DORÉ

DORÉ PROGRAMMER'S GUIDE

Stardent Computer Inc.

Part Number: 340-0107-01

Change History

340-0005-02 Original
340-0107-01 January, 1990

CONTENTS

10 Views, Frames, and Devices

11 Conditionals

12 Methods

13 System Functions

14 User-defined Primitives

Index

List of Figures

PREFACE

This manual explains the key terms and concepts involved in everyday programming with the Doré Library. Each chapter includes sample code fragments, written in C and Fortran, to illustrate the topics under discussion.

If you're new to graphics programming, you will probably want to read this manual thoroughly so that you have a complete picture of how to use the Doré graphics subroutine library. Each chapter begins with a brief overview of the material presented and a list of the terms and concepts introduced in the chapter.

If you're an experienced graphics programmer but are new to Doré, you'll probably dig right into the code examples and illustrations and then move on to the *Doré Reference Manual.* Be sure to skim at least the following chapters of this manual:

- Chapters 1 through 3 (introductory material)
- Chapter 4, "Objects and Groups"
- Chapter 10, "Views, Frames, and Devices"
- Chapter 12, "Methods"

Chapter Summaries

This chapter briefly summarizes the remaining chapters in this manual. It also refers you to related manuals and texts on graphics programming.

Chapter 1, *Overview,* outlines the key features and benefits of the Doré Library.

Chapter 2, *Writing a Doré Program,* serves as an introduction to the structure and components of a Doré program. Although the Doré Library contains several hundred functions, you need only a small

subset to produce both simple 3D models such as those created in the chapter example and elaborate ray-traced images. This chapter also contains information on single- and double-precision versions of the Doré application program interface.

Chapter 3, *Functional Groupings*, lists the major groupings for the Doré functions and describes the Doré naming conventions for functions, types, and constants.

Chapter 4, *Objects and Groups*, describes how to create objects and add them to groups, and how to reference groups within other groups. A simple database for the axle group of a car is constructed in the sample code.

Chapter 5, *Primitive Objects*, explains concepts and offers examples related to the major primitive objects in the Doré.

Chapter 6, *Primitive Attributes*, describes some of the important attribute objects used to modify primitive objects. A detailed discussion of how attribute objects affect the attribute stack is presented.

Chapter 7, *Geometric Transformations*, describes how geometric transformation attribute objects can be used to modify primitive and studio objects. A detailed discussion of the Current Transformation Matrix and how geometric transformations modify it is offered.

Chapter 8, *Text*, describes the text primitives and the attributes that affect them.

Chapter 9, *Cameras and Lights*, describes studio objects and explains how to use them in defining a scene. Important studio attributes are also covered in this chapter.

Chapter 10, *Views, Frames, and Devices*, describes these organizational objects and how a scene is transformed from modeling coordinates to frame coordinates to device coordinates.

Chapter 11, *Conditionals*, discusses the use of conditional elements, such as callbacks, name sets and filters, and the executability set, to conditionally affect execution of the Doré display tree.

Chapter 12, *Methods*, describes the key functions that involve traversal of the Doré database: rendering, picking, and computing bounding volumes.

Chapter 13, *System Functions*, describes key system functions, as well as the use of valuators and input slots to provide asynchronous input to Doré.

Chapter 14, *User-Defined Primitives*, describes how to create new Doré primitives. It also recommends conventions for naming and implementing user-defined primitives.

Related Manuals and Texts

For a complete description of all Doré functions, see the *Doré Reference Manual*. The *Doré Porting and Implementation Manual* provides information for system programmers to help them port Doré to new computer platforms. The manual covers the general structure of the Doré implementation and the specific modules that must be modified or reimplemented to port Doré to a different platform. Appendices E and F of the *Doré Reference Manual* contain information on the capabilities and restrictions for specific platforms and devices.

If you are new to graphics programming, one of the following texts can provide useful background information:

- *Principles of Interactive Computer Graphics* by William M. Newman and Robert F. Sproull, published by McGraw-Hill.
- *Fundamentals of Interactive Computer Graphics* by James D. Foley and Andries Van Dam, published by Addison-Wesley.

Conventions Used in This Manual

The C name for each Doré function appears first in the text, followed by the Fortran name in angled brackets <>. A different font, as shown below, is used for code examples:

```
This is a sample of the font used for program examples.
```

For all code examples, the C code appears first, followed by the corresponding Fortran code.

OVERVIEW

CHAPTER ONE

Doré is a powerful graphics library that enables you to produce both attractive dynamic image sequences and near-photographic quality images. With the Doré Library, you can combine full-color, high-resolution, three-dimensional images with computationally intensive supercomputer applications. Doré is device-independent and is designed to interface easily with existing software applications.

Doré, which stands for Dynamic Object Rendering Environment, is uniquely suited to advance the needs of a variety of users such as the following:

- A scientist in a research center wants to simulate wind tunnels to save the cost of an actual run. His group has developed some very sophisticated simulation packages that run on a large computer. The programs currently output to a commercial raster frame buffer display. This setup has been adequate in the past, but now he wants to enhance his application to view the computed flow fields interactively.

- A team of engineers has developed an expert system that monitors and controls a complex array of valves and gauges at a chemical refinery plant. The expert system was developed internally in PL1, but the released product needs to be in C for portability. A good human interface to the system is also required.

Because of its ease of use, its portability and extensibility, and its powerful graphics features, Doré offers these users a complete solution for their needs. Doré handles a wide range of applications—from the very simple to the more comples and compute intensive applications such as molecular modeling, fluid dynamics, and mechanical stress loadings.

High Quality, Realistic Images

Using Doré, you can create highly realistic, full-color images. With its standard ray tracer, Doré includes some of the most advanced graphics features available today, including transparency, true shadows, and environmental reflection. Images can be rendered in a variety of styles: points, wireframe, faceted, and smooth-shaded surface types. Different parts of an object can be represented in different ways—for example, one part could be a simple wireframe, and another part could be rendered as a colored solid with specular highlights.

Both Dynamic and Ray Traced Images

Doré allows you to easily build applications that can interactively manipulate three-dimensional images. You can rotate objects, or parts of objects, in any direction. Like a camera, you can zoom in on the details of a particular scene, or pan across it. Valuators provide a convenient mechanism for making changes in the Doré database in response to asynchronous external events such as mouse or dial movements and keyboard input.

Doré provides for multiple rendering styles, offering a range from dynamics to elaborate ray-traced imagery. A unique Doré feature is that it uses the same database for generating both real-time dynamic image sequences and highly realistic static images. You don't need additional coding to create a variety of representations of the same image.

Ease of Use

The Doré Library was designed for ease of use. Although some familiarity with basic graphics programming concepts is assumed, you don't have to be a graphics expert to use the Doré library. The Doré programmer needs to supply only one basic description of an object's geometry and its attributes (how it is colored, lit, and modified). Then, Doré, not the programmer, takes care of displaying the object in a variety of styles. Decomposition, shading, and hidden surface algorithms are handled by Doré as well.

Device Independence

Because Doré is device independent, you do not need to change the code every time a piece of graphics hardware changes. And you have the flexibility of being able to run your software on conventional supercomputers, mini-super computers, or

workstations, so that a large number of users across a network can easily share a common graphics interface.

What Is in the Doré Library?

Doré provides a comprehensive set of tools for creating graphics, including:

- primitives, such as polygons and patches, for representing objects. Advanced primitives include polygonal meshes, closed cubic surfaces, and non-uniform rational B-spline surfaces.
- surface properties, such as ambient, diffuse, and specular light reflectance. Other surface properties available in Doré include transparency, shadows, and environmental reflection.
- features that enable you to describe a graphics "scene" that includes the objects, as well as the lights that illuminate them and the cameras used to "view" them.
- rendering representations, including points, wireframe, faceted, and smooth-shaded surface types, and combinations of styles in the same scene.
- a wide array of functions that enable you to edit your graphics database.

For a more detailed look at how the programmer combines these routines to create an image, see Chapter 2, "Writing a Doré Program."

WRITING A DORÉ PROGRAM

CHAPTER TWO

This chapter provides an introduction to the structure of a Doré program and to some of the commonly used functions in the Doré Library. Doré programming is object-based, an approach which leads to programs that are both modular and easily extensible.

The Doré Process

The process of defining a scene with the Doré Library, taking its picture, moving the camera, taking its picture again, modifying the scene, and taking another picture is analogous to actual photography. Imagine an animation artist in his studio preparing a scene for a science fiction movie, using models and cameras to create the illusion of starfighters in a battle scene. The artist

(1) Constructs the models of the starfighters from smaller parts.

(2) Positions the models in the scene, usually in front of a backdrop picture of stars and galaxies.

(3) Positions the camera and chooses a lens to obtain the proper image framing and depth of field.

(4) Adjusts the light level and positions spotlights to make everything in the scene visible, to highlight portions of the scene, and to cast shadows.

(5) Takes a picture. To simulate motion, the artist attaches everything in the scene to moving arms or platforms. Then, all the pieces in the scene can be moved a small amount and another picture taken, then moved again and another picture taken, and so on until the whole sequence is filmed.

A Doré application modeling an internal combustion engine may not be as thrilling as a starfight, but the series of steps from basic

parts construction through scene generation, camera positioning, and lighting is the same. The application must define each part of the engine, from nuts, bolts, and washers, to piston rods, pistons, cylinder head, and engine block. Some of these parts are used in the engine only once (the cylinder head and engine block), while other parts are used more than once (bolts, piston parts). The application positions each part together to build an image of a realistic engine. This model building applies both to construction of new parts and to viewing of existing parts and data, such as the results of stress or thermal analysis.

Objects and Methods

The Doré Library consists of *objects* and *methods*. All data is kept in the form of *objects*. Each geometric primitive, attribute, group, camera, light, device, frame, and view is a separate object. Each object has a set of *methods*, which are a set of internal functions that operate on the object. When a method is invoked, the Doré database is traversed, and the objects in the database are executed using that method. The most commonly used method is *rendering*. Other methods include picking, computing bounding volumes, and printing.

Basic Steps in Programming with the Doré Library

The basic steps in programming with the Doré Library are very similar to those used by the artist in constructing the starfighter scene. The programmer

(1) Creates the objects being modeled using primitives for points, lines, polygons, patches, and surfaces.

(2) Chooses attributes to describe the appearance of the objects—their color, shininess, whether they have shadows on their surfaces, and so on.

(3) Positions and sizes scene objects relative to one another.

(4) Positions various kinds of lights in relation to the scene.

(5) Selects a camera lens and positions the camera in relation to the scene.

(6) Asks for the scene to be rendered.

Now let's take a look at each of these steps in a bit more detail, examine some simple code fragments written in C and Fortran,

and finally, see how all the pieces work together to produce a simple scene with a yellow wireframe cylinder and a magenta box and sphere.

Step 1: Primitives

The Doré Library offers a wide range of primitives for describing the shape of displayable objects. Simple primitives create points, lines, polygons, and text, while more advanced primitives create meshes, patches, and closed cubic surfaces such as spheres and cones.

The sample program, given at the end of this chapter, creates three simple shapes—a cylinder, a box, and a sphere. In C, the function calls to create these shapes are:

```
DoPrimSurf(DcCylinder)

DoPrimSurf(DcBox)

DoPrimSurf(DcSphere)
```

In Fortran, the function calls are:

```
      CALL DOPMS(DCCYL)

      CALL DOPMS(DCBOX)

      CALL DOPMS(DCSPHR)
```

Step 2: Attributes

Next, you select the attributes to describe the appearance of the primitive objects. The sphere and the box are both magenta, and we see the surfaces of these objects. The cylinder is yellow, and it is displayed as a wireframe structure.

Specifying an object's color brings up two important concepts in the Doré Library: the concept of a *color model* and the concept of *how an object responds to light*. Later chapters go into detail on these concepts, so a brief introduction will suffice for now.

Color Model

Colors are specified by a color model and an array of color data. Currently, the Doré Library supports the RGB color model. The amounts of red, green, and blue are specified as real numbers, where (0.0, 0.0, 0.0) is black and (1.0, 1.0, 1.0) is white.

The color for the sphere and box in the example is specified in an array, as follows:

```
C code:

static DtReal magenta[] = {1.0, 0.0, 1.0};  /* amt of R, G, B */

DoDiffuseColor(DcRGB, magenta);


Fortran code:

      REAL*8 MAGENTA(3)

      DATA MAGENTA / 1.0D0, 0.0D0, 1.0D0 /

      CALL DODIFC(DCRGB,MAGENTA)
```

The color of the cylinder is defined separately in another group, the post group. The cylinder is yellow, which is specified as

```
C code:

DtReal yellow []= {0.8, 0.8, 0.0};

DoDiffuseColor(DcRGB, yellow);


Fortran code:

      REAL*8 YELLOW(3)

      DATA YELLOW / 0.8D0, 0.8D0, 0.0D0 /

      CALL DODIFC(DCRGB,YELLOW)
```

How an Object Responds to Light

An object's response to light has five basic components: ambient, diffuse, specular, transparent, and reflection. Each of these components in turn has a color, an intensity, and a switch that specifies whether the component is applied or not. The *diffuse color* is usually thought of as the object's base color. For example, the diffuse color of the cylinder in the example is yellow. The diffuse color of the box and sphere is magenta. The *specular color* is the color of the object's highlights. The *ambient color* is essentially the same as the diffuse color, so it is calculated using the diffuse color value. The *transparent color* is the color of light transmitted through the object.

Default Values for Attributes

If you do not specify a value for a particular attribute, the default value for that attribute is used. In the example, no specular color is specified, so the default color (white) is used. Don't be afraid to trust the default values for attributes in the Doré Library; they have been carefully designed to fit the most common usage.

Colored Vertices

The primitives in this chapter are created using the *DoPrimSurf <DOPMS>* function, which defines a primitive surface in a standard location and with a standard size. Other primitives, such as lines, triangles, polygons, and meshes, all use vertices explicitly to compose the object. With those primitives, you can optionally include color information in the definition of each vertex, to be used for shading computation. If you do not specify colors for the vertices, global color attributes will be used for the whole surface.

Step 3: Positioning and Sizing Scene Objects

Another important category of attributes for primitive objects is the *geometric transformations*, which affect the shape and positioning of objects in three-dimensional space. Important geometric transformations include

DoScale <DOSC>
which scales the object in x, y, and z

DoTranslate <DOXLT>
which moves the object a specified amount in the x, y, and z directions

DoRotate <DOROT>
which rotates the object a specified amount around one of the axes

In the example, the cylinder (which is located at the origin and extends in the positive z direction) is scaled in all three directions:

```
C code:

DoScale (0.4, 0.4, 2.1)


Fortran code:
```

```
CALL DOSC(0.4D0, 0.4D0, 2.1D0)
```

This scaling shrinks the cylinder in the x and y directions, thus making it more slender, and stretches it in the z direction, which lengthens it.

Objects and Groups: DgAddObj()

We now have several primitive objects—a box, a cylinder, and a sphere—and some attributes—diffuse color, scaling, and other attributes you'll learn about soon, such as representation type and surface shading. How do we begin putting them together into an ordered program? The main vehicle for organizing objects and their associated attributes into a hierarchical database is the *group*. A group is an ordered list of object pointers. Each of the *Do-* functions described above returns an object handle that can be added to a group with the function *DgAddObj <DGAO>*.

Figure 2-1 shows a simplified diagram of the objects included in the sample program's post group. Note that the attribute objects *precede* their associated primitive objects, as shown. The ordering of objects within a group is important because the objects within a group are executed in order—first the diffuse color object, then the shading object, and finally the primitive object.

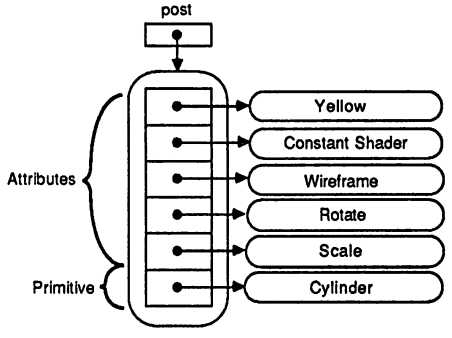


Figure 2-1. Post Group

The code for the post group looks like this:

C code:

```
post = DoGroup(DcTrue);
     DgAddObj(DoDiffuseColor(DcRGB, yellow));
```

```
        DgAddObj(DoSurfaceShade(DcShaderConstant));
        DgAddObj(DoRepType(DcWireframe));
        DgAddObj(DoRotate(DcXAxis, 1.57));
        DgAddObj(DoScale(0.4, 0.4, 2.1));
        DgAddObj(DoPrimSurf(DcCylinder));
    DgClose();
```

Fortran code:

```
      POST = DOG(DCTRUE)
          CALL DGAO(DODIFC(DCRGB, YELLOW))
          CALL DDGAO(DOSRFS(DCSHCN))
          CALL DGAO(DOREPT(DCWIRE))
          CALL DGAO(DOROT(DCXAX, 1.57D0))
          CALL DGAO(DOSC(0.4D0, 0.4D0, 2.1D0))
          CALL DGAO(DOPMS(DCCYL))
      CALL DGCS()
```

Figure 2-2 shows a diagram for the sample program's base group.

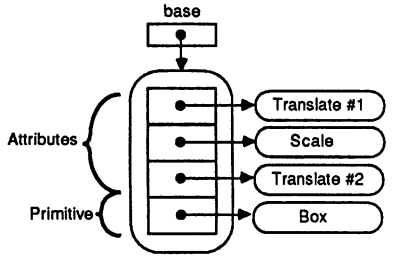


Figure 2-2. Base Group

Code for the base group looks like this:

C code:

```
    base = DoGroup(DcTrue);
        DgAddObj(DoTranslate(0.0, -3.0, 0.0));
        DgAddObj(DoScale(2.0, 2.0, 2.0));
        DgAddObj(DoTranslate(-0.5, -0.5, -0.5));
        DgAddObj(DoPrimSurf(DcBox));
    DgClose();
```

Fortran code:

```
      BASE = DOG(DCTRUE)
          CALL DGAO(DOXLT(0.0D0, -3.0D0, 0.0D0))
          CALL DGAO(DOSC(2.0D0, 2.0D0, 2.0D0))
```

```
        CALL DGAO(DOXLT(-0.5D0, -0.5D0, -0.5D0))
        CALL DGAO(DOPMS(DCBOX))
    CALL DGCS()
```

Step 4: Choosing Lights

Once the primitive objects and their attributes have been defined, the programmer sets up the *studio objects* for the scene: the *cameras* and *lights*. Like primitive objects, cameras and lights have attributes that describe them. In a Doré program, the attributes for the studio objects precede the objects they modify, just as primitive attributes precede the primitives they modify (see Figure 2-3).

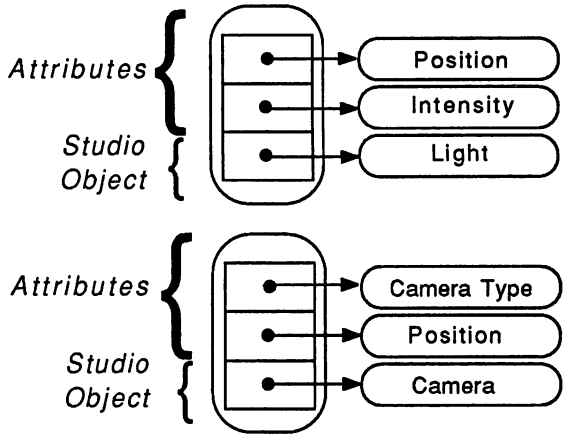


Figure 2-3. Light and Camera Attributes

This example creates a light and positions it at (1.0, 1.0, 0.5). The light has an intensity of 1.0. Code for the lights is

```
C code:

static DtPoint3
     origin = {0.,0.,0.},
     light = {1.0, 1.0, 0.5};

static DtVector3 up = {0.,1.,0.};

def_group = DoGroup(DcTrue);
          .
          .
          .

      DgAddObj(DoPushMatrix());
```

```
        DgAddObj(DoLookAtFrom(origin, light, up));/* positions light */
        DgAddObj(DoLightIntens(1.0));             /*light intensity */
        DgAddObj(DoLight());                      /* makes the light */
    DgAddObj(DoPopMatrix());
DgClose();
```

Fortran code:

```
      REAL*8 ORIGIN(3)
      REAL*8 LIGHT(3)
      REAL*8 UP(3)
C
      DATA ORIGIN / 0.0D0, 0.0D0, 0.0D0 /
      DATA LIGHT  / 1.0D0, 1.0D0, 0.5D0 /
      DATA UP     / 0.0D0, 1.0D0, 0.0D0 /
C
      DEF_GROUP = DOG(DCTRUE)
          .
          .
          .
      CALL DGAO(DOPUMX())
      CALL DGAO(DOLAF(ORIGIN, LIGHT, UP))  !positions the light!
      CALL DGAO(DOLI(1.0D0))               !light intensity!
      CALL DGAO(DOLT())                    !makes the light!
      CALL DGAO(DOPPMX())
      CALL DGCS()
```

The functions *DoPushMatrix* <*DOPUMX*> and *DoPopMatrix* <*DOPPMX*> will be discussed in more detail in Chapter 7. This functional pair is used to localize the effects of geometric transformation attributes. In this case, we want only the light to be affected by the *DoLookAtFrom* <*DOLAF*>; we will include a separate *DoLookAtFrom* to position the camera object.

Step 5: Selecting a Lens and Camera Position

The next step is choosing a camera type and positioning the camera in relation to the scene. Selecting the camera projection—parallel, perspective, or another more complex type of projection—is analogous to a photographer selecting a camera lens. Like the light discussed in the previous section, the camera can be placed in relation to the scene with the *DoLookAtFrom* function:

C code:

```
static DtPoint3
     origin = {0.0, 0.0, 0.0},
     eye_point = {0.0, 0.0, 10.0};
static DtVector3
     up = {0.0, 1.0, 0.0};
```

```
def_group = DoGroup(DcTrue);
     DgAddObj(DoParallel(10.0, -0.1, -20.0));
     DgAddObj(DoPushMatrix());
          DgAddObj(DoLookAtFrom(origin, eye_point, up));
          DgAddObj(DoCamera());
     DgAddObj(DoPopMatrix());
     .
     .
     .
DgClose();
```

Fortran code:

```
      REAL*8 ORIGIN(3)
      REAL*8 EYEPT(3)
      REAL*8 UP(3)
C
      DATA ORIGIN / 0.0D0, 0.0D0,  0.0D0 /
      DATA EYEPT  / 0.0D0, 0.0D0, 10.0D0 /
      DATA UP     / 0.0D0, 1.0D0,  0.0D0 /
C
      DEFGRP = DOG(DCTRUE)
           CALL DGAO(DOPAR(10.0D0, -0.1D0, -20.0D0)) !camera type!
           CALL DGAO(DOPUMX())
               CALL DGAO (DOLAF(ORIGIN, EYEPT, UP))  !camera position!
               CALL DGAO (DOCM())                    !creates the camera!
           CALL DGAO(DOPPMX())
          .
          .
          .
      CALL DGCS()
```

Views, Frames, and Devices

We have now completed the major steps in describing the attributes of the objects and the cameras and lights used with the primitive objects. But we need to give more organization to this collection of objects before the Doré Library knows exactly what we want it to do with the objects.

Views

A *view* is used to collect groups containing *primitive objects* and their attributes with groups containing the *viewing parameters* for those objects (cameras and lights, along with their attributes).

Primitive objects and their attributes are usually added to the display group of a view. For example:

C code:

```
view = DoView();
DgAddObjToGroup(DvInqDisplayGroup(view), obj_group);
```

Fortran code:

```
      VIEW=DOVW()
      CALL DGAOG(DVQIG(VIEW), OBJGRP)
```

Studio objects and their attributes are usually added to the definition group of a view. For example:

C code:

```
view = DoView();
DgAddObjToGroup(DvInqDefinitionGroup(view), def_group);
```

Fortran code:

```
      VIEW=DOVW()
      DGAOG(DVQIG(VIEW),DEFGRP)
```

In this example, we want the primitive objects, as well as the camera and light we created, added to the same view ("view").

Frames

Next, the view is added to a frame. A frame can contain multiple views, much as a bulletin board could have a number of notices pinned to it. In our example, the view fills the entire frame.

