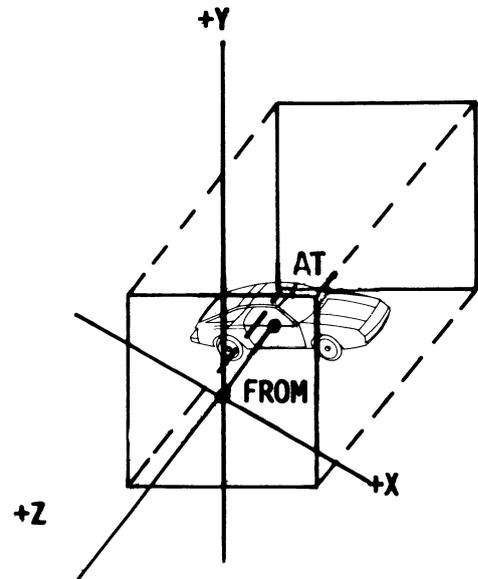
Figure 14 shows the sequence of transformations that makes `Move_Window` encompass the car.



FROM AT
Specification
of Another View



Sportscar Clipped
in Default Window



Move Window
A Window Defined
Around Sportscar
(no clipping)

IA30456

Figure 14. Interrelation of LOOK and WINDOW Transformations

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 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 300 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 tree. 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).

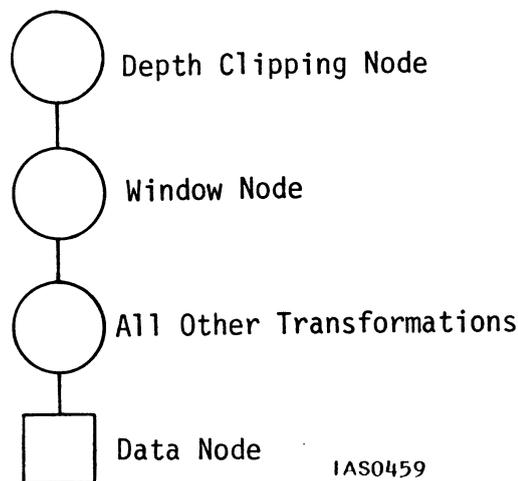


Figure 15. Set Depth Clipping Display Tree

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.

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.

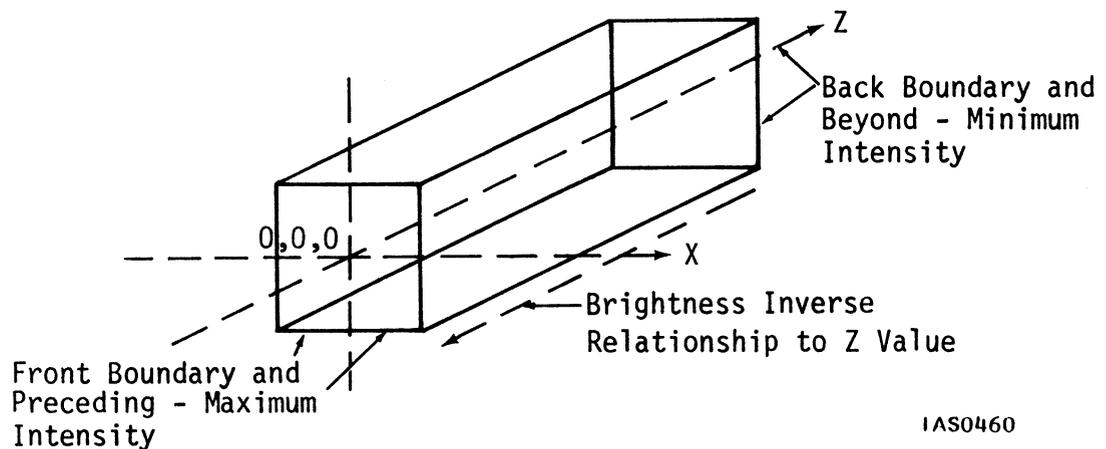
```
@@REMOVE Z_Clip;
```

Optimizing Depth Cueing

One of the ways the PS 300 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.

The brightest intensity occurs at the front Z boundary 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.

If depth clipping is not in effect, portions of objects extending past the front boundary are displayed at the maximum intensity, with no variation in brightness. Portions of objects extending beyond the back boundary are displayed at the minimum intensity, with no variation in brightness. See Figure 16.



IAS0460

Figure 16. Intensity as a Function of Z Location

Exercise 1

Change the Z boundaries of the default WINDOW to see a change in depth cueing for the Car. First display the sportscar 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 Z_Close, the front Z boundary is placed in negative Z (a placement that is legal only for orthographic windows).

```
@@ REMOVE Close;
```

Exercise 2

Refer to "Tutorial Demonstration Package" and run the WINDOW demonstration program.

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 tree 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 the *Command Summary* in Volume 3A.)