C code:

```
frame=DoFrame();
DgAddObjToGroup(DfInqViewGroup(frame), view);
```

Fortran code:

```
      FRAME=DOFR()
      CALL DGAOG(DFQVG(FRAME),VIEW)
```

Devices

Finally, the frame is assigned to a particular device. Figure 2-4 shows the hierarchical relationship between views, frames, and devices.

```
C code:

device = DoDevice("ardentx11", "");
DdSetFrame(device, frame);


Fortran code:

      DEVICE=DOD('ardentx11',9,'',0)
      CALL DDSF(DEVICE, FRAME)
```

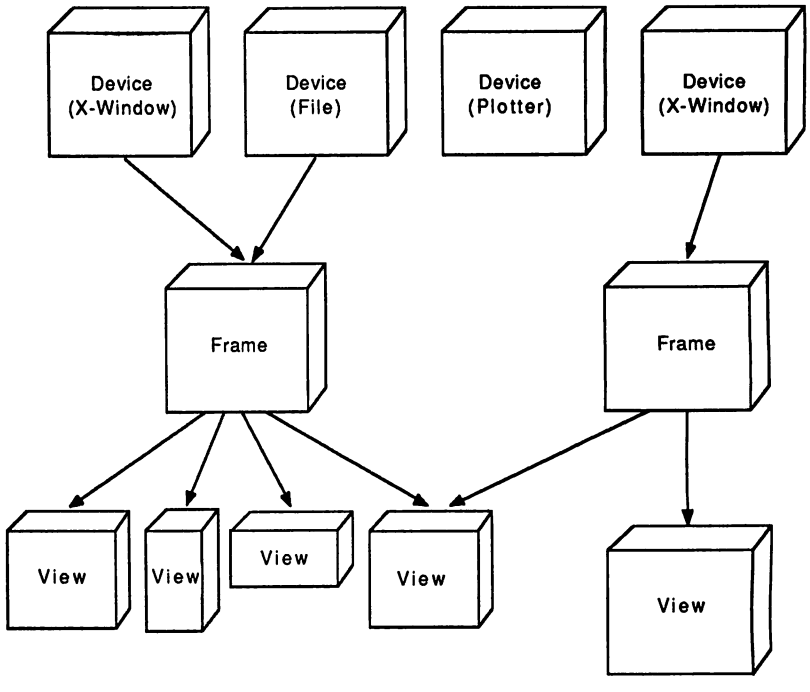


Figure 2-4. Views, Frames, and Devices

Step 6: Rendering the Scene

The final step in displaying a scene is to request the rendering with one of the update commands. Our example uses *DdUpdate <DDU>* to invoke the renderer. This view will be rendered with the real-time renderer, which is the default renderer. With standard Doré, two renderers are currently supported:

DcRealTime <DCRLTM>
for fast, dynamic display rendering

DcProductionTime <DCPRTM>
for the most realistic rendering, which requires the most time

The *DvSetRendStyle <DVSRS>* function selects the renderer.

See Appendix E in the *Doré Reference Manual* for a list of available renderers.

Immediate Mode Execution

In this example, we first created the objects that made up the database hierarchy. Then we invoked the rendering method, which traversed the objects making up the database.

The Doré Library can also be used in *immediate mode*, which is ideal for simple databases that can be traversed very quickly. In immediate mode, objects are created and then immediately executed. The objects exist only during database traversal. See Chapter 12, "Methods," for more information on immediate mode.

Single- and Double-Precision Application Program Interfaces

Different implementations of Doré on different platforms may provide support for either single- or double-precision real numbers in the application program interface; that is, the real number values and arrays passed into Doré subroutine calls may be either single or double precision. Be sure to check the Release Notes of your Doré implementation on whether it provides a double- or single-precision interface.

Some platforms provide support for both single- and double-precision reals. In these cases, you will need to consider tradeoffs among different factors: amount of space required for each precision, the difference in precision between single- and double-precision reals, and the affect on speed of choosing one type of precision over the other.

Another concern in either case is the portability of the Doré application between Doré implementations supporting different precisions. The following paragraphs describe some of the factors that must be taken into consideration to achieve portability.

Single- and Double-Precision Application Program Interfaces (continued)

C Code

If certain standard coding practices are followed, C code should be portable between single- and double-precision Doré implementations. For values of type *DtReal*, the precision should be explicitly declared rather than assumed (for example, through type casting) wherever arrays are concerned. In general, arrays passed in to Doré should be explicitly declared as *DtReals*. Avoid performing I/O with *DtReals*, since the precision will differ with different machines.

Fortran Code

At present, Fortran code may not be portable between hardware that supports only single-precision reals and hardware that supports only double-precision reals. The Fortran examples in this manual all use double-precision reals.

For double precision, all real numbers are declared as REAL*8. For single precision, all real numbers are declared as REAL*4. In double-precision the constants include *D0* after their values; in single-precision implementations, the *D0* is omitted. (Real constants in data statements can be written as double precision for both single- and double-precision implementations because they will be converted to single precision if necessary by most Fortran compilers.) Arrays must be explicitly declared as REAL*4 for single precision or REAL*8 for double precision.

Complete Sample Program

The complete sample program used in this chapter's examples is shown below. The program creates and displays three objects—a cylinder, a box, and a sphere. It also creates a definition group with a camera and a light.

```
Sample Program in C

#include <dore.h>                          /* standard Doré include */


main()
{
     DtObject device, frame, view;         /* declare variables */
     DtObject post, base, def_group, obj_group;
     static DtPoint3
          origin = {0.0, 0.0, 0.0},        /* x, y, z values */
          eye_point = {0.0, 0.0, 10.0},
          light = {1.0, 1.0, 0.5};
```

```
static DtVector3
     up = {0.0, 1.0, 0.0};
static DtReal
     magenta[] = {1.0, 0.0, 1.0},  /* values of red, green, blue */
     yellow[] = {0.8, 0.8, 0.0},
     sky_blue[] = {0.3, 0.3, 1.0},
     sds[] = {0.8};                /* for subdivision specification */

DsInitializeSystem(0);             /* initialize Doré */
device = DoDevice("ardentx11", "");/* open dynamic display device */
frame = DoFrame();                 /* create a frame */
DdSetFrame(device, frame);         /* add frame to the device */
view = DoView();                   /* create a view */
DvSetBackgroundColor(view, DcRGB, sky_blue);
                                   /* view background color*/
DgAddObjToGroup(DfInqViewGroup(frame), view);
                                   /* add view to frame */

def_group = DoGroup(DcTrue);
                                   /* creates group with a parallel */
                                   /* camera and a light */
     DgAddObj(DoParallel(10.0, -0.1, -20.0));
     DgAddObj(DoPushMatrix());
          DgAddObj(DoLookAtFrom(origin, eye_point, up));
          DgAddObj(DoCamera());    /* creates the camera */
     DgAddObj(DoPopMatrix());
     DgAddObj(DoPushMatrix());
          DgAddObj(DoLookAtFrom(origin, light, up));
          DgAddObj(DoLightIntens(1.0));
          DgAddObj(DoLight());     /* creates the light */
     DgAddObj(DoPopMatrix());
DgClose();

post = DoGroup(DcTrue);
     DgAddObj(DoDiffuseColor(DcRGB, yellow));
                                   /* color it yellow */
     DgAddObj(DoSurfaceShade(DcShaderConstant));
                                   /* constant shading*/
     DgAddObj(DoRepType(DcWireframe));
                                   /* wireframe representation */
     DgAddObj(DoRotate(DcXAxis, 1.57));
                                   /* rotate 90 degrees */
     DgAddObj(DoScale(0.4, 0.4, 2.1));
                                   /* size the object */
     DgAddObj(DoPrimSurf(DcCylinder));
                                   /* make a cylinder */
DgClose();

base = DoGroup(DcTrue);
     DgAddObj(DoTranslate(0.0, -3.0, 0.0));
                                   /* move object down */
     DgAddObj(DoScale(2.0, 2.0, 2.0));
                                   /* make it bigger */
     DgAddObj(DoTranslate(-0.5, -0.5, -0.5));
                                   /*move the object again*/
     DgAddObj(DoPrimSurf(DcBox));  /* make a box */
```

```
    DgClose();

    obj_group = DoGroup(DcTrue);
        DgAddObj(DoRepType(DcSurface));
                                        /* surface representation type */
        DgAddObj(DoSubDivSpec(DcSubDivRelative, sds));
        DgAddObj(DoDiffuseColor(DcRGB, magenta));
                                        /* color it magenta */
        DgAddObj(post);                 /* add the post group */
        DgAddObj(base);                 /* add the base group */
        DgAddObj(DoPrimSurf(DcSphere));
                                        /*make a sphere */
    DgClose();

    DgAddObjToGroup(DvInqDefinitionGroup(view), def_group);
    DgAddObjToGroup(DvInqDisplayGroup(view), obj_group);
    DdUpdate(device);                   /* do the drawing */
    printf("Hit return to continue.0);
    getchar();
    DsReleaseObj(device);               /* clean up */
    DsTerminateSystem();                /* shut down */
}
```

Sample Program in Fortran

```
      PROGRAM MAIN
C
      IMPLICIT NONE
      INCLUDE '/usr/include/DORE'
C
      INTEGER*4 DEVICE, FRAME, VIEW     ! declare variables !
      INTEGER*4 POST, BASE, DEFGRP, OBJGRP
      REAL*8 ORIGIN(3)
      REAL*8 EYEPT(3)
      REAL*8 LIGHT(3)
      REAL*8 UP(3)
      REAL*8 MAGENT(3)
      REAL*8 YELLOW(3)
      REAL*8 SKYBLU(3)
      REAL*8 SDS(1)
      CHARACTER DUMMY
C
      DATA ORIGIN / 3* 0.0D0 /          !x, y, z values !
      DATA EYEPT  / 2* 0.0D0, 10.0D0 /
      DATA LIGHT  / 2* 1.0D0, 0.5D0
      DATA UP     / 0.0D0, 1.0D0, 0.0D0 /
      DATA MAGENT / 1.0D0, 0.0D0, 1.0D0 /
      DATA YELLOW / 2* 0.8D0, 0.8D0 /
      DATA SKYBLU / 0.3D0, 0.3D0, 1.0D0 /
      DATA SDS    / 0.8D0 /
C
      CALL DSINIT (0)                   ! initialize Doré !
      DEVICE = DOD('ardentx11',9,'',0)  ! open dynamic display device !
      FRAME = DOFR()                    ! create a frame !
      CALL DDSF (DEVICE,FRAME)          ! add the frame to the device !
      VIEW = DOVW()                     ! create a view !
```

```
      CALL DVSBC (VIEW, DCRGB, SKYBLU)    ! set the background color !
      CALL DGAOG(DFQVG(FRAME), VIEW)      ! add the view to the frame !
C
      DEFGRP = DOG(DCTRUE)  ! create a group for the camera and light !
           CALL DGAO(DOPAR(10.0D0, -0.1D0, -20.0D0))  ! parallel camera !
           CALL DGAO(DOPUMX())
                CALL DGAO (DOLAF(ORIGIN, EYEPT, UP))  ! positions camera!
                CALL DGAO (DOCM())        ! creates the camera !
           CALL DGAO(DOPPMX())
           CALL DGAO(DOPUMX())
                CALL DGAO(DOLAF(ORIGIN, LIGHT, UP))  ! positions light !
                CALL DGAO(DOLI(1.0D0))    ! light intensity !
                CALL DGAO(DOLT())         ! creates the light !
           CALL DGAO(DOPPMX())
      CALL DGCS()
C
      POST = DOG(DCTRUE)
           CALL DGAO(DODIFC(DCRGB, YELLOW))  ! color it yellow !
           CALL DDGAO(DOSRFS(DCSHCN))     ! constant shading !
           CALL DGAO(DOREPT(DCWIRE))      ! wireframe representation !
           CALL DGAO(DOROT(DCXAX, 1.57D0))  ! rotate on x axis 90 degrees !
           CALL DGAO(DOSC(0.4D0, 0.4D0, 2.1D0))  ! size the object !
           CALL DGAO(DOPMS(DCCYL))        ! make a cylinder !
      CALL DGCS()
C
      BASE = DOG(DCTRUE)
           CALL DGAO(DOXLT(0.0D0, -3.0D0, 0.0D0))  ! move object down !
           CALL DGAO(DOSC(2.0D0, 2.0D0, 2.0D0))  ! make it bigger !
           CALL DGAO(DOXLT(-0.5D0, -0.5D0, -0.5D0))  ! move the object !
           CALL DGAO(DOPMS(DCBOX))        ! make a box !
      CALL DGCS()
C
      OBJGRP = DOG(DCTRUE)
           CALL DGAO(DOREPT(DCSURF))      ! surface representation !
           CALL DGAO(DOSDS(DCSDRL, SDS)) ! relative subdivision !
           CALL DGAO(DODIFC(DCRGB, MAGENT))  ! color it magenta !
           CALL DGAO(POST)                ! add the post group !
           CALL DGAO(BASE)                ! add the base group !
           CALL DGAO(DOPMS(DCSPHR))       ! make a sphere !
      CALL DGCS()
C
      CALL DGAOG(DVQDG(VIEW), DEFGRP)     ! add camera and light to view !
      CALL DGAOG(DVQIG(VIEW),OBJGRP)      ! add displayed objects to view !
      CALL DDU(DEVICE)                    ! do the drawing !
      WRITE(6,'("Hit return to continue.")')
      READ(5, '(A)') DUMMY
      CALL DSRO(DEVICE)                   ! clean up !
      CALL DSTERM                         ! shut down !
C
      END
```

FUNCTIONAL GROUPS

CHAPTER THREE

This chapter explains the naming conventions used for Doré functions, types, and constants. Lists of functional groups are presented, along with a brief description of the functions used in this manual. For a complete description of all Doré functions, see the *Doré Reference Manual.*

Doré Naming Conventions

All Doré functions, types, and constants begin with the letter *D*. The second letter identifies whether the name is a Doré type, a constant, or a particular kind of function, as follows:

Dcxxx
a *constant* value, used as a parameter.

Dtxxx
a *type* declaration, used for parameters.

Ddxxx
a *device* function.

Dexxx
a function called when user *extensions* are added to standard Doré. These functions are not typically used by application programs.

Dfxxx
a *frame* function.

Dgxxx
a *group* function.

Doxxx
an *object creation* function. All *Doxxx* functions return object handles.

Dpxxx
a function used to modify Doré *primitives*.

Dsxxx
a *system* command.

DUoxxx
a user-defined primitive object creation function

Dvxxx
a *view* function.

Functional Groups

The functional groups in the Doré Library are listed below in the following order (see also Figure 3-1):

- Object creation functions (includes primitive objects and their attributes, studio objects and their attributes, and organizational objects). Chapters 4 through 9 discuss these functions in detail. Name sets, filters, and the executability set are discussed in Chapter 11.
- Group functions. Chapter 4 focuses on groups.
- View functions. Chapter 10 discusses the organizational objects— views, frames, and devices.
- Frame functions (see Chapter 10).
- Device functions (see Chapter 10). *DdPickObjs*, a device function is discussed in Chapter 12, "Methods."
- System functions and object manipulation functions. Chapter 13 discusses key system functions as well as the use of valuators to provide input. *DsCompBoundingVol*, a system function, is discussed in Chapter 11, "Conditionals."
- Functions required in creating user extensions to Doré. Chapter 14 describes adding user-defined primitives.

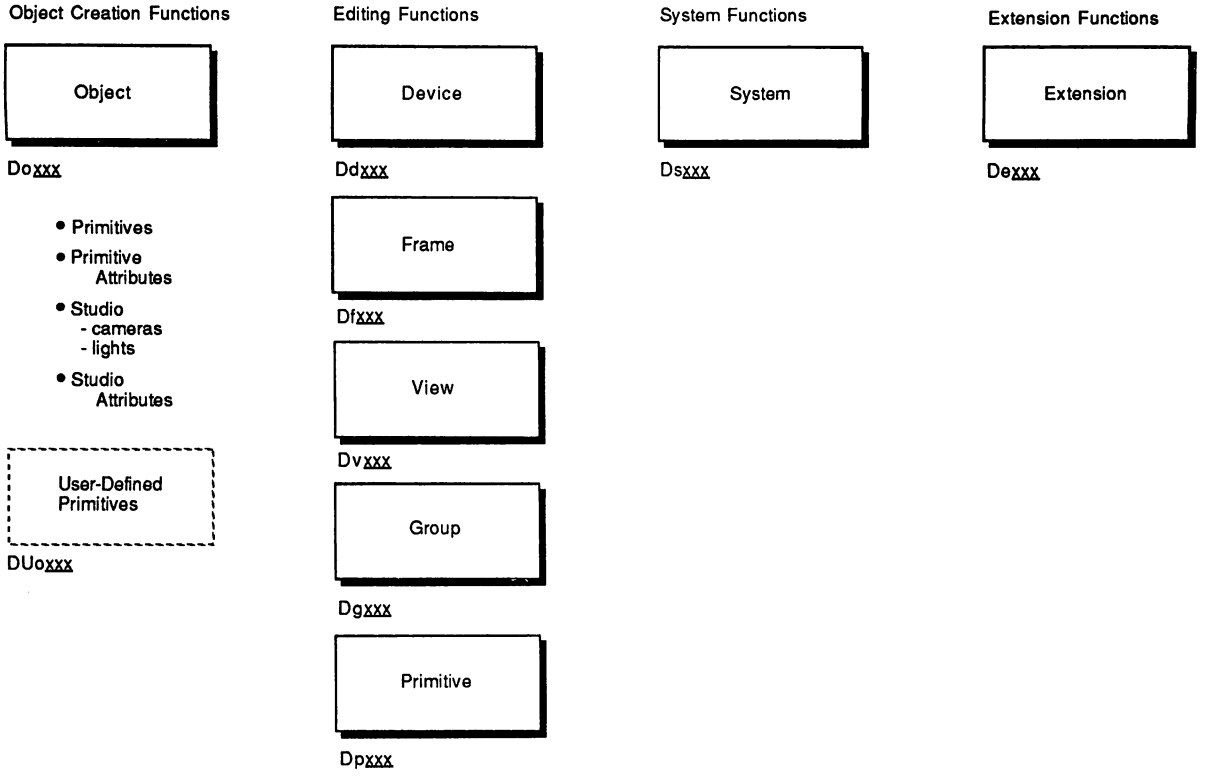


Figure 3-1. Doré Functional Groups

Object Creation Functions

Object creation functions can be divided into the following sub-groups:

- Primitives (including user-defined primitives)
- Primitive Attributes (including text attributes)
- Studio Objects (cameras and lights)
- Studio Object Attributes
- Geometric Transformation Attributes
- Organizational Objects
- Miscellaneous

In the following lists, **boldface** type and brief descriptions denote functions discussed in this manual.

Primitives

DoAnnoText <DOANNT>

DoLineList <DOLINL>
defines a line or series of independent lines.

DoNURBSurf <DONRBS>

DoPatch <DOPAT>

DoPointList <DOPNTL>

DoPolygon <DOPGN>
defines a polygon that may have more than one contour. Contours can be self-intersecting and can intersect each other.

DoPolygonMesh <DOPGNM>

DoPolyline <DOPL>
defines a series of connected line segments.

DoPolymarker <DOPM>

DoPrimSurf <DOPMS>
defines a sphere, cylinder, box, or cone.

DoSimplePolygon <DOSPGN>
defines a simple polygon, which can be convex, concave, or self- intersecting but which has only a single, connected contour.

DoSimplePolygonMesh <DOSPM>

DoText <DOTXT>
creates a text primitive object.

DoTorus <DOTOR>
defines a doughnut-shaped surface.

DoTriangleList <DOTRIL>
defines a triangle or a series of independent triangles.

DoTriangleMesh <DOTRIM>
defines a collection of possibly interconnected triangles.

DoVarLineList <DOVLNL>
defines a variable line list.

DoVarPointList <DOVPTL>
defines a variable point list.

DoVarSimplePolygonMesh <DOVSPM>
defines a variable simple polygon mesh.

DoVarTriangleMesh <DOVTRM>
defines a variable triangle mesh.

DoAmbientIntens <DOAMBI>

DoAmbientSwitch <DOAMBS>
specifies whether the object has an ambient lighting component.

DoBackfaceCullable <DOBFC>
indicates that subsequent primitives form a closed surface, and thus can be backface-culled.

DoBackfaceCullSwitch <DOBFCS>
specifies whether subsequent objects are backface-culled, if possible .

DoBoundingVol <DOBV>
specifies the bounding volume of the following sub-tree.

DoBoundingVolSwitch <DOBVS>
specifies whether to execute subsequent bounding volume objects.

DoClipSwitch <DOCS>

DoClipVol <DOCV>

DoDepthCue <DODC>

DoDepthCueSwitch <DODCS>

DoDiffuseColor <DODIFC>
specifies the diffuse color of subsequent objects.

DoDiffuseIntens <DODIFI>
specifies the diffuse intensity of subsequent objects.

DoDiffuseSwitch <DODIFS>
specifies whether subsequent objects have a diffuse lighting component.

DoExecSet <DOES>
enables or disables execution of objects.

DoFilter <DOFL>
specifies filter membership.

DoGenerateTextureUV <DOGTUV>

DoHiddenSurfSwitch <DOHSS>
specifies whether to perform hidden surface elimination on subsequent primitives. If off, primitive objects are drawn in the exact order given.

DoInterpType <DOIT>
specifies the *interpolation type*, which can be constant, vertex, or surface. Constant interpolation type produces flat-colored surfaces. Vertex interpolation type produces interpolated shading (Gouraud). Surface interpolation type produces pixel-by-pixel shading (Phong).

DoInvisSwitch <DOINVS>
specifies whether subsequent objects are invisible.

DoLightSwitch <DOLTS>
enables/disables illumination of an object from a light.

DoLineType <DOLNT>
specifies the line type for subsequent objects: solid, dashed, dotted, or dot-dash.

DoLineWidth <DOLW>
scales the width of lines for subsequent objects.

DoMarkerFont <DOMF>

DoMarkerGlyph <DOMG>

DoMarkerScale <DOMS>

DoMinBoundingVolExt <DOMBVE>
specifies the smallest renderable bounded object.

DoNameSet <DONS>
modifies the current name set.

DoPickID <DOPID>
specifies an identifier that will be returned on picking.

DoPickSwitch <DOPS>
specifies whether subsequent objects can be picked.

DoReflectionSwitch <DOREFS>
specifies whether subsequent objects reflect the environment.

DoRefractionIndex <DORFRI>

DoRefractionSwitch <DORFRS>

DoRepType <DOREPT>
sets the representation type of subsequent objects to points, wireframe, polygonal outlines, or surfaces.

DoShadeIndex <DOSI>
specifies the shade range to use for subsequent primitive objects.

DoShadowSwitch <DOSHAS>
specifies whether subsequent objects can have shadows cast on them.

DoSpecularColor <DOSPCC>
specifies the specular color of subsequent objects.

DoSpecularFactor <DOSPCF>
specifies the specular spread (the scatter of reflected light).

DoSpecularIntensity <DOSPCI>
specifies the specular intensity.

DoSpecularSwitch <DOSPCS>
specifies whether subsequent objects have a specular lighting component.

DoSubDivSpec <DOSDS>
specifies how objects are to be subdivided.

DoSurfaceShade <DOSRFS>

DoTextureMapBump <DOTMB>

DoTextureMapBumpSwitch <DOTMBS>

DoTextureMapDiffuseColor <DOTMDC>

DoTextureMapDiffuseColorSwitch <DOTMDS>

DoTextureMapEnviron <DOTME>

DoTextureMapEnvironSwitch <DOTMES>

DoTextureMapTranspIntens <DOTMTI>

DoTextureMapTranspIntensSwitch <DOTMTS>

DoTranspColor <DOTC>
specifies the transmitted color.

DoTranspIntens <DOTI>
specifies the intensity of transmitted light.

DoTranspOrientColor <DOTOC>

DoTranspOrientExp <DOTOE>

DoTranspOrientIntens <DOTOI>

DoTranspOrientSwitch <DOTOS>

DoTranspSwitch <DOTS>
specifies whether subsequent objects transmit light.

Text Attributes

DoTextAlign <DOTA>
specifies the horizontal and vertical alignment of text.

DoTextExpFactor <DOTEF>
specifies stretching or compression of characters.

DoTextFont <DOTF>
specifies a text font.

DoTextHeight <DOTH>
specifies the nominal height of a capital letter.

DoTextPath <DOTPA>
specifies the direction of the text string.

DoTextPrecision <DOTPR>

DoTextSpace <DOTSP>
scales the space between characters.

DoTextUpVector <DOTUV>
sets the slant (backbone) of text.

Studio Objects

DoCamera <DOCM>
places a camera in the scene.