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 300 defines perspective windows two ways: using the FIELD_OF_VIEW command and using the EYE 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 17 shows how a perspective window differs from an orthographic window:

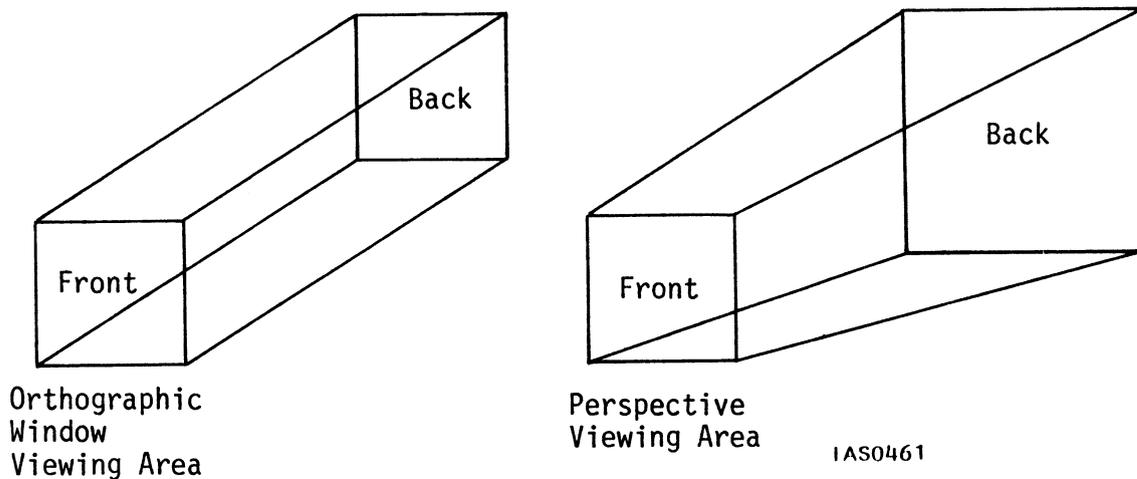


Figure 17. Orthographic Window Compared to Perspective Window

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, height equal to width.

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 18.

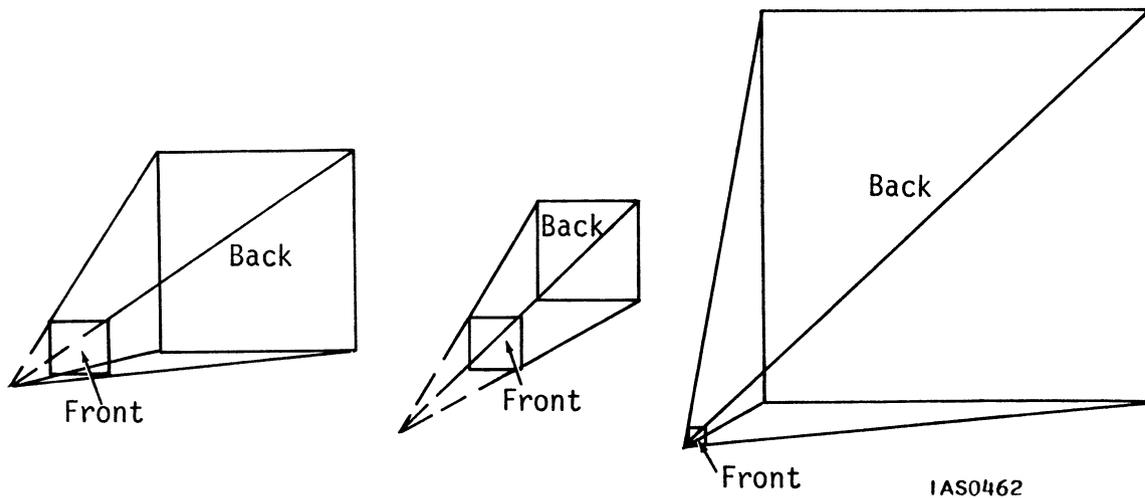


Figure 18. Angles Between Opposing Sides of the Pyramid

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.

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 tree (see Figure 19).

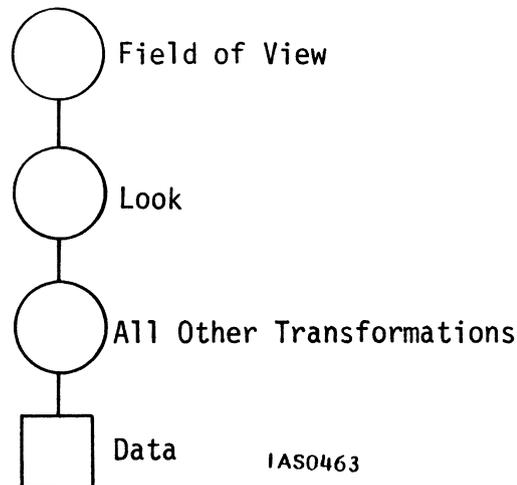
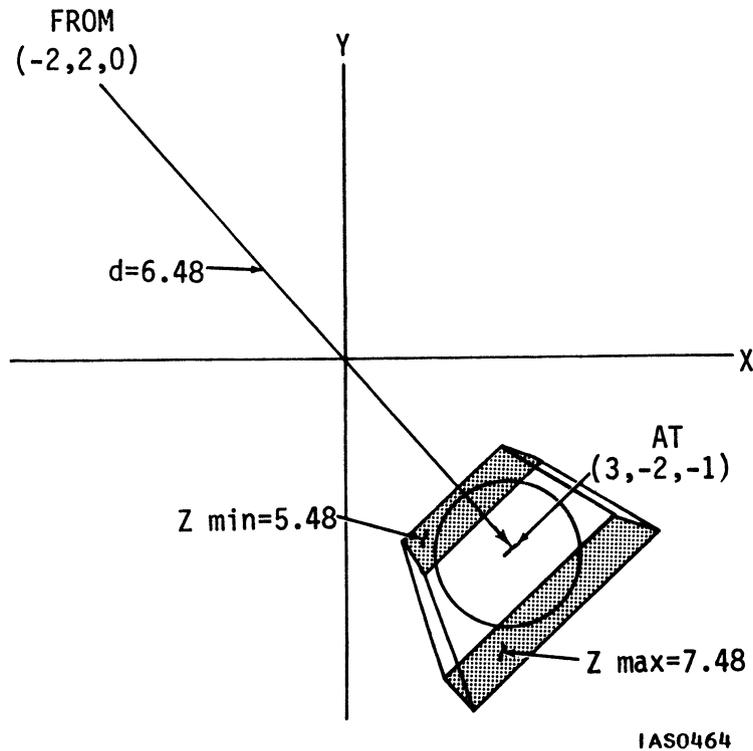


Figure 19. Display Tree With FIELD-OF-VIEW Node

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 20).



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 z max, z min designations.

Figure 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.

Exercise 1

Position the sportscar 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 0,0,5 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 300 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 21).

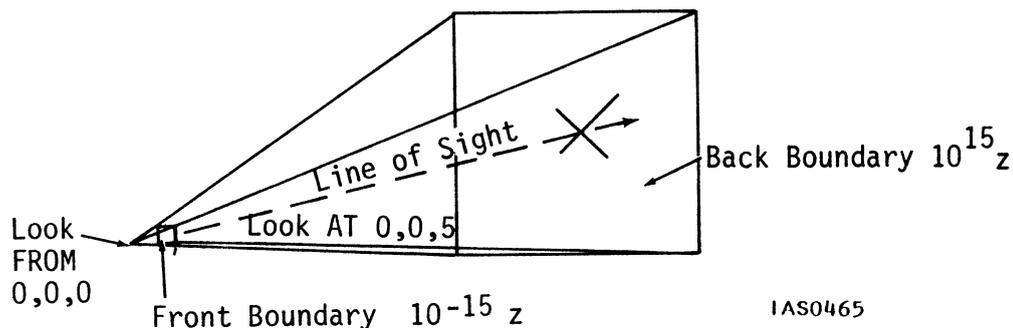


Figure 21. Using FIELD_OF_VIEW With LOOK

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 22 shows the car in the frustum of view just created.

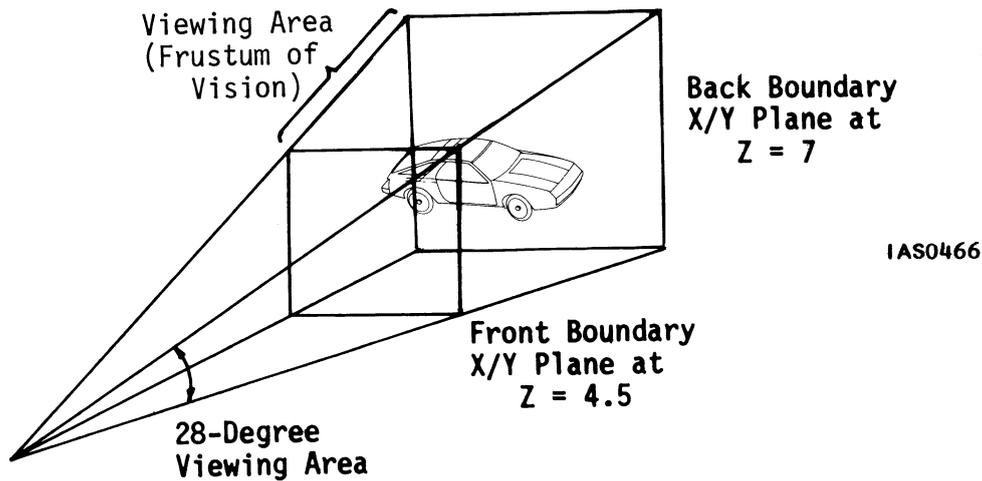


Figure 22. Setting Front and Back Boundaries

Exercise 3

Refer to "Tutorial Demonstration Package" and run the FIELD_OF_VIEW demonstration program. Before you begin, remove Perspective. Enter:

```
@@REMOVE Perspective;
```

Using the EYE Command

In addition to FIELD_OF_VIEW, there is another command that creates a perspective window. Like FIELD_OF_VIEW, the EYE 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 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 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 300 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 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 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 23).