DoLight <DOLT>
places a light in the scene.

Texture Attributes

DoTextureAntiAlias <DOTAA>
DoTextureExtendUV <DOTXUV>
DoTextureExtendUVW <DOTXW>
DoTextureOp <DOTOP>
DoTextureScaleUV <DOTSUV>
DoTextureScaleUVW <DOTSW>
DoTextureTranslateUV <DOTTUV>
DoTextureTranslateUVW <DOTTW>
DoTextureUVIndex <DOTUVI>
DoTextureUVWIndex <DOTWI>

Studio Object Attributes

Studio object attributes include subgroups for lights and cameras.

Light Attributes

DoLightAttenuation <DOLTA>
specifies how light intensity falls off with distance.

DoLightColor <DOLC>
specifies the color of subsequent lights.

DoLightIntens <DOLI>
specifies the intensity of subsequent lights.

DoLightSpreadAngles <DOLTSA>
specifies the width of the light beam.

DoLightSpreadExp <DOLSE>
specifies how light intensity falls off from the angle of light direction.

DoLightType <DOLTT>
specifies the type of subsequent lights: ambient, light at infinity, point light (with and without attenuation), and spot light (with and without attenuation).

Camera Attributes

DoGlbRendMaxObjs <DOGRMO>
for ray tracing, specifies the maximum number of objects in a spatial subdivision box.

DoGlbRendMaxSub <DOGRMS>
for ray tracing, specifies the maximum number of binary subdivisions of a spatial subdivision box.

DoGlbRendRayLevel <DOGRRL>
for ray tracing, specifies the maximum number of ray bounces.

DoParallel <DOPAR>
specifies a parallel projection.

DoPerspective <DOPER>
specifies a perspective projection.

DoProjection <DOPRJ>

DoSampleAdaptive <DOSADP>

DoSampleAdaptiveSwitch <DOSASW>

DoSampleFilter <DOSFLT>

DoSampleJitter <DOSJIT>

DoSampleJitterSwitch <DOSJSW>

DoSampleSuper <DOSSPR>

DoSampleSuperSwitch <DOSSSW>

DoStereo <DOSTER>
specifies that subsequent cameras have two viewpoints, one for each eye. The two eye points are derived from the camera location and direction.

DoStereoSwitch <DOSTES>
specifies whether subsequent cameras are stereo cameras

Geometric Transformation Attributes

DoLookAtFrom <DOLAF>
specifies a series of rotations and a translation.

DoPopMatrix <DOPPMX>
returns to a previous geometric context within a group.

DoPushMatrix <DOPUMX>
creates a geometric context within a group.

DoRotate <DOROT>
specifies a rotation about the *x*, *y*, or *z* axis.

DoScale <DOSC>
specifies a scaling along the *x*, *y*, and *z* axes.

DoShear <DOSHR>
specifies a displacement of coordinates in two dimensions proportional to their distance from one of the three major planes.

DoTransformMatrix <DOTMX>

DoTranslate <DOXLT>
specifies a translation of an object along the *x*, *y*, and *z* axes.

Organizational Objects

DoDevice <DOD>
creates a Doré device.

DoFrame <DOFR>
creates a frame.

DoGroup <DOG>
creates a regular group.

DoInLineGroup <DOILG>
creates an in-line group.

DoInputSlot <DOIS>
creates an input slot for tying valuators into operations on the scene database.

DoView <DOVW>
creates a view.

Miscellaneous Objects

DoCallback <DOCB>
creates an object that, upon execution, calls a user written function.

DoCameraMatrix <DOCMX>
creates an arbitrary viewing matrix.

DoDataPtr <DODP>
points to a user-defined data structure that is used by callback objects.

DoDataVal <DODV>
is a data value used by callback objects.

DoFileRaster <DOFRS>

DoLabel <DOLL>
creates a label object for group editing.

DoMatrix <DOM>

DoPopAtts <DOPPA>
restores previous values of attributes, within a group.

DoPushAtts <DOPUA>
saves current attributes, within a group.

DoRaster <DORS>

Primitive Update Functions

DpUpdVarLineList <DPUVLL>
updates a variable line list primitive object.

DpUpdVarPointList <DPUVPL>
updates a variable point list primitive object.

DpUpdVarSimplePolygonMesh <DPUVSM>
updates a variable simple polygon mesh primitive object.

DpUpdVarTriangleMesh <DPUVTM>
updates a variable triangle mesh primitive object.

Group Functions

DgAddObj <DGAO>
adds an object to the currently open group.

DgAddObjToGroup <DGAOG>
adds an object to the specified group.

DgCheck <DGCK>

DgClose <DGCS>
closes the currently open group.

DgDelEle <DGDE>

DgDelEleBetweenLabels <DGDEL>

DgDelEleRange <DGDER>

DgEmpty <DGE>

DgInqElePtr <DGQEP>
returns the current element pointer position.

DgInqObjAtPos <DGQOP>
returns the object at the specified position within a specified group.

DgInqOpen <DGQO>

DgInqSize <DGQS>

DgOpen <DGO>

DgReplaceObj <DGRO>

DgReplaceObjInGroup <DGROG>

DgSetElePtr <DGSEP>
positions the element pointer in the open group.

DgSetElePtrRelLabel <DGSEPL>
positions the element pointer in the open group relative to the specified label.

View Functions

DvInqActiveCamera <DVQAC>

DvInqBackgroundColor <DVQBC>
returns the background color for a view.

DvInqBoundary <DVQB>

DvInqClearFlag <DVQCF>
inquires the clear flag for a view.

DvInqDefinitionGroup <DVQDG>
returns the definition group for a view.

DvInqDisplayGroup <DVQIG>
returns the display group for a view.

DvInqRendStyle <DVQRS>

DvInqShadeIndex <DVQSI>

DvInqUpdateType <DVQUT>

DvSetActiveCamera <DVSAC>
sets the active camera for a view.

DvSetBackgroundColor <DVSBC>

DvSetBoundary <DVSB>
sets the view boundary.

DvSetClearFlag <DVSCF>

DvSetRendStyle <DVSRS>
sets the rendering style for a view: for example, real time or production time.

DvSetShadeIndex <DVSSI>

DvSetUpdateType <DVSUT>

DvUpdate
updates (redraws) a view.

Frame Functions

DfInqBoundary <DFQB>

DfInqJust <DFQJ>

DfInqViewGroup <DFQVG>

DfSetBoundary <DFSB>
sets the frame boundary.

DfSetJust <DFSJ>
sets the position of the frame within the device.

DfUpdate <DFU>
updates (redraws) a frame on all devices to which it is assigned.

Device Functions

DdInqColorEntries <DDQCE>

DdInqColorTableSize <DDQCTS>

DdInqExtent <DDQE>
returns the device extent.

DdInqFonts <DDQFT>

DdInqFrame <DDQFR>

DdInqNumFonts <DDQNF>

DdInqPickAperture <DDQPA>

DdInqPickCallBack <DDQPC>

DdInqPickPathOrder <DDQPPO>

DdInqPixelData <DDQPXD>

DdInqResolution <DDQR>

DdInqShadeMode <DDQSM>

DdInqShadeRanges <DDQSR>

DdInqViewport <DDQV>
returns the device viewport.

DdInqVisualType <DDQVT>

DdPickObjs <DDPO>
initiates a pick on a device.

DdSetColorEntries <DDSCE>
sets the color values of a color lookup table on a pseudocolor device.

DdSetFrame <DDSF>
attaches a frame to a device.

DdSetPickAperture <DDSPA>
sets the pick aperture for a device.

DdSetPickCallBack <DDSPCB>
specifies a callback object for pick filtering on a device.

DdSetPickPathOrder <DDSPPO>
sets pick path ordering.

DdSetShadeMode <DDSSM>
specifies the shade mode of a pseudocolor device (either bit compression or range intensity mapping).

DdSetShadeRanges <DDSSR>
sets one or more shade range table entries on a pseudocolor device.

DdSetViewport <DDSDV>
sets the device viewport.

DdUpdate <DDU>
updates (redraws) the frame attached to the device.

System Functions

DsCompBoundingVol <DSCBV>
computes the bounding volume of a given sub-tree.

DsExecuteObj <DSEO>
executes a specific object; used by callback functions.

DsExecutionAbort <DSEA>
ends database traversal; used by callback functions.

DsExecutionReturn <DSER>
returns from executing the group in which the callback function was found and continues with traversal of the rest of the database.

DsFileRasterRead <DSFRSR>

DsHoldObj <DSHO>
marks an object as held so that it will not be deleted when it is dereferenced.

DsInitializeSystem <DSINIT>
initializes the system (the first Doré call).

DsInputValue <DSIV>
inputs a value (possibly asynchronously) to a slot.

DsInqClassId <DSQCI>

DsInqCurrentMethod <DSQCM>

DsInqErrorMessage <DSQEM>
DsInqErrorVars <DSQEV>
DsInqExeDepthLimit <DSQEDL>
DsInqHoldObj <DSQHO>
DsInqMethodId <DSQMI>
DsInqNumRenderers <DSQNR>
DsInqObj <DSQOI, DSQOS>
DsInqObjName <DSQONT; DSQONI, DSQONS>
DsInqObjStatus <DSQVOS>
DsInqObjType <DSQOT>
DsInqRendererId <DSQRI>
DsInqRendererNames <DSQRNS>
DsInqSafeFlag <DSQSF>

DsInqValuatorGroup <DSQVG>
returns the valuator group for an input slot.

DsPrintObj <DSPO>
DsRasterUpdate <DSRSU>
DsRasterWrite <DSRSW>

DsReleaseObj <DSRO>
informs the system that an object is no longer needed.

DsSetErrorVars <DSSEV>
DsSetExeDepthLimit <DSSEDL>
DsSetObjName <DSSONI, DSSOND, DSSONS>
DsSetSafeFlag <DSSSF>

DsTerminateSystem <DSTERM>
terminates the system (the last Doré call).

DsTextureUVCount <DSTUVC>
DsTextureUVWCount <DSTWC>

DsUpdateAllViews <DSUAV>
updates all views on all frames on all devices.

DsValuatorSwitch <DSVS>

Doré Extension Functions

DeAddClass <DEAC>
adds a new class to Doré and returns a unique class identification number.

DeCreateObject <DECO>
creates an internal Doré object.

DeDeleteObject <DEDO>
deletes a Doré object.

<DEDOD> (Fortran only)
deallocates space used by the private data of an object.

<DEROD> (Fortran only)
reads from the private data of an object.

<DEWOD> (Fortran only)
writes to the private data of an object.

DeExecute Alternate <DEEA>
executes the current method on an alternate object for a user-defined primitive.

DeInitializeObjPick <DEIOP>
initializes picking for an object.

DeInqPickable <DEQP>
returns whether a class is pickable.

DeInqRenderable <DEQR>
returns whether a class is renderable.

OBJECTS AND GROUPS

CHAPTER FOUR

This chapter defines what Doré objects are and describes the object-based nature of Doré. It also shows how to create objects and how to create hierarchical models using groups. Sample code shows two front car wheels combined to form an axle group with wheels that turn in unison. Concepts and terms introduced in this chapter include objects, object handles, group hierarchies, regular and in-line groups, group editing, labels, and element pointers.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DgAddObj <DGAO>
DgAddObjToGroup <DGAOG>
DgCheck <DGCK>
DgClose <DGCS>
DgDelEle <DGDE>
DgDelEleBetweenLabels <DGDEL>
DgDelEleRange <DGDER>
DgEmpty <DGE>
DgInqElePtr <DGQEP>
DgInqObjAtPos <DGQOP>
DgInqSize <DGQS>
DgInqOpen <DGQO>
DgOpen <DGO>
DgReplaceObj <DGRO>
DgReplaceObjInGroup <DGROG>
DgSetElePtr <DGSEP>
DgSetElePtrRelLabel <DGSEPL>
DoGroup <DOG>
DoInLineGroup <DOILG>
DoLabel <DOLL>
DsHoldObj <DSHO>
DsInqClassId <DSQCI>
DsInqObj <DSQOI, DSQOS>

Related Functions
(continued)

DsInqObjName <DSQONT>
DsInqObjType <DSQONI, DSQONS>
DsInqObjStatus <DSQVOS>
DsReleaseObj <DSRO>
DsSetObjName <DSSOND, DSSONI, DSSONS>

What Is an Object?

Objects are the basic building blocks of the Doré Library. An *object* is a collection of data and a set of methods (functions) that operate on the data. Certain kinds of objects, such as *primitive objects*, represent three-dimensional shapes (cylinders, spheres, toruses, polygons) that can be displayed. Other objects, called *primitive attribute objects*, affect how primitive objects look—for example, their coloring, how they respond to light, whether they can have shadows, and so on. *Geometric transformation objects* are a type of attribute object that affect the shape and positioning of primitive objects in three-dimensional space.

Primitive objects and primitive attribute objects fall into a general category called *display objects*. As their name suggests, display objects expect to be rendered and displayed. If certain conditions are met (they are not invisible, they are not obscured by other objects), you will eventually be able to view the primitive objects in some form, and you will be able to notice the effects of their associated attribute objects.

Object Creation Functions

All Doré functions beginning with the prefix *Do-* **are** *object creation* functions. When a *Do-* function is called, Doré allocates a block of memory and copies the information passed to the function into a region of the memory reserved for that object's data. Doré objects fall into the following general categories:

- primitive objects
- studio objects
- attribute objects
- geometric transformation objects
- organizational objects

A separate chapter of this manual is devoted to each of these categories.

Object Handles

All *Do-* functions return a value of type *DtObject* (in Fortran, an INTEGER*4) that is known as the *handle* to the newly created object. This handle is a unique number that represents a particular object and can be used in further references to it. **You do not have direct access to the information stored in a Doré object.** Instead, you refer to the object by its handle, using function calls such as *DgAddObjToGroup* <*DGAOG*> or *DgReplaceObjInGroup* <*DGROG*> which take object handles as arguments. In this sense, an object handle and the object data are analogous to a bank account number and the money you put into that account. You do not know exactly where your money is stored, and you cannot handle the money directly, but you can use your account number to obtain information about your account, or to retrieve some of the money when you need it.

Structure of an Object

As shown in Figure 4-1, in addition to its data, each object contains

- a type
- a reference count
- a hold flag
- an optional name

The *type* is an integer constant defined for each type of Doré object. The type can be found by calling *DsInqClassId* <*DSQCI*>). This type field determines the format of the information stored in the data field. The *reference count* and *hold flag* are used for Doré memory management (see the section below on "Holding and Releasing Objects"). The *name* is an optional field provided as a programming convenience, which allows you to assign a string or integer name to the object.

Figure 4-2 shows a sample object, a triangle mesh. The handle to this object has been saved in the user variable *my_mesh*. Code to create the object would follow the general form:

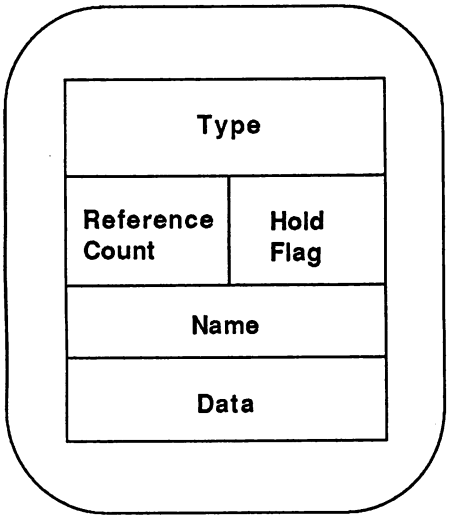


Figure 4-1. Object Diagram

```
C code:

DtObject my_mesh;
my_mesh = DoTriangleMesh(...);


FORTRAN code:

INTEGER*4 MY_MESH
MY_MESH = DOTRIM(...)
```

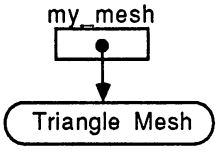


Figure 4-2. Sample Object

Object Diagrams

Object diagrams are very helpful in organizing a Doré application or in trying to understand someone else's Doré code. Given an object diagram, it is simple to write the equivalent Doré code. And given some Doré code, it is simple to generate the diagram.

The figures in this manual use an oval shape to indicate objects. Anything written inside the oval represents information stored in that object. Although all objects have all the fields shown in Figure 4-1, only fields relevant to the current discussion will be included in a given diagram.

Object handles are represented by boxes with arrows pointing to the objects. In Figure 4-2, the object handle is a variable of type *DtObject* (in Fortran, an INTEGER*4). You must immediately save or use the object handle returned from an object creation function. Otherwise, you have no way to access the object, which resides in Doré's private database.

Groups

A *group* is a Doré object that contains an ordered list of object handles. A group itself is also an object. Once you have defined an object or a group of objects, you can refer to it any number of times in a particular scene database. For example, you could model a bolt shape using a group containing polygons and a cylinder and then use that bolt a number of times in a single car wheel group. Then you would be able to reference the basic wheel group four times in a complete car object.

The figures in this manual also use the oval shape to represent groups, since groups are a type of object. Within the group diagram, a small triangle is used for the element pointer. (The element pointer is omitted from the figure if it has no bearing on the discussion.)

Example of a Simple Group

Suppose you want to create a magenta sphere with a surface representation type. Figure 4-3 shows a simplified diagram of the objects included in this group. **Note that attribute objects** *precede* **their associated primitive objects, as shown.** The ordering of objects within a group is important because the objects within a group are applied in order —first, the surface object, then the magenta object, and finally the primitive object (in this case, a sphere).

The code to create this object is shown below:

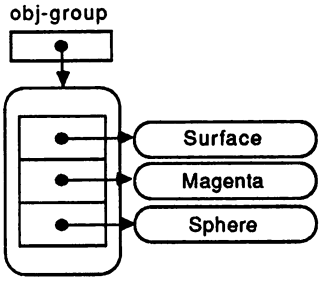


Figure 4-3. Sphere Group

C code:

```
DtObject magenta_obj, surf_obj, sphere_obj, obj_group;
static DtReal magenta[3]={1.0, 0.0, 1.0};

surf_obj=DoRepType(DcSurface);
magenta_obj=DoDiffuseColor(DcRGB, magenta);
sphere_obj=DoPrimSurf(DcSphere);
obj_group=DoGroup(DcFalse);

DgAddObjToGroup(obj_group, surf_obj);
DgAddObjToGroup(obj_group, magenta_obj);
DgAddObjToGroup(obj_group, sphere_obj);
```

FORTRAN code:

```
      INTEGER*4 SURF, MAG, SPHR, OBJGR
C
      DATA MAGENT / 1.0D0, 0.0D0, 1.0D0 /
C
      SURF=DOREPT(DCSURF)
      MAG=DODIFC(DCRGB, MAGENT)
      SPHR=DOPMS(DCSPHR)
      OBJGR=DOG(DCFALS)
C
      CALL DGAOG(OBJGR, SURF)
      CALL DGAOG(OBJGR, MAG)
      CALL DGAOG(OBJGR, SPHR)
```

Creating and Storing the Doré Database

"obj_group" contains objects which together will generate a magenta ball with surface representation type when rendered (other choices for representation type are points, polygonal outlines, or wireframe). The code shown above merely creates a description of this magenta ball, which is stored in the Doré database. **Nothing is actually drawn at this time;** we are simply constructing a database for later use. When one of the update functions is called (*DdUpdate* <DDU>, *DfUpdate* <DFU>, or *DvUpdate*

<*DDU*>, part of the stored database is traversed, and the render method is applied to each object in the database. At this time, the visible primitives will be displayed. As described in Chapter 12, "Methods," different methods operate on the Doré database in different ways.

Construct Your Own Primitive

If Doré does not have a primitive with exactly the geometry and qualities you want, you can construct your own group object containing a set of simpler primitives and attribute objects and use it as you would a primitive. "obj_group" can be used exactly as if Doré had a primitive that was a magenta, surface rep-type ball. **Once you have defined an object, you can instance it multiple times.** For example, the magenta ball group might be a useful "primitive" to represent one type of atom making up a complex molecule. Whenever that atom type is needed, you can simply add the *obj_group* handle to the molecule group.

A Shorthand Method for Coding

For purposes of introduction, the previous example used the most explicit method of creating objects: naming their handles and adding the named handles to a named group (obj_group). There are several other ways to accomplish the same end, which are described below.

The Currently Open Group

As we have seen, certain group editing functions, such as *DgAddObjToGroup* <*DGAOG*>, accept a group handle and operate on that group. Several other group functions, however, do not take a group parameter (for example, *DgAddObj* <*DGAO*>, *DgDelEle* <*DGDE*>, and *DgReplaceObj* <*DGRO*>). These functions operate on the *currently open group*. A group can be explicitly specified as the currently open group with the *DgOpen* <*DGO*> function. Or a group can be opened by specifying *DoGroup(DcTrue)* <*DOG(DCTRUE)*>, where *DcTrue* indicates that the group should be made open when it is created. *DgClose* <*DGCS*> closes the currently open group and returns its object handle. The *DgInqOpen* <*DGQO*> function returns the handle for the currently open group. Open groups nest. That is, if one group is already open in the Doré database and another group is opened as well, the second group becomes the currently open group until it is closed, at which time the first group once again becomes the currently open group.

The open group is a programming convenience, since it eliminates the need to keep renaming the same group as you add objects to it. A shorter way to create the ball group uses the open group:

```
C code:

DtObject magenta_obj, surf_obj, sphere_obj, obj_group;
static DtReal magenta[3]={1.0, 0.0, 1.0};

surf_obj=DoRepType(DcSurface);
magenta_obj=DoDiffuseColor(DcRGB, magenta);
sphere_obj=DoPrimSurf(DcSphere);

obj_group=DoGroup(DcTrue);
     DgAddObj(surf_obj);
     DgAddObj(magenta_obj);
     DgAddObj(sphere_obj);
DgClose();


FORTRAN code:

       INTEGER*4 SURF, MAG, SPHR
       REAL*8 MAGENT(3)
C
       DATA MAGENT / 1.0D0, 0.0D0, 1.0D0 /
C
       SURF=DOREPT(DCSURF)
       MAG=DODIFC(DCRGB, MAGENT)
       SPHR=DOPMS(DCSPHR)
C
       OBJGR=DOG(DCTRUE)
          CALL DGAO(SURF)
          CALL DGAO(MAG)
          CALL DGAO(SPHR)
       CALL DGCS
```

To further simplify coding, the objects themselves can be created and added on-the-fly. Object handles do not need to be stored in variables, since they are immediately added to obj_group. This is the most commonly used style.

```
C code:

obj_group=DoGroup(DcTrue);
     DgAddObj(DoRepType(DcSurface));
     DgAddObj(DoDiffuseColor(DcRGB, magenta));
     DgAddObj(DoPrimSurf(DcSphere));
DgClose();


FORTRAN code:

OBJ_GROUP=DOG(DCTRUE)
```

```
     CALL DGAO (DOREPT (DCSURF))
     CALL DGAO (DODIFC (DCRGB, MAGENT))
     CALL DGAO (DOPMS (DCSPHR))
CALL DGCS()
```

Editing a Group

It is possible to *edit*, by use of function calls, the information stored in the data fields of certain types of objects. Editable objects include groups, views, frames, and devices, which fall into the category of "organizational objects." Views, frames, and devices have special functions that allow them to be edited, just as groups can be edited. In contrast to these organizational objects, most objects cannot be edited; an object can, however, be replaced by another object with the desired parameters. (See also the section below, "Holding and Releasing Objects.")

Group editing functions form another large group of Doré functions. The *Dg-* functions are used for

- opening and closing groups
- adding, deleting, and replacing objects in groups
- positioning the element pointer in a group
- returning information about a group

Groups Within Groups

A group can also contain other groups. For example, the obj_group could contain a post group, as shown in Figure 4-4. The post group has its own color (yellow) and representation type (wireframe), plus other attributes (constant shading, rotation, and so on). Other groups can be added to obj_group as well, as shown in Figure 4-5.