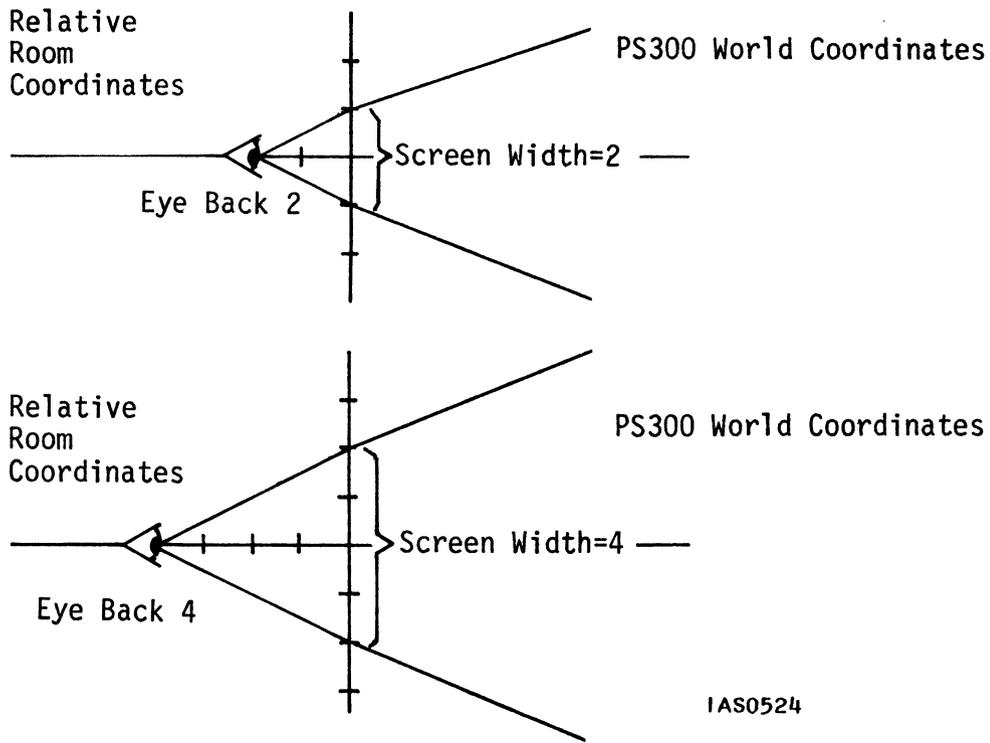


Figure 23. Relative Room Coordinates

The line of sight established by the LOOK transformation may not point at what you are looking at when you use the EYE transformation. The eye transformation creates its own sightline relative to the line of sight established by the LOOK transformation. As shown in Figure 24, the LOOK transformation establishes a line of sight to the viewed object. With EYE, 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.)

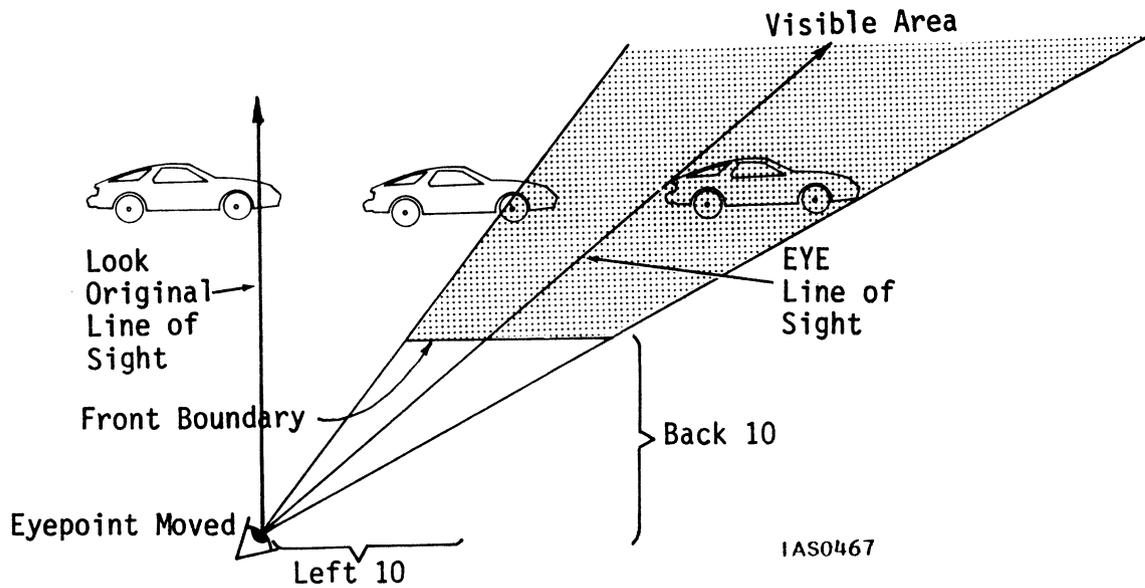
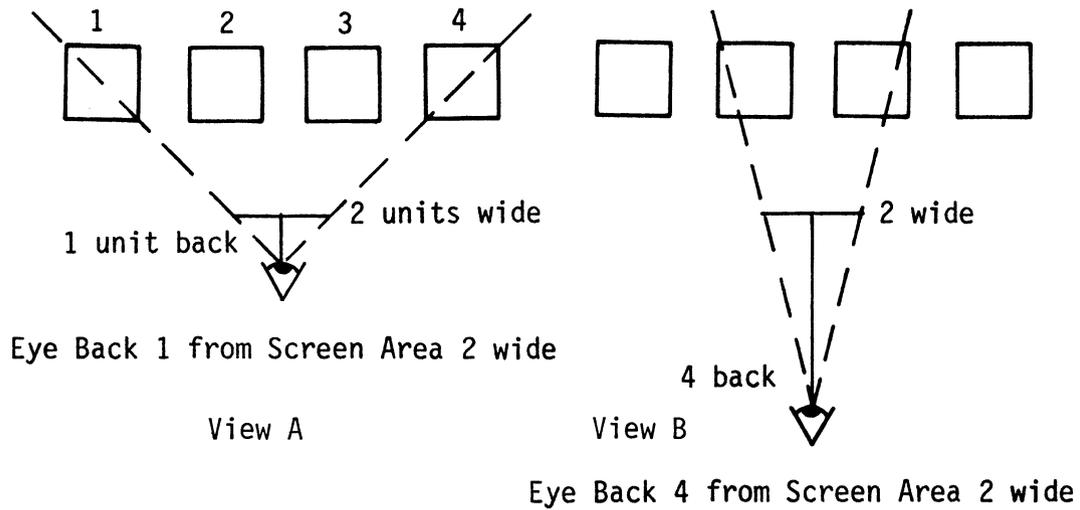


Figure 24. Line of Sight for LOOK and EYE

In the simplest instance of using the EYE 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 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.



IAS0525

Figure 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 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.

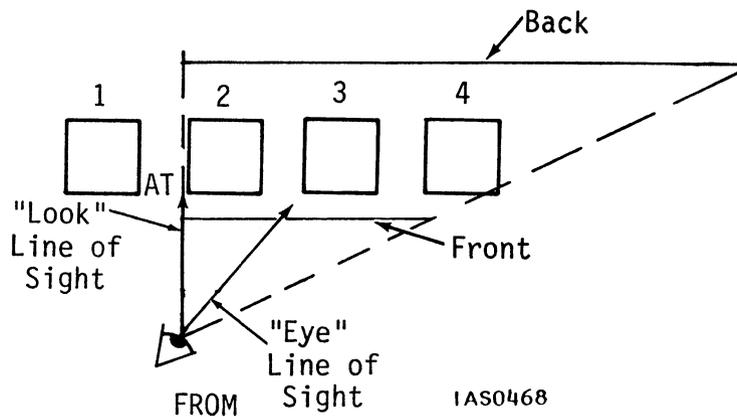


Figure 26. Moving Eyepoint Back and Left

As with `FIELD_OF_VIEW`, you must set boundaries correctly with the `EYE` 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` command, Z boundaries remain orthogonal to the Z axis. For example, in Part A of Figure 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` changes the line of sight, the boundaries do not shift. Instead, the viewing area, the frustum of vision, becomes skewed.

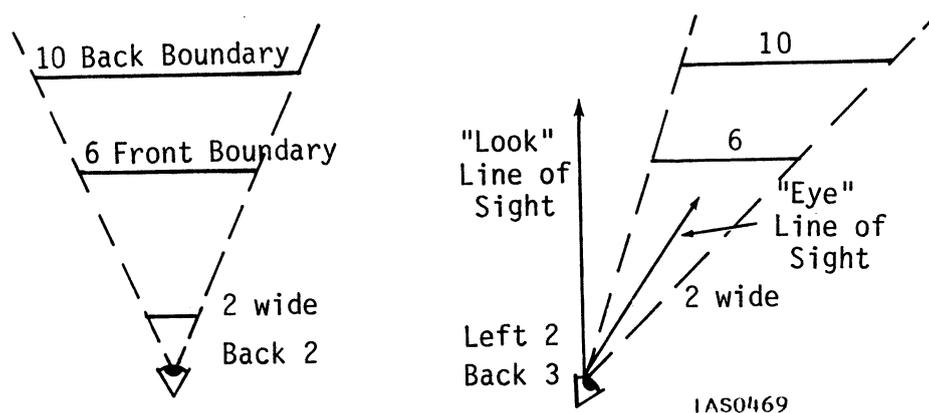


Figure 27. Boundaries Using the `EYE` Command

Exercise 1

Run the `EYE` demonstration program. (Refer to "Tutorial Demonstration Package.")

Exercise 2

Create instances of Car to the right and the left of the original sportscar 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 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 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 sportscars have been placed in front of the front boundary (depth clipping is off by default) to appear at maximum intensity.

```
@@Eye_Locate1 := 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 28.

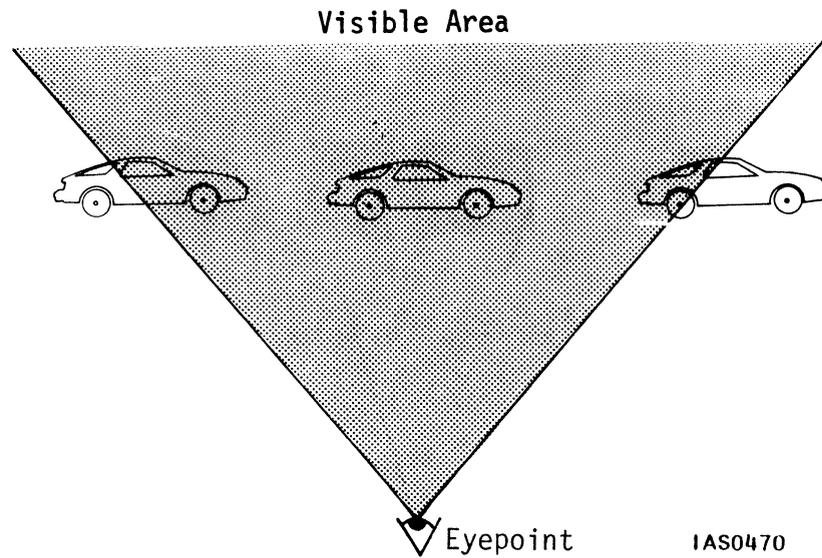


Figure 28. EYE View of Cars

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;

```

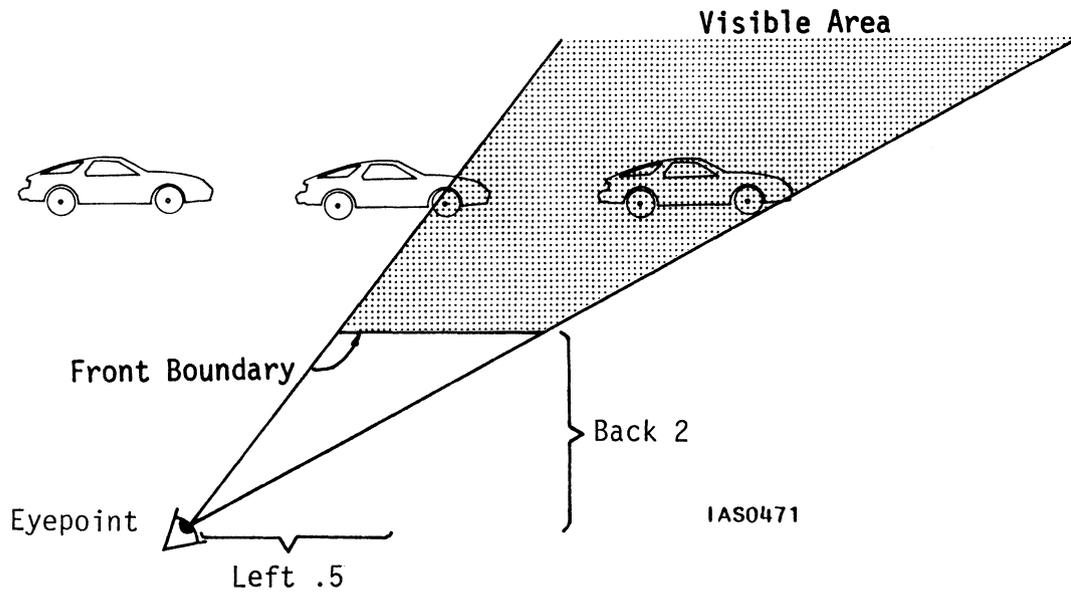


Figure 29. EYE View of Car2

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 command. In the last example, the values in the EYE 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 300 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 300 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.