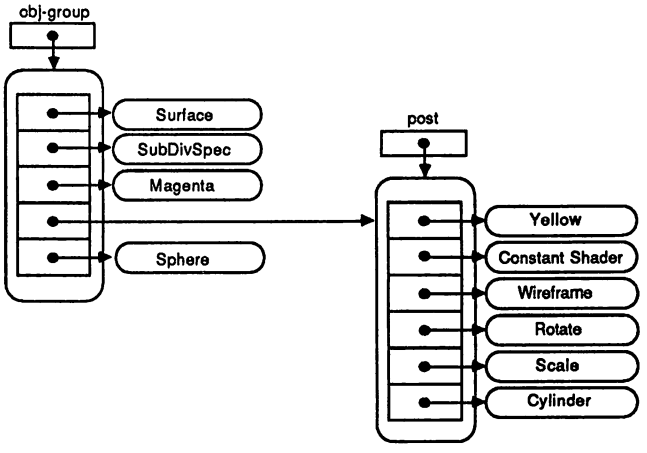


Figure 4-4. Adding the Post Group

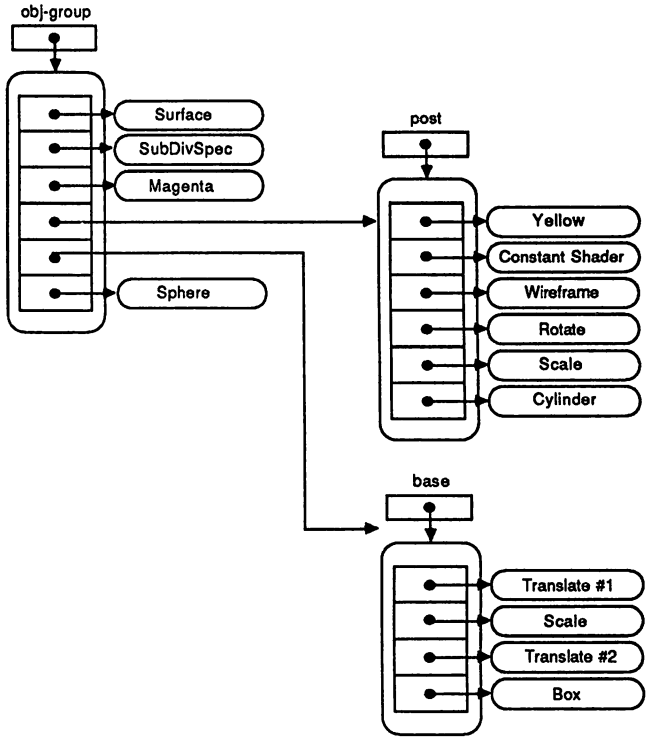


Figure 4-5. Adding the Base Group

The code for these three groups is shown below.

C code:

```
#define DTOR(angle)((angle)*(3.14159265358979/180.0))/*degrees to radians*/

main()
{
     DtObject post, base, obj_group;

     static DtReal
          magenta[] = {1.0, 0.0, 1.0},
          yellow[] = {0.8, 0.8, 0.0},
          sds[] = {0.8};

     post = DoGroup(DcTrue);
          DgAddObj(DoDiffuseColor(DcRGB, yellow));  /* color it yellow */
          DgAddObj(DoSurfaceShade(DcShaderConstant));  /* constant shading */
          DgAddObj(DoRepType(DcWireframe));       /* wireframe representation */
          DgAddObj(DoRotate(DcXAxis, DTOR(90))); /* rotate 90 degrees */
          DgAddObj(DoScale(0.4, 0.4, 2.1));       /* size the object */
          DgAddObj(DoPrimSurf(DcCylinder));       /* make a cylinder */
     DgClose();

     base = DoGroup(DcTrue);
          DgAddObj(DoTranslate(0.0, -3.0, 0.0)); /* move object down */
          DgAddObj(DoScale(2.0, 2.0, 2.0));       /* make it bigger */
          DgAddObj(DoTranslate(-0.5, -0.5, -0.5)); /* move the object */
          DgAddObj(DoPrimSurf(DcBox));              /* make a box */
     DgClose();

     obj_group = DoGroup(DcTrue);
          DgAddObj(DoRepType(DcSurface));        /* surface representation */
          DgAddObj(DoSubDivSpec(DcSubDivRelative, sds)); /* relative subdiv */
          DgAddObj(DoDiffuseColor(DcRGB, magenta)); /* color it magenta */
          DgAddObj(post);                        /* add the post group */
          DgAddObj(base);                        /* add the base group */
          DgAddObj(DoPrimSurf(DcSphere));        /* make a sphere */
     DgClose();
```

Fortran code:

```
      INTEGER*4 POST, BASE, OBJGRP
      INTEGER*8 MAGENT, YELLOW, SDS
      PARAMETER (DEGRAD=0.0174533D0)  ! converts degrees to radians !
      DATA MAGENT / 1.0D0, 0.0D0, 1.0D0 /
      DATA YELLOW / 2* 0.8D0, 0.8D0 /
      DATA SDS    / 0.8D0 /
C
     POST = DOG(DCTRUE)
          CALL DGAO(DODIFC(DCRGB, YELLOW))    ! color it yellow !
          CALL DDGAO(DOSRFS(DCSHCN))          ! constant shading !
          CALL DGAO(DOREPT(DCWIRE))           ! wireframe representation !
          CALL DGAO(DOROT(DCXAX, (DEGRAD*90.0D0))) ! rotate 90 degrees !
          CALL DGAO(DOSC(0.4D0, 0.4D0, 2.1D0)) ! size the object !
          CALL DGAO(DOPMS(DCCYL))             ! make a cylinder !
     CALL DGCS()
```

```
C
      BASE = DOG(DCTRUE)
           CALL DGAO(DOXLT(0.0D0, -3.0D0, 0.0D0))   ! move object down !
           CALL DGAO(DOSC(2.0D0, 2.0D0, 2.0D0))     ! make it bigger !
           CALL DGAO(DOXLT(-0.5D0, -0.5D0, -0.5D0))! move the object !
           CALL DGAO(DOPMS(DCBOX))                  ! make a box !
      CALL DGCS()
C
      OBJGRP = DOG(DCTRUE)
           CALL DGAO(DOREPT(DCSURF))           ! surface representation !
           CALL DGAO(DOSDS(DCSDRL, SDS))       ! relative subdivision !
           CALL DGAO(DODIFC(DCRGB, MAGENT))    ! color it magenta !
           CALL DGAO(POST)                     ! add the post group !
           CALL DGAO(BASE)                     ! add the base group !
           CALL DGAO(DOPMS(DCSPHR))            ! make a sphere !
      CALL DGCS()
```

Nesting Open Groups

An alternative to the preceding example would be to nest the child groups within the parent group. Code would be as follows:

C code:

```
obj_group=DoGroup(DcTrue);
     DgAddObj(DoRepType(DcSurface));
     DgAddObj(DoDiffuseColor(DcRGB, magenta));
     DoGroup(DcTrue);   /*open the post group on-the-fly*/
          DgAddObj(DoDiffuseColor(DcRGB, yellow));
          DgAddObj(DoSurfaceShade(DcShaderConstant));
          DgAddObj(DoRepType(DcWireframe));
          DgAddObj(DoRotate(DcXAxis, DTOR(90)));
          DgAddObj(DoScale(0.4, 0.4, 2.1));
          DgAddObj(DoPrimSurf(DcCylinder));
     DgAddObj(DgClose()); /*closes the post group */
     DoGroup(DcTrue); /* opens the base group on-the-fly*/
          DgAddObj(DoTranslate(0.0, -3.0, 0.0));
          DgAddObj(DoScale(2.0, 2.0, 2.0));
          DgAddObj(DoTranslate(-0.5, -0.5, -0.5));
          DgAddObj(DoPrimSurf(DcBox));
     DgAddObj(DgClose());  /* closes the base group */
     DgAddObj(DoPrimSurf(DcSphere));
DgClose();
```

Fortran code:

```
      OBJGRP=DOG(DCTRUE)
         CALL DGAO(DOREPT(DCSURF))
         CALL DGAO(DODIFC(DCRGB,MAGENT))
         CALL DOG(DCTRUE)      ! opens the post group !
            CALL DGAO(DCDIFC(DCRGB,YELLOW))
            CALL DGAO(DOSRFS(DCSHCN))
            CALL DGAO(DOREPT(DCWIRE))
```

```
            CALL DGAO(DOROT(DCXAX, (DEGRAD*90.0D0)))
            CALL DGAO(DOSC(0.4D0, 0.4D0, 2.1D0))
            CALL DGAO(DOPMS(DCCYL))
         CALL DGAO(DGCS()) ! closes the post group !
C
         CALL DOG(DCTRUE)      ! opens the base group !
            CALL DGAO(DOXLT(0.0D0, -3.0D0, 0.0D0))
            CALL DGAO(DOSC(2.0D0, 2.0D0, 2.0D0))
            CALL DGAO(DOXLT(-0.5D0, -0.5D0, -0.5D0))
            CALL DGAO(DOPMS(DCBOX))
         CALL DGAO(DGCS()) ! closes the base group !
         CALL DGAO(DOPMS(DCSPHR))
      CALL DGCS()
```

Note that the statement *DgAddObj(DoGroup(DcTrue))* is *not* allowed, since it adds the group to itself.

Local Effects of Attributes

An attribute object within a group applies to everything below that point in the group. The diffuse color is set in obj_group to magenta. **"Child" groups such as the post and base groups inherit the attributes of their "parent" group.** The base group is magenta, because it inherits this color from the parent group (obj_group). The post group is different. It changes the diffuse color to yellow. This attribute applies to the following objects in the post group, but not to the following objects in obj_group (because it is a parent group). Chapter 6 contains a detailed description of attribute stacking.

Three general principles apply to primitive attributes within groups used for modeling.

(1) Attributes specified within groups apply only to the objects below that point in the group. (This principle is true only for regular groups, which are the only type we have discussed so far.)

(2) *Child* groups inherit the attributes of their *parent* groups.

(3) Attributes specified closest to a primitive bind most tightly. (This applies only to attributes specified *above* a primitive.)

Note that the post and base groups both inherit the magenta diffuse color, but that the yellow object in the post group overrides that color for the cylinder object because it is specified closer to it.

Creating Objects and Groups

This section shows how to create a group of objects to form a simple car wheel object. Next, it shows how to create right and left wheel groups that include this basic wheel group.

Basic Wheel Group

Figure 4-6 shows the basic wheel group, which is constructed using a torus for the tire and a short cylinder for the hub. Geometric transformation objects are also needed to scale and position the primitives relative to each other. Only those attributes common to all wheels are included in the basic wheel group. Other attributes, like color, are allowed to be inherited.

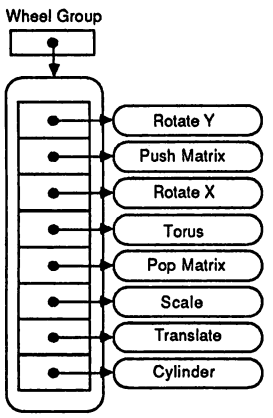


Figure 4-6. Basic Wheel Group

Left and Right Wheel Groups

The left and right car wheels can now easily be created using this "wheel primitive" group, with modifications for the placement and rotation of each wheel. In Example 1 below, both left and right wheel groups include a different diffuse color attribute object. The right and left wheels are translated 4 units apart. (See Figure 4-7.)

The left wheel group contains one sub-group, the basic wheel group. The right wheel group, also a "parent" group, contains

the same "child" group. The same wheel object might also be used elsewhere in the scene, perhaps as rear wheels. **The primitive objects in the child group inherit the attributes of their parent group (or groups),** in this case, the left and right wheel groups.

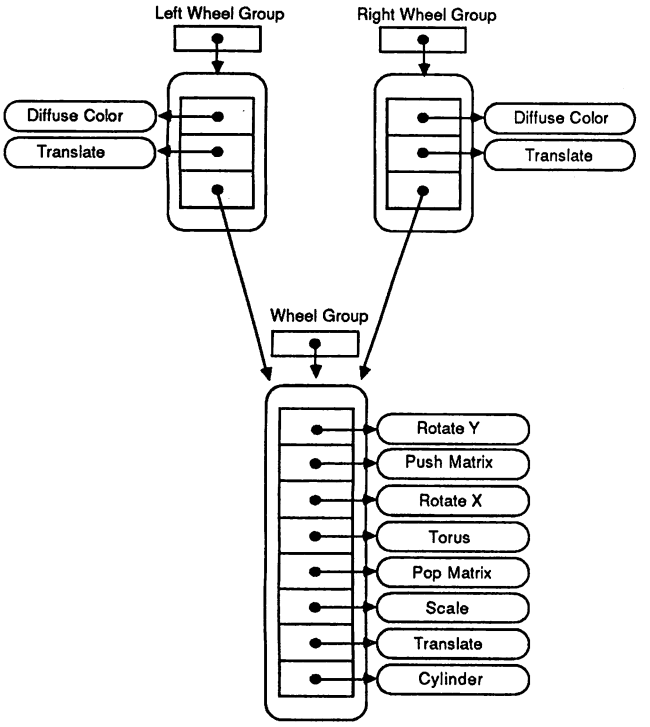


Figure 4-7. Left and Right Wheel Groups

Example 1: Group Structure and Multiple Instances

The code for these groups is shown below in Example 1. The wheel group itself is created with the *DoGroup* <*DOG*> function. The parameter *DcTrue* specifies that the group is open. The group handle has been assigned to the variable *wheel_group* which is then used twice. In the following examples, the DTOR macro can take a floating point or an integer value, but the result is always a floating point value, in radians, which is what *DoRotate* <*DOROT*> requires.

Example 1: Group Structure and Multiple Instances
(continued)

C code:

```
#define DTOR(alpha) (alpha*.0174533) /* converts degrees to radians */

DtObject wheel_group;     /* defines a basic wheel */
DtObject left_wheel;      /* defines the left wheel */
DtObject right_wheel;     /* defines the right wheel */

static DtReal
     blue[] = {0.0, 0.0, 1.0},
     purple[] = {0.8, 0.0, 0.8};

wheel_group = DoGroup(DcTrue);     /* creates and opens the wheel group */
     DgAddObj(DoRotate(DcYAxis, DTOR(90)));  /* rotates the wheel */
     DgAddObj(DoPushMatrix());
               DgAddObj(DoRotate(DcXAxis, DTOR(90))); /* rotates the torus */
               DgAddObj(DoTorus(1.0, 0.3));  /* makes the torus */
     DgAddObj(DoPopMatrix());
     DgAddObj(DoScale (1.0, 1.0, 0.4));       /* scales the cylinder */
     DgAddObj(DoTranslate (0.0, 0.0, -0.5)); /* moves the cylinder */
     DgAddObj(DoPrimSurf (DcCylinder));       /* makes the cylinder */
DgClose();

left_wheel = DoGroup(DcTrue);       /* creates and opens left wheel group */
     DgAddObj(DoDiffuseColor(DcRGB, blue));
     DgAddObj(DoTranslate (-2.0, 0.0, 0.0));
     DgAddObj(wheel_group);    /* references the basic wheel shape */
DgClose();

right_wheel = DoGroup(DcTrue);    /* creates and opens right wheel group */
     DgAddObj(DoDiffuseColor(DcRGB, purple));
     DgAddObj(DoTranslate (2.0, 0.0, 0.0));
     DgAddObj(wheel_group);    /* references the basic wheel shape */
DgClose();
```

Fortran code:

```
      INTEGER*4 WHEEL, LFTWH, RTWH !defines basic wheel, left and right wheels !
      REAL*8 BLUE(3), PURPLE(3)
      PARAMETER (DEGRAD=0.0174533D0) ! converts degrees to radians !
C
      DATA BLUE / 0.0D0, 0.0D0, 1.0D0 /
      DATA PURPLE / 0.8D0, 0.0D0, 0.8D0 /
C
      WHEEL=DOG(DCTRUE) ! creates and opens wheel group !
         CALL DGAO(DOROT(DCYAX, (DEGRAD*90.0D0))) ! rotates the wheel !
         CALL DGAO(DOPUMX())
            CALL DGAO(DOROT(DCXAX, (DEGRAD*90.0D0))) ! rotates the torus !
            CALL DGAO(DOTOR(1.0D0, 0.3D0))
         CALL DGAO(DOPPMX())
         CALL DGAO(DOSC(1.0D0, 1.0D0, 0.4D0)) ! scales the cylinder !
         CALL DGAO(DOXLT(0.0D0, 0.0D0, -0.5D0)) ! moves the cylinder !
         CALL DGAO(DOPMS(DCCYL)) ! makes the cylinder !
      CALL DGCS()
C
```

Example 1: Group Structure and Multiple Instances (continued)

```
      LFTWH=DOG(DCTRUE) ! creates and opens left wheel group !
         CALL DGAO(DODIFC(DCRGB,BLUE)) ! colors it blue !
         CALL DGAO(DOXLT(-2.0D0, 0.0D0, 0.0D0))
         CALL DGAO(WHEEL) ! references the basic wheel shape !
      CALL DGCS()
C
      RTWH=DOG(DCTRUE) ! creates and opens right wheel group !
         CALL DGAO(DODIFC(DCRGB,PURPLE)) ! colors it purple !
         CALL DGAO(DOXLT(2.0D0, 0.0D0, 0.0D0)) ! moves the wheel !
         CALL DGAO(WHEEL) ! references the basic wheel shape !
      CALL DGCS()
```

Dynamics

As described above, rendering of a scene is triggered by one of the update functions (*DdUpdate* <DDU>, *DfUpdate* <DFU>, or *DvUpdate* <DVU>). To create a dynamic sequence of images, you alternately render the scene and then make small changes, or "edits" to the stored database. For example, to make the wheel rotate, you would render the wheel in its original position, then change the rotate object affecting that wheel, then render the wheel again. A series of alternating edits to the rotate object followed by update calls gives the appearance of a moving wheel, in the same way that early cartoonists made animated figures move across the screen by flipping through pages of images with small consecutive changes.

In-line Groups

To make the wheels turn together, we could put identical rotate objects above *each* instance of wheel_group, but we would like to build into the axle definition the fact that the left and right wheels move in unison; that is, we want to move both wheels with *one* edit. One way to do that is to dynamically change the rotate object at the top of the wheel group. The problem with that approach is that it changes the definition of what a wheel is, thereby affecting *all* wheels in the database.

The correct solution involves the use of an *in-line group*. Up to this point, our discussion of how groups behave actually has covered only one type of group, the *regular* group. The other type of group, the *in-line* group, behaves differently with respect to attribute stacking. **An in-line group acts as if its elements were actually part of its parent's group. Attribute values set in an in-line group affect subsequent objects in the parent group.** A useful way to think about the two types of groups is that a regular group is like a subroutine, whereas an in-line group is like a macro. Usually in-line groups contain only attribute objects.

Example 2: In-Line Group

Example 2 shows an in-line group that is referenced by both the left and right wheel groups. The in-line group contains a rotation object, which causes the wheels to move as a pair, and a scale object, which widens and sizes the wheels. An axle group has been added to the example to connect the two wheels. Figure 4-8 shows a diagram of these groups and objects, with the new axle group and the in-line rotation/scale group. The complete black and white image is shown in Figure 4-9.

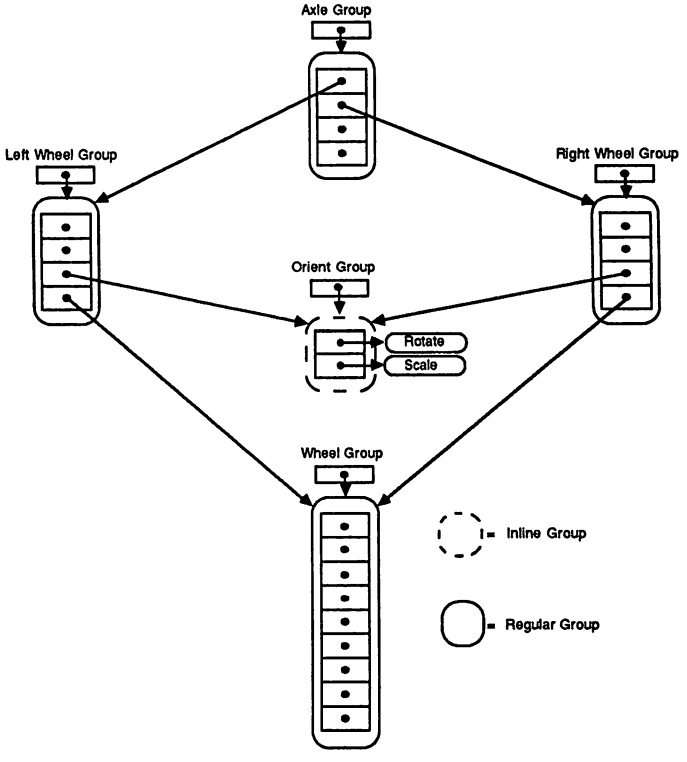


Figure 4-8. Using an In-line Group for the Scale and Rotation Objects

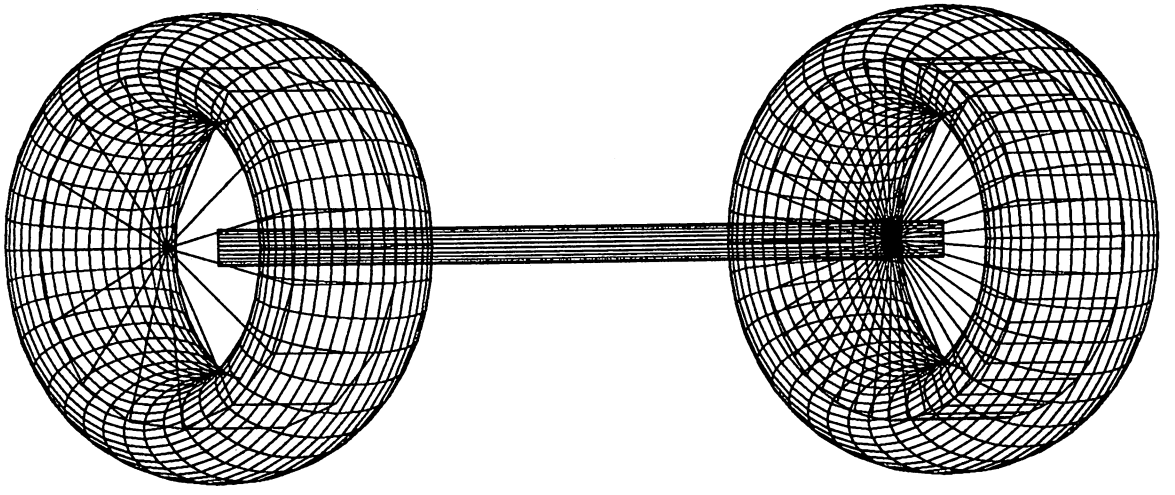


Figure 4-9. Wireframe Wheels and Axle

Element Pointer

Examples 2 and 3 make use of the element pointer. **The *element pointer* is a cursor that specifies the current editing position within each group.** Group elements are not internally numbered; however, the imaginary spaces between elements are. When a group is created, its element pointer is pointing to the space before the first object, space 0. Elements are numbered by their preceding space, so element 0 is actually the first element in a group. Each time an element is added to a group using *DgAddObj* <*DGAO*> or *DgAddObjToGroup* <*DGAOG*>, it is inserted into the space that the element pointer points to and the element pointer is automatically moved down one space. (See Figure 4-10.)

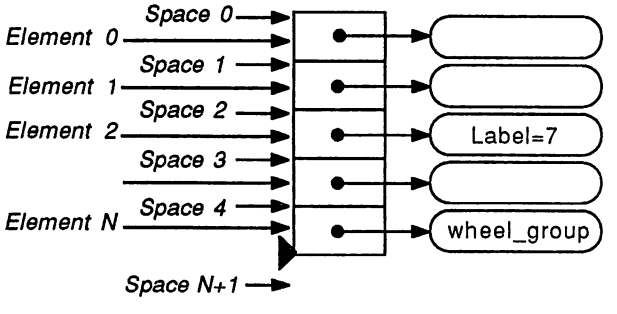


Figure 4-10. Numbering Element Pointer Spaces and Group Elements

Example 2: In-Line Group
(continued)

DgInqElePtr <*DGQEP*> returns the current element pointer location within the currently open group. The second parameter to *DgOpen* <*DGO*> is the "append" flag, which specifies whether the element pointer should remain where it was when the group was closed (*DcFalse*) or should be set to the end of the group so that new objects will be appended there (*DcTrue*). Use *DgSetElePtr* <*DGSEP*> and *DgSetElePtrRelLabel* <*DGSEPL*> to position the element pointer at other locations in the group. (See also "Example 3," below.)

Making the Wheels Move

After you have created the groups, you add them to a view. This process is described in Chapter 10. The loop at the end of Example 2 changes the angle of the wheels in rapid succession so that they appear to turn on their axle.

```
C code:

#define DTOR(alpha) (alpha*.0174533);

DtObject axle_group;      /* defines the axle */
DtObject wheel_group;     /* defines a basic wheel */
DtObject left_wheel;      /* defines the left wheel */
DtObject right_wheel;     /* defines the right wheel */
DtObject orient_group;    /* the in-line group */
DtInt    i;
static DtReal green[] = {0.0, 1.0, 0.0};
static DtReal parms1[] = {.03}, parms2[] = {.02};

wheel_group = DoGroup(DcTrue);      /* creates and opens wheel group */
     DgAddObj(DoDiffuseColor(DcRGB, green));
     DgAddObj(DoPushMatrix());
               DgAddObj(DoRotate(DcXAxis, DTOR(90)));
               DgAddObj(DoTorus(1.0, 0.3));  /* makes the torus */
     DgAddObj(DoPopMatrix());
     DgAddObj(DoScale(1.0, 1.0, 0.4));
     DgAddObj(DoTranslate(0.0, 0.0, -0.5));
     DgAddObj(DoPrimSurf(DcCylinder)); /* makes a cylinder */
DgClose();

orient_group = DoInLineGroup(DcTrue);   /* rotation and scaling group */
     DgAddObj(DoRotate(DcYAxis, DTOR(90)));
     DgAddObj(DoScale(1.0, 1.0, 2.0));
     DgSetElePtr(0, DcBeginning);  /* prepare to alter the rotate object */
DgClose();

left_wheel = DoGroup(DcTrue);       /* left wheel group */
     DgAddObj(DoSubDivSpec(DcSubDivRelative, parms1));
     DgAddObj(DoRepType(DcWireframe));
     DgAddObj(DoTranslate(-2.0, 0.0, 0.0));
     DgAddObj (orient_group);       /* turns and scales the wheel */
```

```
     DgAddObj(wheel_group);    /* references the basic wheel shape */
DgClose();

right_wheel = DoGroup(DcTrue);      /* right wheel group */
     DgAddObj(DoSubDivSpec(DcSubDivRelative, parms2));
     DgAddObj(DoRepType(DcWireframe));
     DgAddObj(DoTranslate(2.0, 0.0, 0.0));
     DgAddObj(orient_group);        /* turns and scales the wheel */
     DgAddObj(wheel_group);    /* references the basic wheel shape */
DgClose();

axle_group = DoGroup(DcTrue); /*puts left and right wheels onto an axle*/
     DgAddObj(left_wheel);
     DgAddObj(right_wheel);
     DgAddObj(DoRotate(DcYAxis, DTOR(90)));
     DgAddObj(DoScale (0.1, 0.1, 4.0));
     DgAddObj(DoTranslate (0.0, 0.0, -0.5));
     DgAddObj(DoPrimSurf (DcCylinder));  /* creates the axle */
DgClose();

/* The objects are then added to a view called axle_view.
See Chapter 10 for more information on views. */

DvUpdate(axle_view);
DvSetUpdateType(DcUpdateDisplay); /* updates display group only */

for (i = 0; i<45; i++) {
     DgReplaceObjInGroup(orient_group, DoRotate(DcYAxis, DTOR(i)));
     DvUpdate(axle_view);
}
```

Fortran code:

```
      INTEGER*4 AXLE, WHEEL, LFTWH, RTWH, ORIENT, I
      REAL*8 GREEN(3)
      REAL*8 PARMS1(1)
      REAL*8 PARMS2 (1)
      PARAMETER (DEGRAD=0.0174533D0) ! converts degrees to radians !
      CHARACTER DUMMY
C
      DATA GREEN / 0.0D0, 1.0D0, 0.0D0 /
      DATA PARMS1 / 0.03D0 /
      DATA PARMS2 / 0.02D0 /
C
      WHEEL=DOG(DCTRUE) ! creates and opens wheel group !
         CALL DGAO(DODIFC(DORGB, GREEN))
         CALL DGAO(DOPUMX())
            CALL DGAO(DOROT(DCXAX,(DEGRAD*90.0D0))) ! rotates the torus !
            CALL DGAO(DOTOR(1.0D0, 0.3D0)) ! makes the torus !
         CALL DGAO(DOPPMX())
         CALL DGAO(DOSC(1.0D0, 1.0D0, 0.4D0))
         CALL DGAO(DOXLT(0.0D0, 0.0D0, -0.5D0))
         CALL DGAO(DOPMS(DCCYL)) ! makes a cylinder !
      CALL DGCS()
C
```

Example 2: In-Line Group
(continued)

```
      ORIENT=DOILG(DCTRUE) ! rotation and scaling group !
         CALL DGAO(DOROT(DCYAX, (DEGRAD*90.0D0)))
         CALL DGAO(DOSC(1.0D0, 1.0D0, 2.0D0))
         CALL DGSEP(0.0D0, DCBEG) ! prepare to alter rotate object !
      CALL DGCS()
C
      LFTWH=DOG(DCTRUE) ! left wheel group !
         CALL DGAO(DOSDS(DCSDRL, PARMS1))
         CALL DGAO(DOREPT(DCWIRE))
         CALL DGAO(DOXLT(-2.0D0, 0.0D0, 0.0D0))
         CALL DGAO(ORIENT) ! turns and scales the wheel !
         CALL DGAO(WHEEL) ! references the basic wheel shape !
      CALL DGCS()
C
      RTWH=DOG(DCTRUE) ! right wheel group !
         CALL DGAO(DOSCS(DCSDRL, PARMS2))
         CALL DGAO(DOREPT(DCWIRE))
         CALL DGAO(DOXLT(2.0D0, 0.0D0, 0.0D0))
         CALL DGAO(ORIENT) ! turns and scales the wheel !
         CALL DGAO(WHEEL)  ! references the basic wheel shape !
      CALL DGCS()
C
      AXLE=DOG(DCTRUE) ! puts left and right wheels onto an axle !
         CALL DGAO(LFTWH)
         CALL DGAO(RTWH)
         CALL DGAO(DOROT(DCYAX, (DEGRAD*90.0D0)))
         CALL DGAO(DOSC(0.1D0, 0.1D0, 4.0D0))
         CALL DGAO(DOXLT(0.0D0, 0.0D0, -0.5D0))
         CALL DGAO(DOPMS(DCCYL)) ! creates the axle !
      CALL DGCS
C     The objects are added to a view called AXVW.  See
C     Chapter 10 for more information on views.
      CALL DVU(AXVW)
      CALL DVSUT(DCUDIS) ! updates display group only !
C
      DO 20 I=0, 45
      CALL DGROG(ORIENT, DOROT(DCYAX, (DEGRAD*I)))
      CALL DVU(AXVW)
20    CONTINUE
```

Example 3: Using Labels to Position the Element Pointer

Example 3 shows how labels can be used within a group. **Labels are objects that act as place markers within groups and are used by several group editing functions.** This example contains three routines. The first makes three different front axles. The second makes a car. The third, which is called in response to a user request, substitutes the specified axle in the car group.

In the make_car routine, a label object is added before the front axle object. When the change_axle routine is called, the element pointer is moved to the beginning of the group with *DgSetElePtr* <*DGSEP*> because *DgSetElePtrRelLabel* <*DGSEPL*> searches only

from the current element pointer position to the end of the group. Then the element pointer is moved down one space past the FRONT_AXLE label object (with *DgSetElePtrRelLabel* <*DGSEPL*>). The element pointer now points to the front axle object. The change_axle routine allows different front axle groups to be substituted in the car group (with *DgReplaceObj* <*DGRO*>).

C code:

```
#define FRONT_AXLE  4
#define CHASSIS     5
#define HOOD        6
#define REAR_AXLE   7
DtObject  car, front_axle_1, front_axle_2, front_axle_3, hood, chassis,
          rear_axle_1;
make_axles()
{
     front_axle_1 = DoGroup(DcTrue);
          .           /* add objects to make front axle 1 */
          .
          .
     DgClose();
     DsHoldObj(front_axle_1);
     front_axle_2 = DoGroup(DcTrue);
          .           /* add objects to make front axle 2 */
          .
          .
     DgClose();
     DsHoldObj(front_axle_2);

     front_axle_3 = DoGroup(DcTrue);
          .          /* add objects to make front axle 3 */
          .
          .
     DgClose();
     DsHoldObj(front_axle_3);
}
make_car()
{
     car = DoGroup(DcTrue);
          DgAddObj(DoLabel(CHASSIS));
          DgAddObj(chassis);
          DgAddObj(DoLabel(REAR_AXLE));
          DgAddObj(rear_axle_1);
          DgAddObj(DoLabel(FRONT_AXLE));
          DgAddObj(front_axle_1);
          DgAddObj(DoLabel(HOOD));
          DgAddObj(hood);
     DgClose();
}
change_axle(axle_number)
DtInt axle_number;
{
     DgOpen(car, DcFalse);
```

Example 3: Using Labels to Position the Element Pointer (continued)

```
        DgSetElePtr(0, DcBeginning);
           DgSetElePtrRelLabel(FRONT_AXLE, 1);
           switch(axle_number)
                {
                case 1:
                          DgReplaceObj(front_axle_1);
                          break;
                case 2:
                          DgReplaceObj(front_axle_2);
                          break;
                case 3:
                          DgReplaceObj(front_axle_3);
                          break;
                }
     DgClose();
}
```

Fortran code:

```
      INTEGER*4 FRONT, CHASS, HOOD, REAR
      INTEGER*4 CAR, FRTAX(3), RAX1
      COMMON /AXLES/FRTAX(3),RAX1,CAR,FRONT,CHASS,HOOD,REAR
C
      SUBROUTINE MKAXL
      INTEGER*4 FRONT,CHASS,HOOD,REAR
      INTEGER*4 CAR,FRTAX(3),RAX1
      COMMON /AXLES/ FRTAX(3),RAX1,CAR,FRONT,CHASS,HOOD,REAR
      FRTAX(1)=DOG(DCTRUE)
         .          ! add objects to make front axle 1 !
         .
         .
      CALL DGCS()
      CALL DSHO(FRTAX(1))
C
      FRTAX(2)=DOG(DCTRUE)
         .          ! add objects to make front axle 2!
         .
         .
      CALL DGCS()
      CALL DSHO(FRTAX(2))
C
      FRTAX(3)=DOG(DCTRUE)
         .          ! add objects to make front axle 3!
         .
         .
      CALL DGCS()
      CALL DSHO(FRTAX(3))
      END
C
      SUBROUTINE MKCAR
      INTEGER*4 FRONT,CHASS,HOOD,REAR
      INTEGER*4 CAR,FRTAX(3),RAX1
      COMMON /AXLES/FRTAX(3),RAX1,CAR,FRONT,CHASS,HOOD,REAR
      CAR=DOG(DCTRUE)
         CALL DGAO(DOLL(CHS)
```

```
            CALL DGAO(CHASS)
            CALL DGAO(DOLL (RAX)
            CALL DGAO(RAX1)
            CALL DGAO(DOLL(FAX)
            CALL DGAO(FRTAX(1))
            CALL DGAO(DOLL(HD))
            CALL DGAO(HOOD)
         CALL DGCS()
         END
C
         SUBROUTINE CHGAXL (AXNBR)
         INTEGER*4 AXNBR
         INTEGER*4 FRONT,CHASS,HOOD,REAR
         INTEGER*4 CAR,FRTAX(3),RAX1
         COMMON /AXLES/FRTAX(3),RAX1,CAR,FRONT,CHASS,HOOD,REAR
         CALL DGO(CAR, DCFALS)
         CALL DGSEP(0, DCBEG)
         CALL DGSEPL(FAX,1)
      CALL DGRO(FRTAX(AXNBR))
         CALL DGCS()
         END
```

Using an Empty In-Line Group

A useful way to make multiple replacements of a single object is to create an empty in-line group and then make multiple calls to *DgReplaceObjInGroup* <*DGROG*>, using a new rotate object each time. Figure 4-11 shows the empty in-line group, which will contain a rotate object after the first call to *DgReplaceObjInGroup*. (The *DoPushMatrix* <*DOPUMX*> and *DoPopMatrix* <*DOPPMX*> are used to localize the effect of the rotate object; see Chapter 7.)

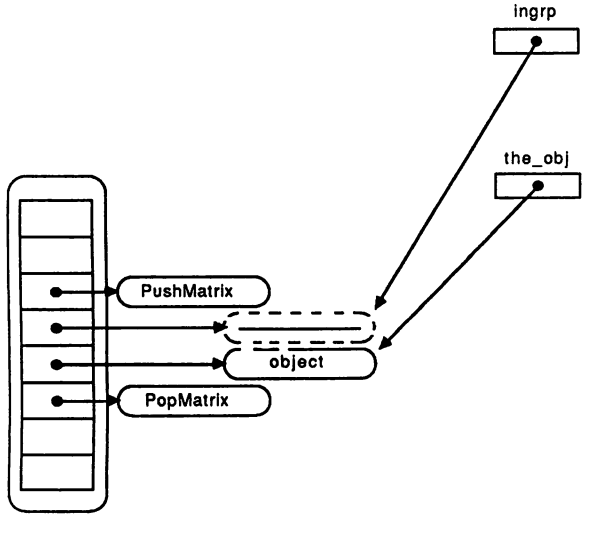


Figure 4-11. Making Replacements in an Empty In-Line Group

Using an Empty In-Line Group (continued)

```
C code:

ingrp = DoInLineGroup(DcFalse);     /* empty in-line group */


DgAddObj(DoPushMatrix());
     DgAddObj(ingrp);
     DgAddObj(the_obj);
DgAddObj(DoPopMatrix());



for (i=0; i<360; i++){
     DdUpdate(device);
     DgReplaceObjInGroup(ingrp, DoRotate (DcYAxis, DTOR(i)));
}


Fortran code:

      INGRP = DOILG(DCFALS)

      CALL DGAO (DOPUMX())
         CALL DGAO(INGRP)
         CALL DGAO(OBJECT)
      CALL DGAO(DOPPMX())

      DO 600 I=0,359
         CALL DDU(DEVICE)
         CALL DGROG(INGRP, DOROT(DCYAX, DEGRAD*I))
600   CONTINUE
```

With *DgReplaceObjInGroup* <*DGROG*> (and other -replace functions), the element pointer does not move. It continues to point to the same object (the only object in *ingrp*), so that object is continually replaced by new rotate objects.

Holding and Releasing Objects

Each object has a *reference count* that is incremented by one each time the object is added to an organizational object (group, view, and so on). The object's reference count is decremented by one each time the object is removed from the organizational object. When the reference count reaches 0 (that is, there are no references to the object in the Doré database), Doré automatically deletes the object from the Doré database.

If you do not want the object deleted, use the *DsHoldObj* <*DSHO*> function to retain the object in the Doré database. Use *DsReleaseObj* <*DSRO*> to cancel the hold on the object. This

method for holding objects in memory is useful, but should not be overused, since it is unwise to waste memory on the storage of unneeded objects.

In Example 3, suppose you replace one axle group with a new one, but then want to see the original axle group again. The application would need to make sure that the initial axle group object was not reclaimed when it was bumped from the car group. Its reference count would go from 1 to 0, and Doré would check its hold flag to see if the object should be deleted. Because the axle creation routine calls *DsHoldObj <DSRO>* on each of the axle groups after they are created, they will not be deleted from the Doré database when they are removed from the car group.

Chapter Summary

Chapter 4 describes the basic building block of the Doré Library, the *object*. An object is a collection of data and a set of methods that operate on that data. Related objects are organized into *groups*. Objects are affected by the attributes set by previous objects in their group. If one group contains another group, the "child" group also inherits the attributes of its "parent" group.

Object creation simply stores geometric information in the Doré database. Nothing is actually drawn until one of the update functions (*DdUpdate, DfUpdate, DvUpdate <DDU, DFU, DVU>*) is called. At that time, part of the stored database is traversed, and the render method is applied to each object in the database.

Organizational objects include groups, views, frames, and devices. A number of group editing functions allow you to change the information stored in a group's "data" field. Objects can be added to and deleted from groups. A variety of *group editing functions* enable you to manipulate objects within groups. *Label objects* act as place markers within groups and are used by several group editing functions.

Although nonorganizational objects cannot be edited, they can be *replaced* by other objects with the desired value. A hold-and-release technique is available to save an object in the Doré database even though all Doré references to that particular object have been temporarily deleted. An object is referred to by its *object handle*. You can also create a *variable* to facilitate multiple references to the same object.

PRIMITIVE OBJECTS

CHAPTER FIVE

This chapter describes the major primitive objects available in the Doré Library and how to specify them. Concepts and terms introduced in this chapter include color models, vertex types, geometric normals, vertex normals, mesh objects, the right-hand rule, the inside/out rule, and variable-data primitives.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DoAnnoText <DOANNT>
DoLineList <DOLINL>
DoLineType <DOLNT>
DoLineWidth <DOLW>
DoMarkerFont <DOMF>
DoMarkerGlyph <DOMG>
DoNURBSurf <DONRBS>
DoPatch <DOPAT>
DoPointList <DOPNTL>
DoPolygon <DOPGN>
DoPolygonMesh <DOPGNM>
DoPolyline <DOPL>
DoPolymarker <DOPM>
DoPrimSurf <DOPMS>
DoSimplePolygon <DOSPGN>
DoSimplePolygonMesh <DOSPM>
DoText <DOTXT>
DoTorus <DOTOR>
DoTriangleList <DOTRIL>
DoTriangleMesh <DOTRIM>
DoVarLineList <DOVLNL>
DoVarPointList <DOVPTL>
DoVarSimplePolygonMesh <DOVSPM>
DoVarTriangleMesh <DOVTRM>
DpUpdVarLineList <DPUVLL>

Related Functions
(continued)

DpUpdVarPointList <DPUVPL>
DpUpdVarSimplePolygonMesh <DPUVSM>
DpUpdVarTriangleMesh <DPUVTM>

Primitive Objects

A *primitive object* is a displayable geometric object, such as a line, triangle, polygon, or patch. Lines, triangles, polygons, and meshes all use vertices explicitly to compose the primitive object. Spheres, cylinders, boxes, cones, and toruses are examples of primitive objects which do not use explicit vertex data. Text is also classified as a primitive object (see Chapter 8, "Text"). When a method is applied to an object (see Chapter 12), a primitive object uses the current primitive attributes. **Primitive objects are bound most tightly to the attributes closest to (and above) them.** In the following example

```
C code:

DgAddObj (DoScale (1.0, 1.0, 0.4));
DgAddObj (DoTranslate (0.0, 0.0, -0.5));
DgAddObj (DoPrimSurf (DcCylinder));


Fortran code:

      CALL DGAO(DOSC(1.0D0, 1.0D0, 0.4D0))
      CALL DGAO(DOXLT(0.0D0, 0.0D0, -0.5D0))
      CALL DGAO(DOPMS(DCCYL))
```

the primitive object, a cylinder, is first translated so that it is centered on the origin (-0.5 on the z axis), and then it is scaled disproportionately in the *x, y, z* directions. (Actually, the scale object is *executed* first, but it *applies* last.)

DoPrimSurf

DoPrimSurf <DOPMS> creates a primitive object that defines one of the following closed surfaces:

- sphere
- cylinder
- box
- cone

The primitive surface is defined in a standard location with a standard size and orientation, and you can apply geometric transformations to alter its form, as shown in the cylinder example above.

DoTorus

DoTorus <*DOTOR*> creates a primitive object that defines a doughnut- shaped surface. The car wheel example in Chapter 4 used a torus for the tire. The syntax for *DoTorus* is

```
DoTorus (bigradius, smallradius)
```

where

bigradius
specifies the distance from the center of the torus hole to the center of the ring

smallradius
specifies the radius of the cross-section of the ring itself.

The torus is centered at the origin and is in the *xz* plane.

DoLineList

DoLineList <*DOLINL*> creates a primitive object that defines a list of independent lines. The syntax for *DoLineList* is

```
DoLineList (colormodel, vertextype, linecount, vertices)
```

where

colormodel
specifies the color model used by the vertex type,

vertextype
specifies what type of information will be given for each vertex (location only, or whether information on vertex normals and/or vertex colors will be given as well)

linecount
specifies the number of lines in the list

vertices
specifies the vertex information for the two endpoints of each line, in floating point values

Color model, vertex type, and *vertices,* described in more detail below, are also used in many other primitive object functions, such as *DoTriangleList* <*DOTRIL*>, *DoSimplePolygon* <*DOSPGN*>, and *DoPolygon* <*DOPGN*>.

Color Model

The Doré Library currently supports the RGB (red-green-blue) color model (*DcRGB <DCRGB>*). The Doré include file lists the possible values for *DtColorModel.*

Vertex Type

The *vertextype* parameter can be one of the following:

DcLoc <DCL>
location of the vertex in x, y, z

DcLocNrm <DCLN>
location plus a vertex normal with an x, y, z direction

DcLocClr <DCLC>
location plus a vertex color

DcLocNrmClr <DCLNC>
location, vertex normal, and vertex color for each vertex

Examples of Vertex Locations

The simplest vertex type is *DcLoc <DCL>*, which consists of three floating point values that specify the location in modeling coordinates for each vertex. The following example draws a line from the origin to point (1.0, 1.0, 1.0). *LineLoc* is the linear array of vertices that form the line segment.

```
C code:

DtObject line;
static DtReal LineLoc[] = {0.0, 0.0, 0.0, 1.0, 1.0, 1.0};

line = DoLineList (DcRGB, DcLoc, 1, LineLoc);


Fortran code:

      INTEGER*4 LINE
      REAL*8 LINELC(6)
C
      DATA LINELC / 0.0D0, 0.0D0, 0.0D0, 1.0D0, 1.0D0, 1.0D0 /
C
      LINE=DOLINL(DCRGB,DCL,1,LINELC)
```

Vertex Normals

Vertices of type *DcLocNrm <DCLN>* consist of their *x, y, z* location, specified by three floating point values, and a vertex *normal*, which is used for shading purposes and is specified by its *x, y, z* direction, in floating point values. A vertex *normal* is a vector of unit length that is assumed to be perpendicular to the primitive object at that point.

When *DoInterpType <DOIT>* is set to *DcVertexShade <DCVXSH>* or *DcSurfaceShade <DCSFSH>*, the Doré Library uses vertex normals to calculate the shading across an object. If you specify *DcLoc <DCL>* and do not include vertex normals in the definition of the primitive object, in some cases Doré will compute the vertex normals for you. (See the description of *smooth flag*, below.) When you want to affect the appearance of an object by using different normals from the actual normals, specify vertices of type *DcLocNrm <DCLN>* or *DcLocNrmClr <DCLNC>*.

Vertex Colors

Vertices of type *DcLocClr <DCLC>* consist of their *x, y, z location*, specified by three floating point values, and a vertex *color* defined by the *colormodel* argument. For the RGB color model, red, green, and blue intensities are each specified by a floating point value, normally between 0.0 and 1.0. The following example draws a line with a red vertex at the origin and a blue vertex at (1.0, 1.0, 1.0). If *DoInterpType <DOIT>* is set to *DcConstantShade <DCCNSH>* (which averages the vertex colors), the line appears magenta.

```
C code:

DtObject line;
static DtReal LineVerts[] =
               {0.0, 0.0, 0.0,   1.0, 0.0, 0.0,
                1.0, 1.0, 1.0,   0.0, 0.0, 1.0};

line = DoLineList(DcRGB, DcLocClr, 1, LineVerts);

Fortran code:

      INTEGER*4 LINE
      REAL*8 LVRTS(12)
C
      DATA LVRTS / 0.0D0, 0.0D0, 0.0D0,   1.0D0, 0.0D0, 0.0D0,
     1             1.0D0, 1.0D0, 1.0D0,   0.0D0, 0.0D0, 1.0D0 /
C
      LINE=DOLINL(DCRGB,DCLC,1,LVRTS)
```

If you create objects with vertex colors (with vertices of type *DcLocClr* <DCLC> or *DcLocNrmClr* <DCLNC>), any colors specified by *DoDiffuseColor* <DODIFC> are ignored by those objects. Vertex colors and normals are "attached" to the primitive object in a permanent way.

Vertex Normals and Colors

Vertices of type *DcLocNrmClr* <DCLNC> are specified by nine floating point values for each vertex (assuming the RGB color model is used), as follows:

location
three floating point values for *x*, *y*, and *z* location

normal
three floating point values for *x*, *y*, and *z* direction

color
three floating point values for red, green, and blue

DoPolyline

DoPolyline <DOPL> creates a primitive object that defines a series of *connected* line segments. (For independent line segments, use *DoLineList* <DOLINL>.) The syntax for *DoPolyLine* is

```
DoPolyline (colormodel, vertextype, vertexcount, vertices)
```

where

vertexcount
is the number of vertices specified in the *vertices* array

The other parameters for *DoPolyline* <DOPL> are the same as those for *DoLineList* <DOLINL>, described above.