Using a 4x4 Matrix to Specify a Perspective Window

The EYE 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 tree. The EYE operation node should also be directly above the LOOK operation node in the display tree.

You can build your own customized 4x4 matrix in lieu of the one created by the FIELD_OF_VIEW or EYE 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 *Command Summary* in Volume 3A.)

SPECIFYING A VIEWPORT

In addition to the two types of viewing transformations, establishing a line of sight and specifying a viewing window, the PS 300 lets you specify a portion of the full screen in which an orthographic or perspective window is displayed. Up to this point, all windows specified in examples have been projected onto the full screen of the PS 300. The PS 300 maps a window to the full 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 tree. A logical placement, though, is above the windowing transformation.

Each viewport is defined in terms of a current viewport; 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 30). The default intensity range available for any 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 range have nothing to do with world coordinate values.

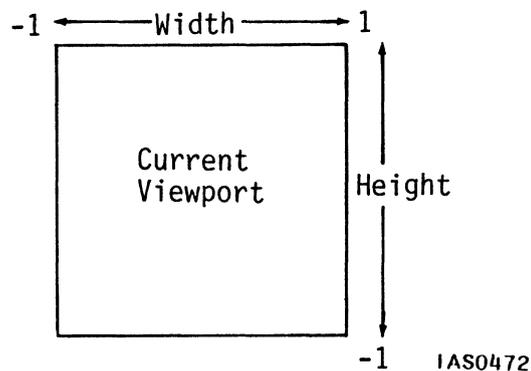


Figure 30. Current Viewport Dimensions

Exercise 1

First display the Car in the default full-screen 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.

Exercise 2

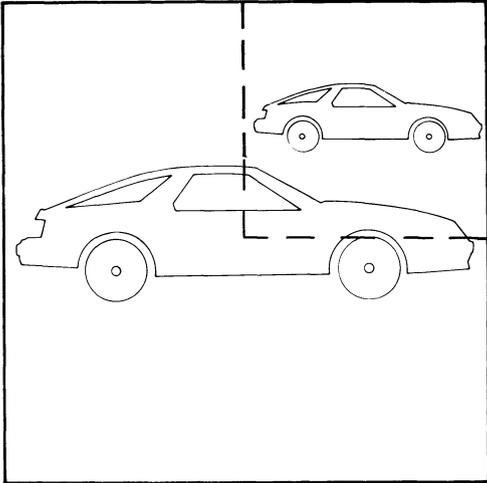
Define a viewport to be the upper right corner of the default full-screen viewport by entering;

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

```
@@DISPLAY Port2;
```

```
@@REMOVE Car;
```

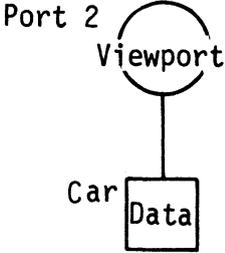
Now the upper right corner of the screen becomes the current viewport and the default window is mapped to it (Figure 31).



IAS0473

Figure 31. Port2 - Upper Right Quadrant

The display tree for this viewport applied to car is shown in Figure 32.



IAS0474

Figure 32. Display Tree for Port2

Example 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;

```

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

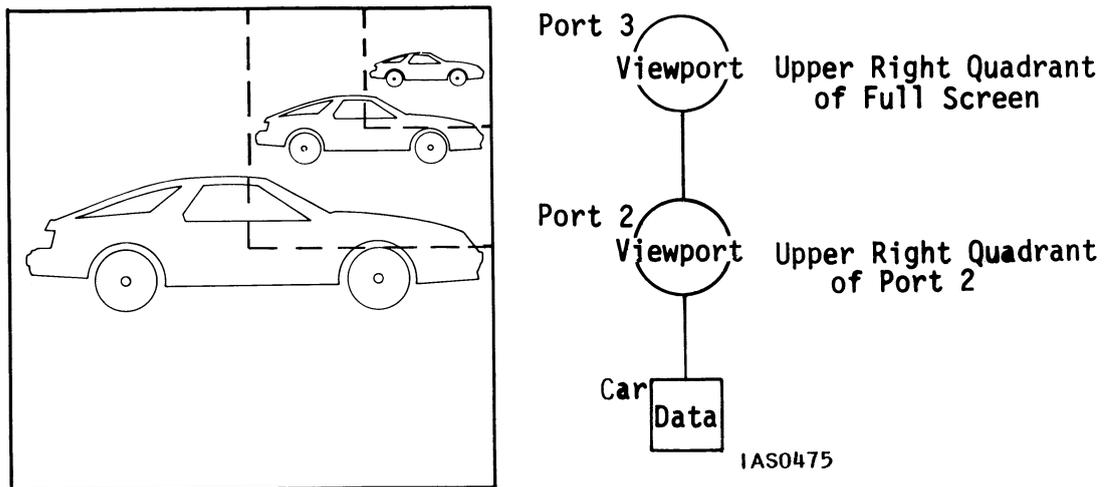


Figure 33. Port3 and Associated Display Tree

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