Example

The following example draws a sample polyline, shown in Figure 5-1.

```
C code:

DtObject poly;

static DtReal Heart[] =
```

```
          { 0.0,  0.2,   0.0,
            0.07,  0.3,  0.0,
            0.16,  0.34, 0.0,
            0.23,  0.33, 0.0,
            0.3,   0.26, 0.0,
            0.32,  0.14, 0.0,
            0.28,  0.0,  0.0,
            0.12, -0.23, 0.0,
            0.0,  -0.3,  0.0,
           -0.12, -0.23, 0.0,
           -0.28,  0.0,  0.0,
           -0.32,  0.14, 0.0,
           -0.3,   0.26, 0.0,
          - 0.23,  0.33, 0.0,
          - 0.16,  0.34, 0.0,
          - 0.07,  0.3,  0.0,
            0.0,   0.2,  0.0,

            0.1,  0.1, 0.0,
            0.1, -0.1, 0.0,
           -0.1, -0.1, 0.0,
           -0.1,  0.1, 0.0,
            0.1,  0.1, 0.0,
           -0.1, -0.1, 0.0,
            0.0, -0.2, 0.0,
            0.1, -0.1, 0.0,
           -0.1,  0.1, 0.0};

poly = DoPolyline (DcRGB, DcLoc, 26, Heart);
```

Fortran code:

```
      INTEGER*4 POLY
      REAL*8 HEART(78)
C
      DATA HEART / 0.0D0,  0.2D0, 0.0D0,  0.07D0,  0.3D0,  0.0D0,
     1             0.16D0, 0.34D0,0.0D0,  0.23D0,  0.33D0, 0.0D0,
     2             0.3D0,  0.26D0,0.0D0,  0.32D0,  0.14D0, 0.0D0,
     3             0.28D0, 0.0D0, 0.0D0,  0.12D0, -0.23D0, 0.0D0,
     4             0.0D0, -0.3D0, 0.0D0, -0.12D0, -0.23D0, 0.0D0,
     5            -0.28D0, 0.0D0, 0.0D0, -0.32D0,  0.14D0, 0.0D0,
     6            -0.3D0,  0.26D0,0.0D0, -0.23D0,  0.33D0, 0.0D0,
     7            -0.16D0, 0.34D0,0.0D0, -0.07D0,  0.3D0,  0.0D0,
     8             0.0D0,  0.2D0, 0.0D0,
C
                   0.1D0,  0.1D0, 0.0D0,  0.1D0,  -0.1D0, 0.0D0,
     9            -0.1D0, -0.1D0, 0.0D0, -0.1D0,   0.1D0, 0.0D0,
     1             0.1D0,  0.1D0, 0.0D0, -0.1D0,  -0.1D0, 0.0D0,
     2             0.0D0, -0.2D0, 0.0D0,  0.1D0,  -0.1D0, 0.0D0,
     3            -0.1D0,  0.1D0, 0.0D0  /
C
      POLY=DOPL(DCRGB, DCL, 26, HEART)
```

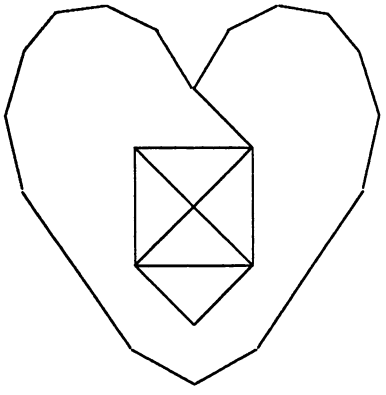


Figure 5-1. Sample Polyline

Line Type and Line Width Attributes

The *DoLineList* <*DOLINL*> and *DoPolyLine* <*DOPL*> primitive objects can use the *DoLineType* <*DOLNT*> and *DoLineWidth* <*DOLW*> attribute objects to determine the current line style and width. The line type attribute also affects the wires generated by the *DcWireframe* <*DCWIRE*> representation type. Possible line type values are

DcLineTypeSolid <DCLTS>
solid

DcLineTypeDash <DCLTD>
dashed

DcLineTypeDot <DCLTDT>
dotted

DcLineTypeDotDash <DCLTDD>
dot and dash repeated

The line width argument to *DoLineWidth* <*DOLW*> is a scaling factor, specified as a floating point value. The default value for *DoLineWidth* is 1.0.

DoTriangleList

DoTriangleList <*DOTRIL*> creates a primitive object that defines a list of independent triangles. The syntax for *DoTriangleList* is

```
DoTriangleList (colormodel, vertextype, trianglecount, vertices)
```

where

trianglecount
is the number of triangles in the list

vertices
specifies the vertex information for the three vertices of each triangle, in floating point values

Geometric Normal

The *geometric normal* is assumed to be perpendicular to the surface of the triangle. The *geometric normal,* which is used for backface culling (see below), should not be confused with a *vertex normal,* which is used for shading interpolations.

Right-hand Rule

The Doré Library computes the geometric normal for you according to the *right-hand rule.* The right-hand rule specifies that if the fingers of the right hand curl around and follow the boundary of the surfaces in the order the vertices are specified, then the outstretched thumb of the right hand points in the same general direction as the normal calculated from the surface (see Figure 5-2).

In order to have front-facing normals, specify the vertices in a counterclockwise order relative to the viewer. The polygon in Figure 5-2 is front-facing, according to the right-hand rule.

Backface Culling

Backface culling is an efficiency technique in which surfaces whose normals point away from the viewer are not drawn at all. For opaque, closed surfaces, backface culling involves using the geometric normal to define the front and back faces of the surface. Then, surfaces that are back facing, and hence cannot be seen, are discarded before the object is shaded and rendered. (The torus in the Chapter 4 examples was backface culled. See also Plates 3 and 4.)

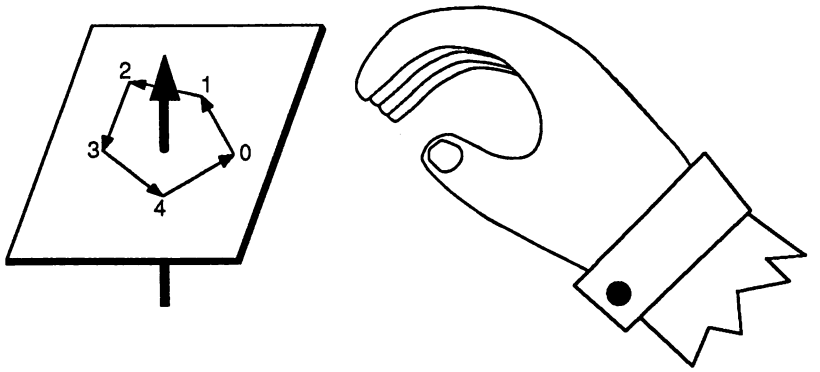


Figure 5-2. Using the Right-hand Rule to Specify the Geometric Normal

Triangle Examples

The following example draws a triangle with vertices of type *DcLoc <DCL>*. The Doré Library computes the geometric normal according to the right-hand rule, and the vertices are specified in counterclockwise order.

C code:

```
DtObject tri;
static DtReal  Verts[] =
               {5.0, 0.0, 0.0,
                0.0, 2.5, 0.0,
               -6.0, 1.2, 0.0};

tri = DoTriangleList (DcRGB, DcLoc, 1, Verts);
```

Fortran code:

```
      INTEGER*4 TRI
      REAL*8 VERTS(9)
C
      DATA VERTS / 5.0D0, 0.0D0, 0.0D0,
     1             0.0D0, 2.5D0, 0.0D0,
     2            -6.0D0, 1.2D0, 0.0D0 /
C
      TRI=DOTRIL(DCRGB, DCL, 1, VERTS)
```

The next example draws a triangle with vertices of type *DcLocNrm <DCLN>*.

C code:

```
DtObject trilist;
static DtReal  Verts[] =
               {0.0, 1.0, 0.0,    -1.0,  1.0, 0.0,
                0.0, 0.0, 0.0,    -1.0, -1.0, 0.0,
                1.0, 0.0, 0.0,     1.0, -1.0, 0.0};

trilist = DoTriangleList (DcRGB, DcLocNrm, 1, Verts);
```

Fortran code:

```
      INTEGER*4 TRILST
      REAL*8 VERTS (18)
C
      DATA VERTS / 0.0D0, 1.0D0, 0.0D0   -1.0D0,  1.0D0, 0.0D0,
     1             0.0D0, 0.0D0, 0.0D0,  -1.0D0, -1.0D0, 0.0D0,
     2             1.0D0, 0.0D0, 0.0D0,   1.0D0, -1.0D0, 0.0D0 /
C
      TRILST=DOTRIL(DCRGB, DCLN, 1, VERTS)
```

Mesh Objects

DoTriangleList <DOTRIL>, as well as a number of other primitive objects, has a corresponding mesh object function:

DoTriangleList <DOTRIL>; DoTriangleMesh <DOTRIM>
DoSimplePolygon <DOSPGN>; DoSimplePolygonMesh <DOSPM>
DoPolygon <DOPGN>; DoPolygonMesh <DOPGNM>

In general, mesh objects are useful because they

- consume less memory than would be consumed if their components were specified individually
- can be rendered more quickly than a set of individual primitives
- can be used to approximate a smooth surface
- can guarantee that contiguous mesh elements are adjacent, with no "cracks" between them

DoTriangleMesh

DoTriangleMesh <DOTRIM> creates a primitive object that defines a collection of triangles. Normally, the triangles are interconnected, forming a surface, but they need not all be in one piece.

The syntax for *DoTriangleMesh* is

```
DoTriangleMesh (colormodel, vertextype, vertexcount, vertices,
                triangle count, triangles, smoothflag)
```

(See above for descriptions of the *colormodel* and *vertextype* parameters.)

where

vertexcount
is the number of vertices

vertices
is an array listing all data for each vertex, as floating point values. Each vertex need be listed only once.

trianglecount
is the number of triangles in the mesh

triangles
is an array containing three indices for each triangle. These indices correspond to the mesh vertex numbers for the three coordinates of each triangle. The vertices are numbered in the order specified in *vertices*, starting with 0. The last vertex referenced is assumed to connect back to the first.

smoothflag
indicates whether the object is intended to approximate a smooth surface. If *smoothflag* is *DcTrue* and if vertex normals were not provided, the geometric normals from a vertex's adjacent surfaces are averaged, and the result is used for shading. If *smoothflag* is *DcFalse*, vertex normals are not generated.

The following function call creates an icosahedron.

C code:

```
DtObject trimesh;
static DtReal Vertices[] =
          { 0.0,       1.0,       1.618034,      /* vertex 0 */
            0.0,       1.0,      -1.618034,      /* vertex 1 */
            0.0,      -1.0,       1.618034,      /* vertex 2 */
            0.0,      -1.0,      -1.618034,
            1.0,       1.618034, 0.0,
            1.0,      -1.618034, 0.0,
           -1.0,       1.618034, 0.0,
           -1.0,      -1.618034, 0.0,
            1.618034, 0.0,       1.0,
```

```
            1.618034, 0.0,      -1.0,
           -1.618034, 0.0,       1.0,
           -1.618034, 0.0,      -1.0};          /* vertex 11 */

static DtInt TriList[] =
          {  0,  4,  6,        /*vertices for first triangle */
             6,  4,  1,
             1,  4,  9,
             9,  4,  8,
             8,  4,  0,
             8,  0,  2,
             2,  0, 10,
            10,  0,  6,
            10,  6, 11,
            11,  6,  1,
            11,  1,  3,
             3,  1,  9,
             3,  9,  5,
             5,  9,  8,
             5,  8,  2,
             5,  2,  7,
             7,  2, 10,
             7, 10, 11,
             7, 11,  3,
             7,  3,  5};

trimesh = DoTriangleMesh (DcRGB, DcLoc, 12, Vertices, 20,
TriList, DcFalse);


Fortran code:

      INTEGER*4 TRIMSH
      REAL*8 VERTS (36)
      INTEGER*4 TRILST(60)
C
      DATA VERTS /0.0D0, 1.0D0,       1.618034D0,  ! vertex 0 !
     1       0.0D0,       1.0D0,      -1.618034D0,  ! vertex 1 !
     2       0.0D0,      -1.0D0,       1.618034D0,  ! vertex 2 !
     3       0.0D0,      -1.0D0,      -1.618034D0,
     4       1.0D0,       1.618034D0, 0.0D0,
     5       1.0D0,      -1.618034D0, 0.0D0,
     6      -1.0D0,       1.618034D0, 0.0D0,
     7      -1.0D0,      -1.618034D0, 0.0D0,
     8       1.618034D0, 0.0D0,       1.0D0,
     9       1.618034D0, 0.0D0,      -1.0D0,
     1      -1.618034D0, 0.0D0,       1.0D0,
     2      -1.618034D0, 0.0D0,      -1.0D0 /       ! vertex 11 !
C
      DATA TRILST / 0,  4,  6,     ! vertices for first triangle !
     1              6,  4,  1,
     2              1,  4,  9,
     3              9,  4,  8,
     4              8,  4,  0,
     5              8,  0,  2,
     6              2,  0, 10,
```

```
7              10,  0,  6,
8              10,  6, 11,
9              11,  6,  1,
1              11,  1,  3,
2               3,  1,  9,
3               3,  9,  5,
4               5,  9,  8,
5               5,  8,  2,
6               5,  2,  7,
7               7,  2, 10,
8               7, 10, 11,
9               7, 11,  3,
1               7,  3,  5 /
C
      TRIMSH=DOTRIM(DCRGB,DCL,12,VERTS,20,TRILST,DCFALS)
```

Plate 5 shows a triangle mesh with a *surface* representation type (see Chapter 6 for a discussion of representation type) and with the smooth flag set to *DcFalse*. Plate 6 shows the surface triangle mesh with the smooth flag set to *DcTrue* so that it approximates a smooth surface.

DoSimplePolygon

DoSimplePolygon <*DOSPGN*> creates a primitive object that defines a simple polygon. A *simple polygon* is a planar collection of vertices that form a single, connected, contour. The polygon can contain any number of vertices, and it can be convex, concave, or self-intersecting (see the description of "Shape," below).

The syntax for *DoSimplePolygon* is

```
DoSimplePolygon (colormodel, vertextype, vertexcount, vertices,
                 shape)
```

See above for descriptions of *colormodel* and *vertextype*.

where

vertex count
is the number of vertices

vertices
is an array listing all data for each vertex of the polygon as floating point numbers

shape
provides a hint about the geometry of the simple polygon.

Possible values for *shape* are

DcConvex <DCCNVX>
contour is wholly convex

DcConcave <DCCNCV>
contour is not self-intersecting but is not wholly convex

DcComplex <DCCPLX>
contour may be self-intersecting

Correct specification of the contour shape can speed up rendering. Any polygon will render correctly if you specify *DcComplex <DCCPLX>*. If the polygon is one of the simpler types (*DcConvex <DCCNVX>* or *DcConcave <DCCNCV>*), it will render more quickly if you specify the correct type. (In some cases, for example if you specify *DcConvex <DCCNVX>*, and the polygon is actually *DcComplex*, the results are undefined.)

Vertices and the Right-hand Rule

As discussed above in the "Geometric Normal" section, the order in which you specify vertices affects the geometric normal calculated by the Doré Library. For polygons, the Doré Library uses the first three non-colinear points to determine the geometric normal. For polygons with multiple contours, the first contour is used to compute the geometric normal. To have front-facing normals, specify the vertices in a counterclockwise order.

Figure 5-3 shows a sample polygon, created by the function call *DoSimplePolygon <DOSPGN>*.

C code:

```
DtObject poly;
static DtReal pentagon[] =
          {0.0,      1.0,     0.0,
           -.951,     .309,   0.0,
           -.587,    -.809,   0.0,
            .587,    -.809,   0.0,
            .951,     .309,   0.0};

poly = DoSimplePolygon (DcRGB, DcLoc, 5, pentagon, DcConvex);
```

Fortran code:

```
      INTEGER*4 POLY
      REAL *8 PENT (15)
C
```

```
      DATA PENT / 0.0D0,     1.0D0,     0.0D0,
     1            -.951D0,    .309D0,   0.0D0,
     2            -.587D0,   -.809D0,   0.0D0,
     3             .587D0,   -.809D0,   0.0D0,
     4             .951D0,    .309D0,   0.0D0 /
C
      POLY=DOSPGN(DCRGB,DCL,5,PENT,DCCNVX)
```

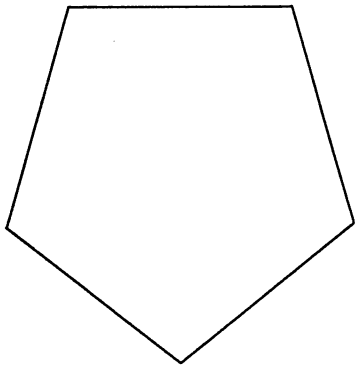


Figure 5-3. Example of a Simple Polygon

DoPolygon

DoPolygon <*DOPGN*> creates a primitive object that defines a polygon that may have more than one contour—for example, an outer boundary or boundaries plus interior holes or islands. The contours can be self-intersecting and can intersect each other. (For polygons with only one contour, use *DoSimplePolygon* <*DOSPGN*>.)

The syntax for *DoPolygon* is

```
DoPolygon (colormodel, vertextype, contourcount, contours,
           vertices, shape);
```

(See above for descriptions of the *colormodel, vertextype,* and *shape* parameters.)

where

contourcount
is the number of contours in the polygon

contours
is an array of integers that lists how many vertices each contour contains

vertices

is an array of vertices for each contour of the polygon, beginning with vertex 0. The last vertex of a contour is implicitly connected to the first vertex of that contour.

Figure 5-4 shows the complex polygon created by the following function call.

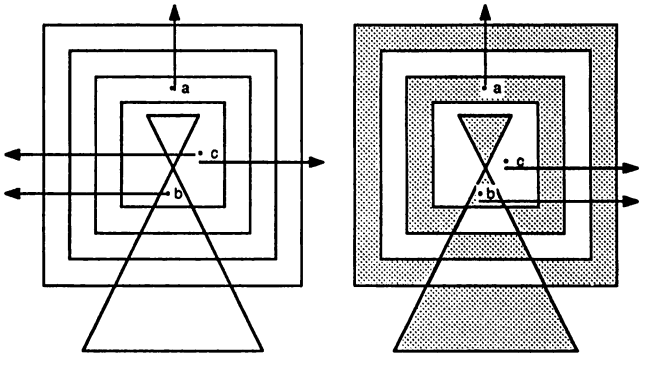
C code:

```
DtObject poly;
static DtReal Verts [] =
     { 3.0, 3.0, 0.0,
      -3.0, 3.0, 0.0,
      -3.0,-3.0, 0.0,
       3.0,-3.0, 0.0,

       5.0, 5.0, 0.0,
      -5.0, 5.0, 0.0,
      -5.0,-5.0, 0.0,
       5.0,-5.0, 0.0,

       7.0, 7.0, 0.0,
      -7.0, 7.0, 0.0,
      -7.0,-7.0, 0.0,
       7.0,-7.0, 0.0,

       9.0, 9.0, 0.0,
      -9.0, 9.0, 0.0,
      -9.0,-9.0, 0.0,
       9.0,-9.0, 0.0,

       1.0, 1.0, 0.0,
      -1.0, 1.0, 0.0,
       6.0,-13.0, 0.0,
      -6.0,-13.0, 0.0};

static DtInt Conts[] = {4, 4, 4, 4, 4};

poly = DoPolygon (DcRGB, DcLoc, 5, Conts, Verts, DcComplex);
```

Fortran code:

```
      INTEGER*4 POLY
      REAL*8 VERTS (60)
      INTEGER*4 CONTS(5)
C
      DATA VERTS / 3.0D0, 3.0D0, 0.0D0,
     1             -3.0D0, 3.0D0, 0.0D0,
     2             -3.0D0,-3.0D0, 0.0D0,
     3              3.0D0,-3.0D0, 0.0D0,
C
     4              5.0D0, 5.0D0, 0.0D0,
```

```
     5              -5.0D0, 5.0D0, 0.0D0,
     6              -5.0D0,-5.0D0, 0.0D0,
     7               5.0D0,-5.0D0, 0.0D0,
C
     8               7.0D0, 7.0D0, 0.0D0,
     9              -7.0D0, 7.0D0, 0.0D0,
     1              -7.0D0,-7.0D0, 0.0D0,
     2               7.0D0,-7.0D0, 0.0D0,
C
     3               9.0D0, 9.0D0, 0.0D0,
     4              -9.0D0, 9.0D0, 0.0D0,
     5              -9.0D0,-9.0D0, 0.0D0,
     6               9.0D0,-9.0D0, 0.0D0,
C
     7               1.0D0, 1.0D0, 0.0D0,
     8              -1.0D0, 1.0D0, 0.0D0,
     9               6.0D0,-13.0D0, 0.0D0,
     1              -6.0D0,-13.0D0, 0.0D0 /
C
     DATA CONTS / 4, 4, 4, 4, 4 /
C
     POLY = DOPGN(DCRGB, DCL, 5, CONTS, VERTS, DCCPLX)
```



Representation Type: Wireframe Representation Type: Surface

Figure 5-4. Complex Polygon

Inside/Out Rule

For a set of contours such as those shown in Figure 5-4, it is not obvious at first glance whether a given point is inside the polygon. For polygons and polygon meshes, use the inside/out rule, as follows. Starting at the point in question, draw a line out of the polygon. Then count the number of contours it intersects. If it crosses an odd number of contours, the point is *in* the polygon (for example, points *a* and *b* in Figure 5-4). If it crosses

an even number of contours, the point is outside the polygon (point *c* in Figure 5-4).

DoSimplePolygonMesh

DoSimplePolygonMesh <DOSPM> creates a primitive object that defines a collection of polygons. Normally, the polygons are interconnected, but they need not all be in one piece. The syntax for *DoSimplePolygonMesh* is

```
DoSimplePolygonMesh(colormodel, vertextype, vertexcount,
                vertices, polygoncount, contours, vertexlist,
                shape, smoothflag)
```

See above for descriptions of the *colormodel, vertextype, shape,* and *smoothflag.*

where

vertex count
is the total number of vertices

vertices
is an array listing all data for each vertex of the mesh as floating point numbers

polygoncount
is the number of polygons in the mesh

contours
is an array of *polygoncount* integers that says how many indices are used to define each simple polygon

vertexlist
specifies the connectivity of the vertices. *Vertexlist* is a one-dimensional array listing the vertices for each contour. The first polygon in the mesh begins with the vertex specified by *vertexlist[0]* and continues using the vertices specified by the following entries in *vertexlist*. The last vertex of a contour is implicitly connected to its first vertex.

Figure 5-5 shows the simple polygon mesh created by the following function call:

```
C code:

DtObject spmesh;
```

```
static DtReal verts []=
     {10.0, 0.0, 0.0,
      10.0, 10.0, 0.0,
      0.0, 10.0, 0.0,
      0.0, 0.0, 0.0,
      0.0, 10.0, -10.0,
      0.0, 0.0, -10.0,
      10.0, 0.0, -10.0,
      10.0, 10.0, -10.0};
static DtInt contours [] = {4,4,4,4,4,4};
static DtInt vlist [] =
     {0,1,2,3,
      3,2,4,5,
      5,4,7,6,
      6,7,1,0,
      0,3,5,6,
      1,7,4,2};

spmesh = DoSimplePolygonMesh(DcRGB, DcLoc, 8, verts,
        6, contours, vlist, DcConvex, DcFalse);
```

Fortran code:

```
      INTEGER*4 SPMESH, CONTS(6), VLIST(24)
      REAL*8 VERTS(24)
C
      DATA VERTS / 10.0D0,  0.0D0,   0.0D0,
     1             10.0D0, 10.0D0,   0.0D0,
     2              0.0D0, 10.0D0,   0.0D0,
     3              0.0D0,  0.0D0,   0.0D0,
     4              0.0D0, 10.0D0, -10.0D0,
     5              0.0D0,  0.0D0, -10.0D0,
     6             10.0D0,  0.0D0, -10.0D0,
     7             10.0D0, 10.0D0, -10.0D0 /
C
      DATA CONTS / 4,4,4,4,4,4 /
C
      DATA VLIST / 0,1,2,3,
     1             3,2,4,5,
     2             5,4,7,6,
     3             6,7,1,0,
     4             0,3,5,6,
     5             1,7,4,2 /
C
      SPMESH=DOSPM(DCRGB, DCL, 8, VERTS, 6, CONTS,
     1             VLIST, DCCNVX, DCFALS)
```

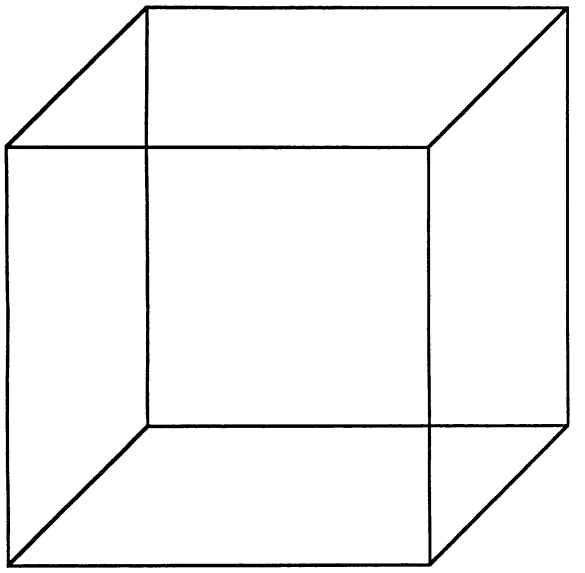


Figure 5-5. Example of a Simple Polygon Mesh

Variable Data Primitives

As described in Chapter 4, standard Doré primitives copy all data used to create an object into the object's private data space. Since the application program no longer has access to this private data, it cannot modify the object.