```
@@INITIALIZE DISPLAY;
```

Displaying Multiple Viewports

The PS 300 allows multiple viewports to be displayed simultaneously. The exercises that follow create four 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

Exercise 1

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 module, you will need to add the two following lines of code to the above listing:

```

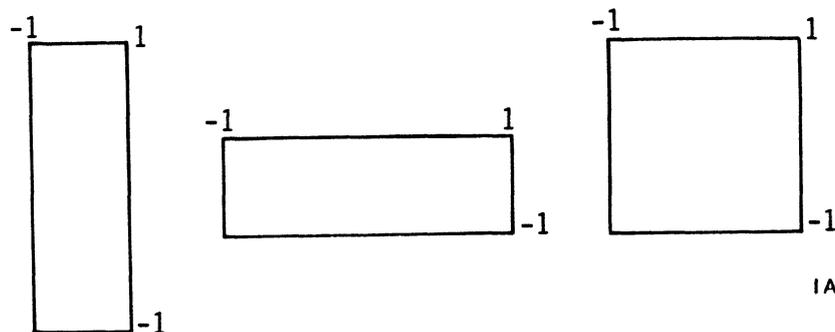
@@Perspective := FIELD_OF_VIEW 28 FRONT=4.5 BACK=7
                APPLIED TO Look;

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

```

Using Non-Square Viewports

Sometimes a non-square viewport is needed. Remember all viewports are defined in terms of a current viewport having dimensions of -1 to 1 in height and width. These dimensions apply to non-square viewports as well (see Figure 34).



IAS0476

Figure 34. Dimensions of a Non-Square Current Viewport

A non-square viewport can cause distortion of displayed data. To compensate for such distortion, objects can be viewed in a non-square 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 2:1, 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 non-square front boundaries. Perspective windows always have square front boundaries, so objects are distorted if a perspective window is displayed in a non-square 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.

Exercise 1

Map a square window to a non-square viewport to observe the resulting distortion. First impose the PS 300 default orthographic (square) window by removing the previous perspective view:

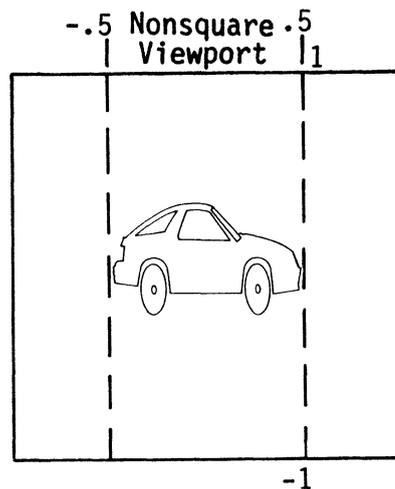
```
@@REMOVE Four_View;
```

Then create the non-square 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 35).



Default Window (1 to -1 in X, 1 to -1 in Y)

IAS0477

Figure 35. Square Window Mapped to Non-Square Viewport

Exercise 2

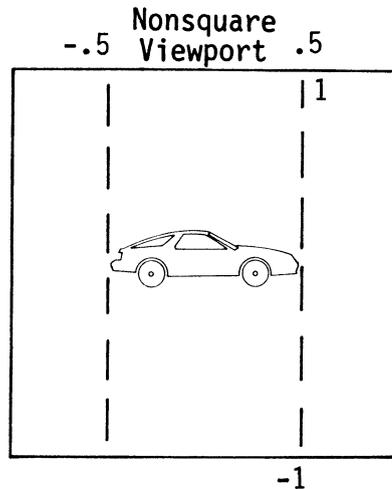
Compensate for the distortion by creating a non-square window for the non-square viewport by entering:

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

@@ DISPLAY Non_Square_Window;

@@ REMOVE Non_Square;
```

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 36).



Nonsquare Window with Matching Aspect Ratio
(1 to -1 in X, 2 to -2 in Y)

IAS0478

Figure 36. Non-Square Window Mapped to Non-Square Viewport

To clear the display enter:

```
@@REMOVE Nonsquare_Window;
```

Setting an Intensity Range for a Window in the Viewport

A 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 (dimkest, back boundary) to 1 (brightest, front boundary).

Set the 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.

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
- PS 300 Display(s) on which data will be appear
- Colors of lines that form the image.

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) and LOOK transformations in a display tree.

Setting Intensity

Remember that with the VIEWPORT command, an intensity range can be specified which applies to the window being displayed in the current viewport. In addition to this method, 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 the *Command Summary* in Volume 3A 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 *Command Summary* in Volume 3A for information on assigning intensities to vectors using the VECTOR_LIST command.)

Setting Displays ON and OFF

If you have more than one display screen on your PS 300, the SET DISPLAY attribute can be used to make objects appear on some displays and not appear on others. This selection capability is built into an object's display tree by adding a SET DISPLAY node.

By default, when an object is displayed, it appears on all the screens in the PS 300 configuration. Selected displays can be turned on and off for an object by supplying a display number in the SET DISPLAYS command. For example, if you have four displays (0, 1, 2, 3) in your system, the following commands would cause Another_Object to appear only on displays designated 2 and 3:

```
@@Set := SET DISPLAYS 1,2 OFF APPLIED TO Another_Object;  
@@DISPLAY Set;
```

In multi-screen systems, all screens can be turned off or on for a given object by using the following commands:

```
Displays_Off := SET DISPLAYS ALL OFF APPLIED TO An_Object;
```

or

```
Displays_On := SET DISPLAYS ALL ON APPLIED TO An_Object;
```

If the SET DISPLAYS ALL ON(OFF) is placed above a SET DISPLAYS 0 (1, 2, 3) node in a display tree, the displays specified in the lower node can be controlled with the SET DISPLAYS ALL node.