In contrast, *variable data primitives* do not copy vertex data into the object's private space. They maintain pointers to the vertex data, which remains in user data space. An application can thus efficiently modify the object, without the overhead of copying the data. Variable data primitives include

DoVarPoint List <DOVPTL>
variable point list; allows the application to modify the number of points, vertex locations, normals, and colors.

DoVarLineList <DOVLNL>
variable line list; allows the application to modify number of lines, vertex locations, normals, and colors.

DoVarTriangleMesh <DOVTRM>
variable triangle mesh; allows the application to modify vertex locations, normals, and colors (but does not allow the application to modify the number of triangles in the mesh or the connectivity of the mesh).

DoVarSimplePolygonMesh <DOVSPM>
variable simple polygon mesh; allows the application to modify vertex locations, normals, and colors (but does not allow the application to modify the number of polygons in the mesh or the connectivity of the mesh).

Each of these object creation routines has a corresponding *Dp-* routine that informs Doré that the user data has changed:

DpUpdVarPointList <DPUVPL>

DpUpdVarLineList <DPUVLL>

DpUpdVarTriangleMesh <DPUVTM>

DpUpdVarSimplePolygonMesh <DPUVSM>

Advantages of Using Variable Data Primitives

Variable data primitives are most useful for objects that change often. For example, a mesh object might change its vertex locations or colors dynamically. If you use a variable data primitive to create the mesh object, the object does not need to be recreated each time its data changes. The data can simply be updated with one of the primitive update calls. Variable data primitives are also useful for very large primitives, even if they are not changing often. In this case, use of a variable data primitive saves the time and space required when Doré copies the data of a standard primitive into Doré's private space.

The tradeoff, of course, is that you, the programmer, rather than Doré, are in complete control of the data for the primitive. For standard primitives, Doré manages the data. For variable data primitives, you are responsible for ensuring that the data is managed properly.

Example

The following example shows how to use the variable triangle mesh functions. The variable triangle mesh is created with the call

```
DoVarTriangleMesh (colormodel, vertexcount, vlocs, vnorms,
                   vcolors, tricount, triangles, smoothflag)
```

where

colormodel
specifies the color model used by the vertex type.

vertexcount
specifies the number of vertices in the mesh.

vlocs, vnorms, vcolors
specify separate arrays for vertex locations, normals, and colors. (In other Doré primitives, there is only one array for all three.) If vertex normals are not supplied, specify *DcNullPtr <DCNULL>* for *vnorms*. Similarly, if vertex colors are not supplied, specify *DcNullPtr <DCNULL>* for *vcolors*.

tricount
specifies the number of triangles in the mesh.

triangles
specifies the connectivity of the vertices. *triangles* is a one-dimensional array containing *tricount* triplets of integers. Each triplet is an ordered list of mesh vertex numbers for the three coordinates of the triangle.

smoothflag
specifies for Doré to calculate the geometric normal for shading purposes by averaging the geometric normals from a vertex's adjacent surfaces. This assumes that the triangle mesh defines a smooth surface.

The variable triangle mesh is updated with the call *DpUpdVarTriangleMesh <DPUVTM>*. The syntax is

```
DpUpdVarTriangleMesh (object, vlocs, vnorms, vcolors,
recompute_norms)
```

where

object
is the handle to the variable triangle mesh primitive object.

vlocs, vnorms, vcolors
are *DcNullPtr <DCNULL>* for arrays that have not changed, or a pointer to the data that has changed.

recompute_norms
specifies whether normals should be recomputed if *vlocs* is not null. The vertex normals will be recomputed only if *smoothflag* was set to true when the object was created and if the shading needs them.

This section describes how variable data primitives are currently used in Doré. The interface is, however, subject to change in future releases of Doré.

The following code example shows modifying an existing vertex location array. In this case, the update call specifies to recompute the normals. Figure 5-6 shows several variations of the variable triangle mesh used in this example.

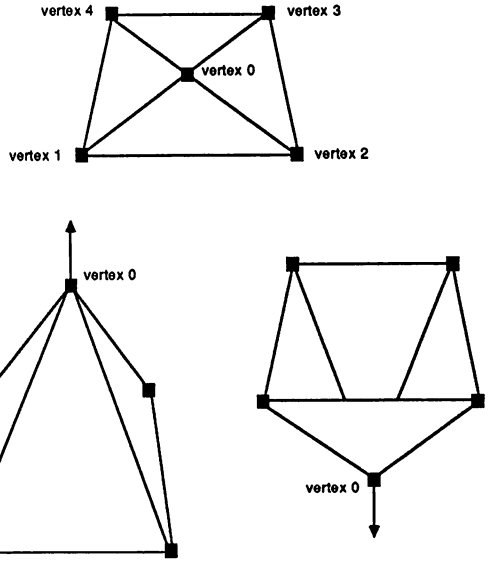


Figure 5-6. Changing Vertex Locations in a Variable Data Primitive

C code:

```
static DtRealTriple vlocs[] = {
     0.0,  0.0, 0.0,
    -1.0, -1.0, 0.0,
     1.0, -1.0, 0.0,
     1.0,  1.0, 0.0,
    -1.0,  1.0, 0.0
};
static DtRealTriple vcolors[] = {
    1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 1.0, 1.0, 0.0, 1.0
};


     DtObject vartrimsh;
      DtReal new_z_value;
      static DtInt triangles = {
         1, 2, 0, 2, 3, 0, 3, 4, 0, 4, 1, 0
      };


      vartrimsh = DoVarTriangleMesh (DcRGB, 5, vlocs, DcNullPtr, vcolors,
                  4, triangles, DcTrue);
```

```
DsHoldObj(vartrimsh);

object_group = DoGroup(DcTrue);
     DgAddObj(DoRotate(DcXAxis, DTOR(-80)));
     DgAddObj(DoRepType(DcSurface));
     DgAddObj(vartrimsh);
DgClose();

DsHoldObj(object_group);
DgAddObjToGroup(DvInqDisplayGroup(view); object_group);
DdUpdate(device);
getchar();

while (/*something*/)  {
   /* calculate new z-value for center point
      use new z-value */
   vlocs[0][2] = new_z_value;
   DpUpdVarTriangleMesh(vartrimsh, vlocs, DcNullPtr, DcNullPtr,
        DcTrue);
   DdUpdate(device);
   getchar();
   }
```

Fortran code:

```
        REAL*8 VLOCS(3,5), VCOLOR(3,5),NEWZ
        INTEGER*4 VARTRI,TRIS(3,4),OBJGRP, DEVICE
        CHARACTER*1 DUMMY
C
        PARAMETER (DEGRAD=0.0174533D0)  !CONSTANT FOR DEGREE->RADIAN
C
        DATA VLOCS / 0.0D0, 0.0D0, 0.0D0,
     X              -1.0D0,-1.0D0, 0.0D0,
     X               1.0D0,-1.0D0, 0.0D0,
     X               1.0D0, 1.0D0, 0.0D0,
     X              -1.0D0, 1.0D0, 0.0D0/
C
        DATA VCOLOR/ 1.0D0, 1.0D0, 1.0D0,
     X               1.0D0, 0.0D0, 0.0D0,
     X               0.0D0, 1.0D0, 0.0D0,
     X               0.0D0, 0.0D0, 1.0D0,
     X               1.0D0, 0.0D0, 1.0D0/
C
        DATA TRI/ 1,2,0,   2,3,0,   3,4,0,    4,1,0 /
C
C
        CALL DSINIT
        VARTRI=DOVTM(DCRGB, 5, VLOCS, DCNULL, VCOLOR, 4, TRIS, DCTRUE)
        CALL DSHO(VARTRI)
C
        OBJGRP=DOG(DCTRUE)
               CALL DGAO(DOROT(DCXAX, DEGRAD*-80.0D0))
               CALL DGAO(DOREPT(DCSURF))
               CALL DGAO(VARTRI)
        CALL DGCS
```

```
C
      CALL DSHO(OBJGRP)
      CALL DSAOG(DVQIG(VIEW),OBJGRP)
      CALL DDU(DEVICE)
      READ(5,*)DUMMY
C
10    CONTINUE          !Loop to here based on user condition
C           CALCULATE NEW Z-VALUE FOR CENTER POINT
      VLOCS(3,1)= NEW_Z_VALUE !Your computation goes here
      CALL DPUVTM(VARTRI,VLOCS,DCNULL,DCNULL,DCTRUE)
      CALL DDU(DEVICE)
      READ(5,*)DUMMY
      IF(NOT_DONE_YET) GO TO 10     !Your condition goes here
C
      CALL DSTERM
```

Chapter Summary

Chapter 5 describes the major primitive objects available in the Doré Library and how to specify them. *Primitive objects* are displayable geometric objects—such as lines, triangles, polygons, and patches. Spheres, cylinders, boxes, cones, and toruses are additional examples of primitive objects available in Doré.

Lines, triangles, polygons, and meshes all use *vertices* explicitly to compose the primitive object. Each vertex consists of three values that specify the *location* of the vertex in modeling coordinates. Depending on the vertex type, the vertex can also include a *vertex normal,* used for shading calculations, and/or a *vertex color.*

When a primitive object is executed, the *primitive attributes* immediately preceding the primitive object bind most tightly—for example, if several diffuse color objects are executed before a sphere object, the diffuse color object *lowest* in the group hierarchy but still preceding the sphere is the one that determines the sphere's color.

The *geometric normal* for a primitive object is assumed to be perpendicular to the surface of the object. Doré computes the geometric normal according to the *right-hand rule. Backface culling* is an efficiency technique in which surfaces whose geometric normals point away from the viewer are not drawn at all.

Doré provides *mesh objects,* such as *DoTriangleMesh, DoSimplePolygonMesh,* and *DoPolygonMesh <DOTRIM, DOSPM, DOPGNM>,* for specifying collections of interconnected primitive

objects. Mesh objects are an efficient way to describe collections of triangles or polygons and can be used to approximate smooth surfaces.

Simple polygons, created with *DoSimplePolygon* <*DOSPGN*>, consist of a single contour and can be convex, concave, or self-intersecting. Polygons, created with *DoPolygon* <*DOPGN*>, can have more than one contour. For polygons with more than one contour, the *inside/out rule* is used to determine whether a given point is inside or outside the polygon.

Variable data primitives, in contrast to standard primitives, do not copy vertex data into the private data space of the object. They are useful for very large primitives and for primitives whose data changes rapidly. Although variable data primitives are more efficient than standard Doré primitives, their use places an additional burden on the programmer, who is completely responsible for the management of the object data when variable data primitives are used.

PRIMITIVE ATTRIBUTES

CHAPTER SIX

This chapter describes attribute stacking in detail. It includes examples of primitive attribute objects that affect a primitive object's surface properties and color. Primitive attribute objects relating to the primitive object's display representation, such as *DoRepType <DOREPT>* and *DoSubDivSpec <DOSDS>*, are also discussed. Concepts and terms introduced in this chapter include primitive attribute objects; the pushing and popping of attributes; ambient, diffuse, specular, transparent, and reflection lighting components; subdivision level; subdivision; and representation type.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DoAmbientIntens <DOAMBI>
DoAmbientSwitch <DOAMBS>
DoDiffuseColor <DODIFC>
DoDiffuseIntens <DODIFI>
DoDiffuseSwitch <DODIFS>
DoInterpType <DOIT>
DoInvisSwitch <DOINVS>
DoLightSwitch <DOLTS>
DoPopAtts <DOPPA>
DoPushAtts <DOPUA>
DoReflectionSwitch <DOREFS>
DoRefractionIndex <DORFRI>
DoRefractionIndexSwitch <DORFRS>
DoRepType <DOREPT>
DoShadowSwitch <DOSHAS>
DoSpecularColor <DOSPCC>
DoSpecularFactor <DOSPCF>
DoSpecularIntens <DOSPCI>
DoSpecularSwitch <DOSPCS>
DoSubDivSpec <DOSDS>

Related Functions
(continued)

DoSurfaceShade <DOSRFS>
DoTranspColor <DOTC>
DoTranspIntens <DOTI>
DoTranspOrientColor <DOTOC>
DoTranspOrientExp <DOTOE>
DoTranspOrientIntens <DOTOI>
DoTranspOrientSwitch <DOTOS>
DoTranspSwitch <DOTS>

Primitive Attributes

A *primitive attribute* is an attribute used to affect how a primitive object looks. Earlier chapters introduced a number of primitive attribute objects, including *DoDiffuseColor <DODIFC>*, *DoRepType <DOREPT>*, and *DoSpecularIntens <DOSPCI>*.

The general primitive attribute category comprises several subgroups, including:

- Geometric transformation attributes (see Chapter 7)
- Text attributes (see Chapter 8)

All primitive attribute object functions begin with the prefix *Do-* and are a subcategory of the *object creation* functions. All *Doxxx* functions create an object and return its handle.

Doré Methods

In Chapters 1 through 4, we concentrated on *creating* objects and adding them to the Doré database. At object creation time (when a *Do-* call is made), the object data is stored in Doré's private database. Nothing else is actually done with the data at object creation time, and no primitives are drawn.

A *method* is a function that causes a *traversal* of the Doré database. These functions, which include rendering, picking, printing, and computing a bounding volume, use the data stored in the Doré database. Rendering, the most common method, traverses some or all of the Doré database and causes an image to be drawn.

Each object type has all methods defined for it. The phrase "to execute an object" means that some method is being applied to an object. This chapter describes what happens when the rendering method is applied to primitive attributes within a group, and how the attribute stacks are modified during a rendering traversal.

Chapter 12, "Methods," discusses the main methods in more detail.

The Rendering Method

A rendering traversal is triggered by one of the update functions (*DdUpdate* <DDU>, *DfUpdate* <DFU>, *DvUpdate* <DVU>). The rendering method for a primitive attribute object is to set the current value of that particular global attribute. The rendering method for a primitive object is to draw the stored shape into the frame buffer using all the current values for the attributes that apply to it.

The rendering method for a group object is to

(1) Save the current values of all the global attributes.

(2) Render all the objects in the group in order.

(3) Restore all the global attributes to the values they had just before the group was rendered.

The following example details this process.

Attribute Stacking

Chapter 4, "Objects and Groups" describes in general terms how attribute values are inherited within groups and how attribute values are saved and restored at group boundaries. The following paragraphs provide a conceptual explanation of how attributes stack in Doré. Actually, Doré uses a much more efficient method to handle any number of attributes with very little overhead. The effect of that method is described here.

Every attribute in Doré has a default value (see Figure 6-1), and by definition every primitive object has every primitive attribute. Certain primitive objects, however, are not affected by most of the primitive attributes. For example, *DoPolygon* is not affected by the current text font, and *DoText* is not affected by the current line type.

When a rendering traversal occurs and a regular group is entered, a copy of each current attribute value is pushed onto the top of each attribute stack, as shown in Figure 6-2. If this group is at the top of the display tree, the default values are pushed onto the top of each stack. (For convenience, this figure shows only a few of the attributes included in Doré.) Doré keeps track of the previous

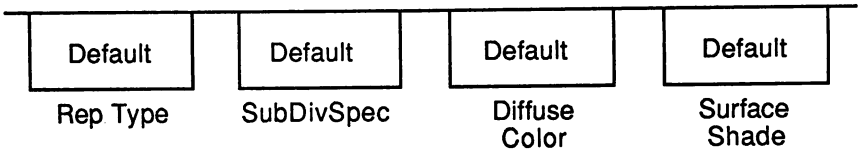


Figure 6-1. Attribute Stack

stack tops because all the pushed attribute values, along with their subsequent modifications, will be popped when that group is exited.

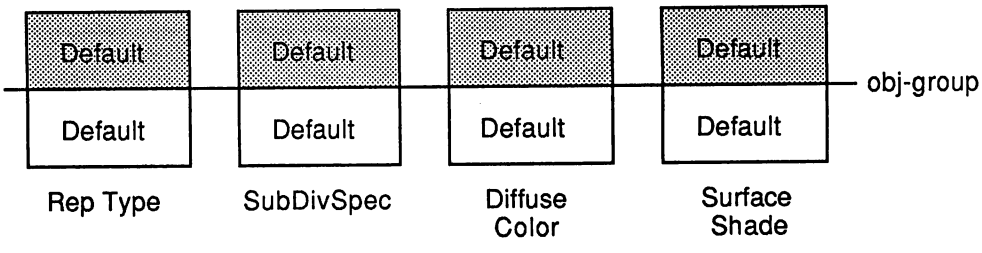


Figure 6-2. Pushing Attribute Values When a Group is Entered

Modifying the Top of Stack

This new top of stack (the copy) is now modified by subsequent primitive attribute objects in the group. For example, the following code fragment contains *DoRepType* <*DOREPT*>, *DoSubDivSpec* <*DOSDS*>, *DoDiffuseColor* <*DODIFC*>, and *DoSurfaceShade* <*DOSRFS*> objects (see Figure 6-3).

```
C code:

post = DoGroup(DcTrue);
     DgAddObj(DoDiffuseColor(DcRGB, yellow));
     DgAddObj(DoSurfaceShade(DcShaderConstant));
     DgAddObj(DoRepType(DcWireframe));
     DgAddObj(DoPrimSurf(DcCylinder));
DgClose();

obj_group = DoGroup(DcTrue);
     DgAddObj(DoRepType(DcSurface));
     DgAddObj(DoSubDivSpec(DcSubDivRelative, sds));
     DgAddObj(DoDiffuseColor(DcRGB, magenta));
     DgAddObj(DoPrimSurf(DcCone));
     DgAddObj(post);
```

```
    DgAddObj(DoPrimSurf(DcSphere));
DgClose();


Fortran code:

      POST=DOG(DCTRUE)
         CALL DGAO(DODIFC(DCRGB,YELLOW))
         CALL DGAO(DOSRFS(DCSHCN))
         CALL DGAO(DOREPT(DCWIRE))
         CALL DGAO(DOPMS(DCCYL))
      CALL DGCS()
C
      OBJGRP=DOG(DCTRUE)
         CALL DGAO(DOREPT(DCSURF))
         CALL DGAO(DOSDS(DCSDRL,SDS))
         CALL DGAO(DODIFC(DCRGB, MAGENTA))
         CALL DGAO(DOPMS(DCCONE))
         CALL DGAO(POST)
         CALL DGAO(DOPMS(DCSPHR))
      CALL DGCS()
```

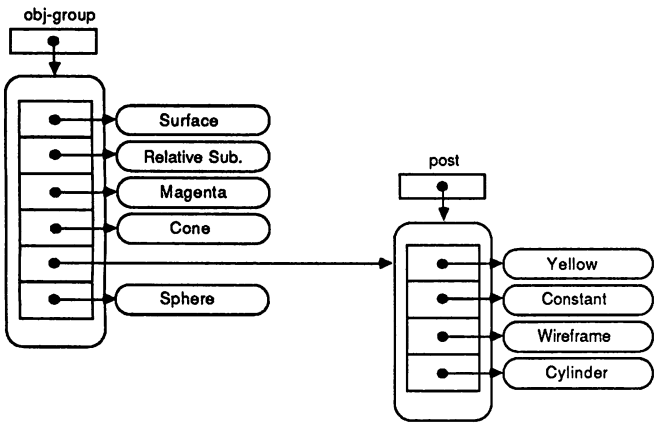


Figure 6-3. Post Group and Obj_Group

Each attribute object modifies (usually, it replaces) the existing attribute value on the top of stack. See Figure 6-4. The cone object is now executed using the current values for each global attribute. The cone has a surface representation type, a relative subdivision specification, and a diffuse color of magenta.

In contrast to most other attribute objects, geometric transformation objects modify the top of the current transformation matrix stack, *but they do not replace it*. Chapter 7, "Geometric Transformations," describes the current transformation matrix and geometric transformation objects in more detail. Name sets, filters, and the executability set are other examples of objects that modify a current top of stack but do not replace it (see Chapter 11).

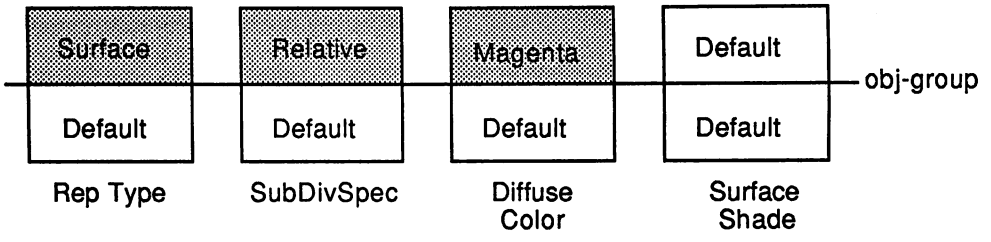


Figure 6-4. Modifying Primitive Attribute Values

Next, the post group is entered and a copy of all the current attribute stack values is made. Again, Doré keeps track of where the copied stack of attribute values begins because all the copied attribute values, along with their subsequent modifications, will be popped when the post group is exited. (See Figure 6-5.)

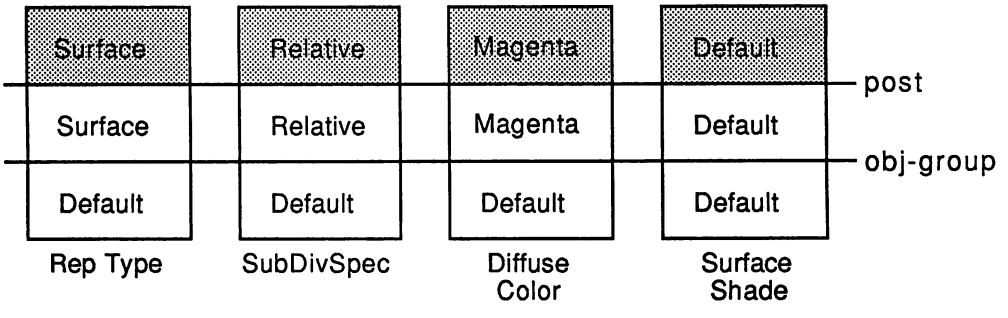


Figure 6-5. Entering the Post Group

Within the post group, the diffuse color attribute is changed to yellow, as shown in Figure 6-6. The cylinder inherits the rest of the attribute values from obj_group (including a large number of defaults not shown here).

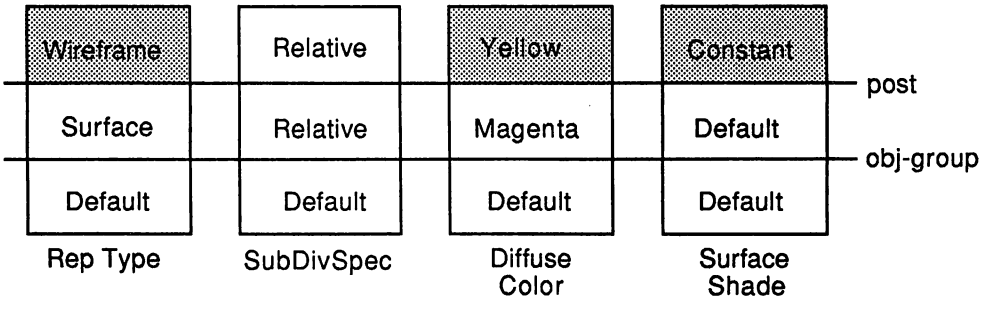


Figure 6-6. Replacing Attribute Values within the Post Group

When the post group is exited, the attribute values are popped off their stacks to the level before the post group was entered (Figure 6-7).