For example, if

```
Set_All := SET DISPLAYS ALL ON THEN Set_1;  
Set_1 := SET DISPLAYS 1 OFF THEN Object;
```

displaying Set_All will override Set_1, and Object will appear on display 1. However, if Set_1 is displayed, Object will not appear on display 1.

Both SET DISPLAYS nodes can be toggled on and off using Booleans (refer to Volume 3A, *Command Summary*).

Exercise

To demonstrate the effect of a SET DISPLAYS attribute in a system having just one display, apply a SET DISPLAY 1 OFF attribute to Car.

```
@@No_Display := SET DISPLAY 0 OFF APPLIED TO Car;
```

```
@@DISPLAY No_Display;
```

The car does not appear on the screen. Now toggle the OFF condition to ON in the SET DISPLAY node:

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

The Car is now displayed.

Setting Color

If your system has the optional Color Shadow Mask (CSM) Calligraphic Display, you can specify colors for objects in two ways.

One method makes all vectors in a displayed object the same color. The second method lets you blend color along line segments between endpoints; that is, a different color can be specified for each vector in the VECTOR_LIST, and the line connecting two vectors blends from the color of the first vector to the color of the second vector. This section teaches you how to display entire objects (entire vector lists or character strings) as a single color. (For color blending between vectors, refer to the VECTOR_LIST command in the *Command Summary* in Volume 3A.)

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, the condition of a SET INTENSITY command, and the setting of the INTENSITY dial on the PS 300 Display.

The PS 300 lets you choose from 120 hues. Selectable hues correspond to the values on the color wheel shown in Figure 37, with blue at 0 and 360, red at 120, and green at 240.

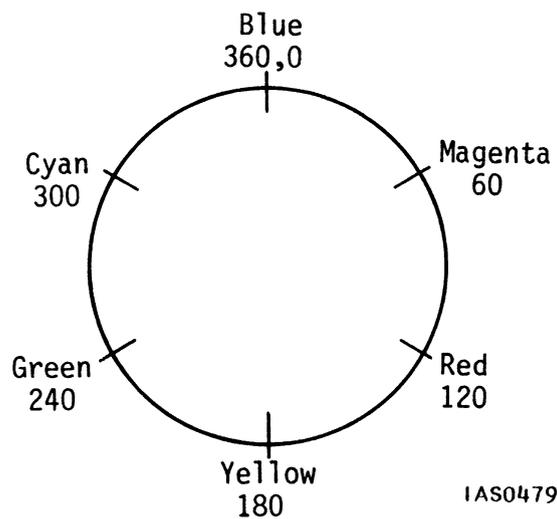


Figure 37. Color Wheel

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).

Before displaying colored objects, the refresh rate of the PS 300 Line Generator should be slowed. Slower speed means that endpoint matching will be optimized. (When displaying in color, running the line generator at full speed will make endpoint matching less accurate. The 19-inch Calligraphic Color monitor must be run at half speed for good endpoint matching.)

The Line Generator is slowed with a SET CSM ON command. This command is usually invoked as a system default in PS 300 systems using the color monitor. Your system manager should include such commands in the SITE.DAT (site data) file that is loaded when the system is booted. (Refer to Volume 5 for further SITE.DAT information.)

The following exercise assumes that the CSM ON condition has been set by the SITE.DAT file.

Exercise

1. Make the Car red, fully saturated.

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

```
@@DISPLAY Redcar;
```

2. 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;
```

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), and setting up a portion of the PS 300 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 full screen.

The WINDOW command creates orthographic views. The FIELD_OF_VIEW and EYE 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, the line of sight can be offset, creating a skewed frustum of vision.

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

The following sections summarize concepts in this module.

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 tree and immediately below the windowing transformation (WINDOW, FIELD_OF_VIEW, EYE).

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. They can be nonsquare. See the section of this module on VIEWPORTS for uses of nonsquare WINDOWS.

- The command format to specify a WINDOW is:

```
name := WINDOW X=xmin:xmax Y=ymin:ymax [FRONT boundary = zmin  
BACK boundary = zmax] APPLIED TO name;
```
- The WINDOW transformation is done in a 4x4 matrix. To work properly, the WINDOW transformation must be the topmost matrix node in a display tree.

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 tree.

Important Concepts About the EYE Command

- EYE 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 is always square.
- With the EYE transformation, the line of sight is not necessarily colinear with the from/at line in LOOK.
- If the eye position is not colinear 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 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 transformation is performed by a 4x4 matrix. To work properly, the EYE operation node must be above all other transformation nodes and directly above the LOOK operation node in the display tree.

Important Concepts About Viewports

- A viewport is the area of the PS 300 screen to which a window is mapped.
- A viewport is defined in terms of a current viewport. The dimensions of any current viewport is -1 to 1 in X and in Y.
- Viewports may be nested in other viewports without limit.
- Multiple viewports can be displayed simultaneously.
- Non-square viewports distort displayed objects unless the viewed window has the same aspect ratio as the non-square viewport.
- An intensity range for a window (WINDOW, EYE, etc) can be specified for a viewport.
- The command format to specify a VIEWPORT is:

```
name := VIEWport HORIZontal - hmin:hmax VERTICAL = vmin:vmax  
[INTENSity = imin:imax] APPLIED TO name1;
```

- 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 tree.

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) and LOOK transformations in a display tree.
- 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 DISPLAY attribute is used to enable and disable the display of objects on specific PS 300 displays. In multi-screen systems, all or some screens can be turned on or off for a given object using this attribute.
- 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.

2

3

4