The sphere object is now executed using the current attribute values, including surface representation type, a relative subdivision specification, and a diffuse color of magenta. Notice how the sphere is not affected by the attributes set in the post group, since those values were popped from the attribute stacks when the post group was exited. (Notice also that the sphere and cone both inherit exactly the same primitive attributes set earlier in obj_group.)

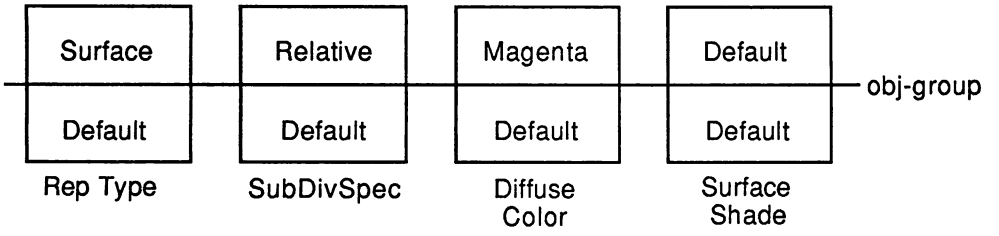


Figure 6-7. Popping Attribute Values When the Post Group is Exited

When obj_group is exited, the attribute stacks are again popped to their level before obj_group was entered (Figure 6-8).

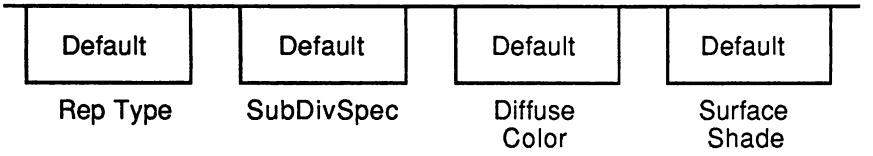


Figure 6-8. Popping Attribute Values When Obj_Group is Exited

DoPushAtts and DoPopAtts

You can also explicitly push and pop sets of attribute values using the functions *DoPushAtts* <*DOPUA*> and *DoPopAtts* <*DOPPA*> within a group. However, since attribute values are automatically pushed and popped at group boundaries, it is usually preferable to structure the Doré database so that the group structure causes attributes to push and pop "naturally."

Affecting an Object's Display Representation

The underlying three-dimensional form of any object is called its *fundamental representation.* The representation of this object as it is displayed can be referred to as its *display representation.* This display representation is affected by an object's interpolation type, its representation type, and its subdivision specification, as described below.

Representation Type: DoRepType

DoRepType <*DOREPT*> specifies how subsequent primitive objects are displayed—with corner points, edges, outlines, or solid sheets. The representation can be specified as points (*DcPoints* <*DCPNTS*>), wires (*DcWireframe* <*DCWIRE*>), polygonal outlines (*DcOutline* <*DCOUTL*>), or surfaces (*DcSurface* <*DCSURF*>).

Interpolation Type: DoInterpType

DoInterpType <*DOIT*> specifies the kind of interpolation to be used when an object is rendered. The type of shading resulting from this interpolation can be specified as

DcConstantShade <DCCNSH>
specifies to average the information for each vertex and generate a single shade for each face or edge of the object

DcVertexShade <DCVXSH>
specifies to shade each vertex of the object, then linearly interpolate the vertex shade across the object (essentially Gouraud shading)

DcSurfaceShade <DCSFSH>
specifies to interpolate all information from each vertex linearly across the object and then render each point on the object (essentially Phong shading)

Subdivision Specification: DoSubDivSpec

DoSubDivSpec <*DOSDS*> specifies how to subdivide all subsequent primitive objects. Some primitive objects, such as spheres, cylinders, toruses, and patches, are *analytic objects;* with the real-time renderer, these objects are decomposed into points, lines, and triangles that are an *approximation* of the analytic surface. The

more you divide up the surface of such objects, the closer you approximate the actual object. With the ray tracing renderer, the true mathematical object is rendered, without approximation.

The syntax for *DoSubDivSpec* is

```
DoSubDivSpec (type, parms)

DtReal parms []
```

The type of subdivision can be specified as

DcSubDivFixed <DCSDFX>
For this type, the number of segments along an edge equals 2 to the power of (parms[0]-1). In Figure 6-9, *parms[0]* is equal to 3, so 4 line segments are used to approximate the curved surface.

DcSubDivSegments <DCSDSGL>
For this type, the number of segments along an edge equals parms[0], or is as close to that number as possible.

DcSubDivAbsolute <DCSDAB>
For this type, *parms[0]* equals the maximum deviation allowed anywhere between the curved surface and the nearest plane in the planar approximation of the object. In Figure 6-10, *parms[0]* is equal to .1, which means that the curved surface cannot be more than .1 units away from any plane used to represent the curved surface.

DcSubDivRelative <DCSDRL>
For this type, *parms[0]* equals the ratio of the deviation to the length of the side of the approximating polygon. In Figure 6-11, *parms[0]* is equal to 0.03, which means that there can be no more than a 3 percent deviation between the curved surface and the approximating planar representation. In general, specifying a deviation between 1.5 percent and 4 percent for relative subdivision will result in a reasonable representation of many objects.

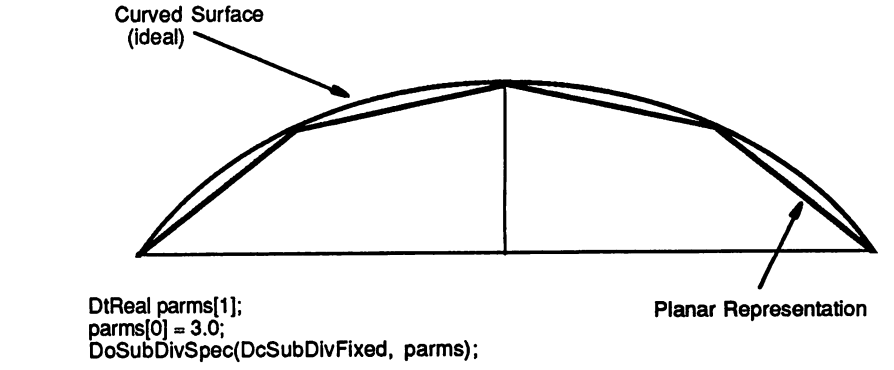


Figure 6-9. Fixed Subdivision Level

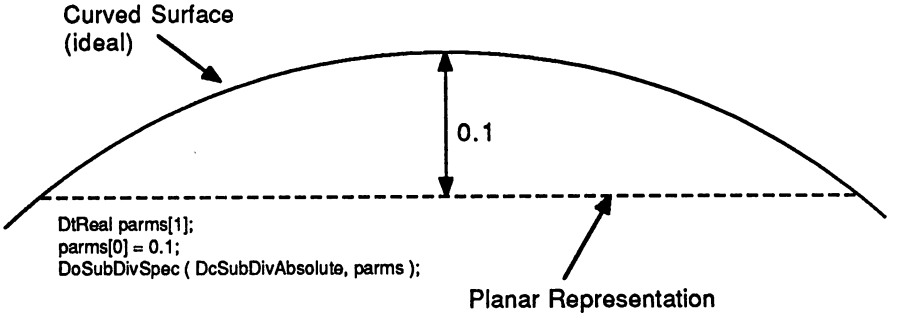


Figure 6-10. Absolute Subdivision Level

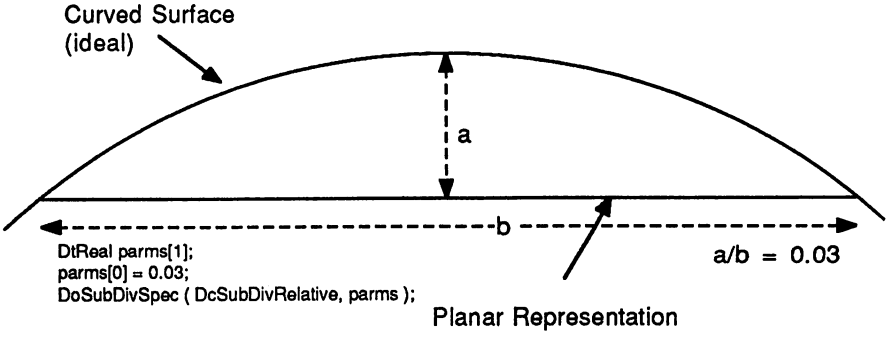


Figure 6-11. Relative Subdivision Level

The black and white car wheels in Chapter 4 use *DcWireframe* for the representation type and a different subdivision level on each wheel.

Surface Lighting Components

An object's response to light has five basic components, as shown in Figure 6-12. The basic components of surface lighting are

- Ambient component
- Diffuse component
- Specular component
- Transparent component
- Reflection component

The Doré functions that affect these components are listed in Figure 6-12. The following sections present a brief description of each surface lighting component and related functions. For more information on shading model equations, see one of the graphics programming reference books cited in the Preface.

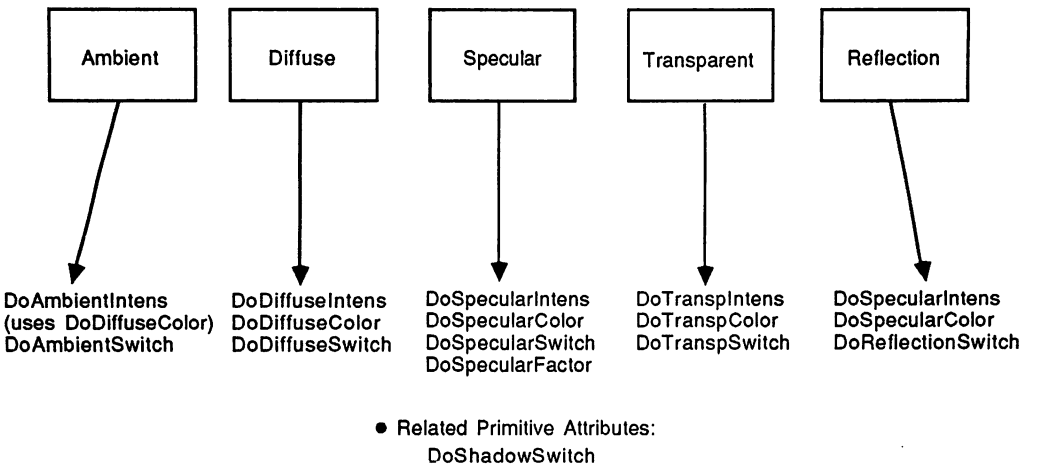


Figure 6-12. Surface Lighting Components and Related Functions

Component Switches

Each lighting component has a corresponding primitive attribute that specifies whether an object's surface properties include that component when the primitive object is executed. For example, if the diffuse switch (*DoDiffuseSwitch* <*DODIFS*>) is on (*DcOn*), subsequent primitive objects will be executed using the current values for the diffuse lighting attributes (diffuse color, diffuse intensity). However, if the diffuse switch is off, subsequent primitive objects will ignore the other diffuse lighting attributes.

Note, though, that attribute values continue to be modified even when their related component switch is off. In the following example, the diffuse color value is green and the diffuse intensity value is 0.7. The cylinder will be executed without a diffuse lighting component, however, because the switch is off. This example creates a cylinder with a green ambient color (recall that the ambient lighting component is derived from the diffuse lighting component).

C code:

```
DgAddObj(DoDiffuseColor(DcRGB, red));
DgAddObj(DoDiffuseSwitch(DcOff));
DgAddObj(DoDiffuseColor(DcRGB, green));
DgAddObj(DoDiffuseIntens(.7));
DgAddObj(DoPrimSurf(DcCylinder));
```

Fortran code:

```
      CALL DGAO(DODIFC(DCRGB, RED))
      CALL DGAO(DODIFS(DCOFF))
      CALL DGAO(DODIFC(DCRGB, GREEN))
      CALL DGAO(DODIFI(0.7D0))
      CALL DGAO(DOPMS(DCCYL))
```

Intensity Attributes

Each lighting component also has a corresponding intensity attribute object that sets its value (*DoAmbientIntens* <*DOAMBI*>, *DoDiffuseIntens* <*DODIFI*>, *DoSpecularIntens* <*DOSPCI*>, and *DoTranspIntens* <*DOTI*>). Intensity values, usually between 0.0 and 1.0, are weighting factors for each lighting component. As a general rule, the sum of the diffuse intensity and specular intensity values should be between 0.0 and 1.0. The ambient intensity should usually be small (between 0.0 and 0.4). For a particular effect, however, you may want to experiment with intensity values and deviate from these typical values.

Diffuse Lighting Component

The diffuse lighting component refers to the response of a surface to incident (non-ambient) light, where the reflected light is scattered equally in all directions. *DoDiffuseColor* <*DODIFC*> specifies an object's base color.

Ambient Lighting Component

The ambient lighting component refers to the object's response to ambient light. Ambient light, like diffuse light, is light given off equally in all directions, but without regard to light positions. The diffuse color represents the object's base color; the value for diffuse color is used to calculate the ambient color as well.

If rendering speed is important, you can turn off the specular lighting component and use only the ambient and diffuse lighting components. Turn off the ambient component if you want the object to appear as if it is in outer space. Portions that are not otherwise lit will then appear completely black.

The food processor examples in this chapter have the ambient lighting component turned on: *DoAmbientSwitch(DcOn)* <*DOAMBS(DCON)*>.

Specular Lighting Component

The specular lighting component refers to the object's highlights. Like the other surface lighting components, specular lighting has a color and an intensity value. In addition, specular lighting has a *specular factor*, which controls the size of specular highlights. A specular factor of 0 indicates a matte finish that has no specular highlights. The higher the specular factor, the smaller and sharper the highlights become.

The *DoSpecularColor* <*DOSPCC*> object specifies the highlight color of a surface in response to light from directed light sources. The specular color can be thought of as the color of the object's shiny highlights. For example, the highlight on a plastic surface appears white, as in the food processor shown in Plate 7, which has specular highlights along the rim of the bowl. The highlights on a metallic surface are the same color as the non-highlighted (diffuse) color of the metal.

DoSpecularFactor <*DOSPCF*> creates an appearance object that specifies the sharpness of the specular reflection of subsequent primitive objects. Values closer to 0 result in fuzzy, less precise highlights. High values (e.g., 200) result in sharp, precise highlights.

The wheels and axle in Plate 1 have a specular lighting component.

Transparent Lighting Component

The transparent lighting component refers to the light being transmitted through a surface. The food processor example shown in Plate 7 has a transparent bowl.

DoTranspColor <*DOTC*> specifies the *filter color* of a particular surface. Using the *DcRGB* color model, a red object that is behind a transparent object with its red component having a value of 0.0 and its blue and green components each having values of 1.0 would appear black, since no red light is transmitted through the object. A yellow object would appear green when viewed from behind this object, as shown in Figure 6-13.

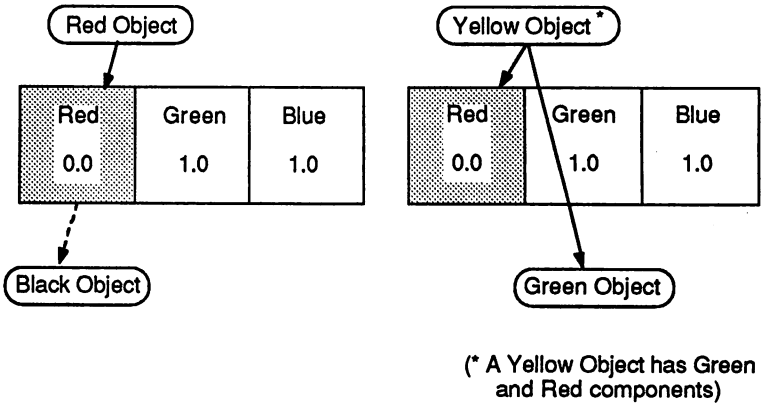


Figure 6-13. Transparent (Filter) Color

Reflection Lighting Component

Reflection is an object's response to light reflected from primitive objects and directly from light sources. In determining this response, the specular color and specular intensity are used, as shown in Figure 6-12.

Example

The following example produces an image of a food processor. Each part of the food processor is specified using a series of patch objects (omitted from this example for the sake of brevity). The arrays defining each color are included at the beginning of the example.

The code shown here produces the image shown in Plate 7, with the following attributes:

- Representation type is *DcSurface* <*DCSURF*>
- Interpolation type is *DcSurfaceShade* <*DCSFSH*>
- The ambient switch is on, which enables ambient light
- The diffuse switch is on, which enables diffuse light
- The specular switch is on, which enables specular light
- The transparent switch is on, which enables transparent light
- The counter top is backface-culled (*DoBackfaceCullable* <*DOBFC*> and *DoBackfaceCullSwitch* <*DOBFCS*> are on)
- For the other objects, the backface culling switch is off, so that the backfacing surfaces on the other objects are rendered

It's Easy to Change the Image

By changing values for only a few primitive attributes, you can easily change the rendering of a particular image.

- To produce a wireframe food processor image, specify *DcWireframe* <*DCWIRE*> for *DoRepType*.
- To eliminate the specular component, change the value of *DoSpecularSwitch* <*DOSPCS*> to *DcOff*.

- For faster rendering, specify *DcRealTime <DCRLTM>* for *DvSetRendStyle <DVSRS>*. (The shadows and reflections are ignored with the real-time renderer.)

C code:

```
DtObject button1, button2, button3, base, counter, blade,
        cutter, plunger, pot, shaft, plate, object_group;
DtObject BUTTON, BUTTON2, BUTTON3, BASE, BLADE, CUTTER,
        PLUNGER, POT, SHAFT, PLATE; /* patch objects */
static DtReal
     yellow[] = {1.0, 1.0, 0.0},
     orange[] = {1.0, 0.8, 0.0},
     greenishyellow[] = {0.8, 1.0, 0.0},
     white[] = {1.0, 1.0, 1.0},
     grey[] = {1.0, 0.8, 0.9},
     blue_grey[] = {0.6, 0.7, 1.0},
     green_blue[] = {0.5, 0.5, 1.0},
     dull_white[] = {0.2, 0.2, 0.2},
     parms[] = {.03};

button1 = DoGroup(DcTrue);  /* creates and opens button1 group */
     DgAddObj(DoDiffuseColor(DcRGB, yellow));
     DgAddObj(DoSpecularColor(DcRGB, yellow));
     DgAddObj(DoTranspColor(DcRGB, yellow));
     DgAddObj(BUTTON);       /* patch objects to make button1 */
DgClose();

button2 = DoGroup(DcTrue);  /* creates and opens button2 group */
     DgAddObj(DoDiffuseColor(DcRGB, greenishyellow));
     DgAddObj(DoSpecularColor(DcRGB, greenishyellow));
     DgAddObj(DoTranspColor(DcRGB, greenishyellow));
     DgAddObj(BUTTON2);  /* patch objects to make button2 */
DgClose();

button3 = DoGroup(DcTrue);  /* creates and opens button3 group */
     DgAddObj(DoDiffuseColor(DcRGB, greenishyellow));
     DgAddObj(DoSpecularColor(DcRGB, greenishyellow));
     DgAddObj(DoTranspColor(DcRGB, greenishyellow));
     DgAddObj(BUTTON3);  /* patch objects to make third button
group */
DgClose();

base = DoGroup(DcTrue);     /* creates and opens base group */
     DgAddObj(DoDiffuseColor(DcRGB, white));
     DgAddObj(DoSpecularColor(DcRGB, white));
     DgAddObj(BASE));    /* patch objects to make the base */
DgClose();

counter = DoGroup(DcTrue);   /*creates and opens counter group */
     DgAddObj(DoDiffuseColor(DcRGB, grey));
     DgAddObj(DoDiffuseIntens(0.8));
     DgAddObj(DoReflectionSwitch(DcOn));
     DgAddObj(DoSpecularColor(DcRGB, grey));
     DgAddObj(DoSpecularIntens(1.0));
```

```
        DgAddObj(DoSpecularSwitch(DcOn));
        DgAddObj(DoTranslate(0.0, -3.5, 0.0));  /* lowers slab */
        DgAddObj(DoScale(12.0, 1.0, 12.0));   /* stretches it out */
        DgAddObj(DoTranslate(-0.5, -0.5, -0.5))  /* to origin */
        DgAddObj(DoPrimSurf(DcBox));
DgClose();

blade = DoGroup(DcTrue); /* creates and opens the blade group */
        DgAddObj(DoDiffuseColor(DcRGB, blue_grey));
        DgAddObj(DoSpecularColor(DcRGB, blue_grey));
        DgAddObj(BLADE);     /* patch objects to make the blades */
DgClose();

cutter = DoGroup(DcTrue); /* creates and opens cutter group */
        DgAddObj(DoDiffuseColor(DcRGB, white));
        DgAddObj(DoSpecularColor(DcRGB, white));
        DgAddObj(DoDiffuseIntens(1.0));
        DgAddObj(DoSpecularIntens(1.0));
        DgAddObj(CUTTER);    /* patch objects to make the cutter */
DgClose();

plunger = DoGroup(DcTrue); /* creates and opens plunger group */
        DgAddObj(DoDiffuseColor(DcRGB, white));
        DgAddObj(DoSpecularColor(DcRGB, white));
        DgAddObj(PLUNGER);  /* patch objects to make the plunger */
DgClose();

pot = DoGroup(DcTrue);    /* creates and opens the pot group */
        DgAddObj(DoTranspSwitch(DcOn)); /* turns on transparency */
        DgAddObj(DoDiffuseColor(DcRGB, white));
        DgAddObj(DoSpecularColor(DcRGB, white));
        DgAddObj(DoTranspColor(DcRGB, white));
        DgAddObj(DoDiffuseIntens(0.2));
        DgAddObj(DoTranspIntens(0.7));
        DgAddObj(POT);       /* patch objects to make the pot */
DgClose();

shaft = DoGroup(DcTrue); /* creates and opens the shaft group */
        DgAddObj(DoDiffuseColor(DcRGB, green_blue));
        DgAddObj(DoSpecularColor(DcRGB, green_blue));
        DgAddObj(SHAFT);     /* patch objects to make the shaft */
DgClose();

plate = DoGroup(DcTrue); /*creates and opens the plate group */
        DgAddObj(DoDiffuseColor(DcRGB, dull_white));
        DgAddObj(DoSpecularColor(DcRGB, dull_white));
        DgAddObj(PLATE);     /* patch object to make the plate */
DgClose();

object_group = DoGroup(DcTrue);/*creates and opens object group*/
        DgAddObj(DoRotate(DcYAxis, -0.76));
        DgAddObj(DoRepType(DcSurface));
        DgAddObj(DoInterpType(DcSurfaceShade));
        DgAddObj(DoAmbientSwitch(DcOn));
        DgAddObj(DoDiffuseSwitch(DcOn));
        DgAddObj(DoSpecularSwitch(DcOn));
```

Example
(continued)

```
    DgAddObj(DoTranspSwitch(DcOn));
    DgAddObj(DoShadowSwitch(DcOn));
    DgAddObj(DoReflectionSwitch(DcOn));
    DgAddObj(DoSubDivSpec(DcSubDivRelative, parms));
    DgAddObj(DoBackfaceCullSwitch(DcOn));
    DgAddObj(DoBackfaceCullable(DcOn));
    DgAddObj(counter);
    DgAddObj(base);
    DgAddObj(button1);
    DgAddObj(button2);
    DgAddObj(button3);
    DgAddObj(blade);
    DgAddObj(plate);
    DgAddObj(cutter);
    DgAddObj(shaft);
    DgAddObj(plunger);
    DgAddObj(DoBackfaceCullSwitch(DcOff));
    DgAddObj(pot);
DgClose();

Fortran code:

      REAL*4 BTN1, BTN2, BTN3, BASE, CNTR, BLADE,
     1       CTR, PLNGR, POT, SHAFT, PLATE, OBJGRP
      REAL*4 B1, B2, B3, BA, BLD, CUT, PLUNG, PT, SHFT, PLAT
C            PATCH OBJECTS
      REAL*8 YELLOW(3), ORANGE(3), GRNYEL(3), WHITE(3) GREY(3),
     1       BLUGRY(3), GREBLU(3), DULWHI(3), PARMS(1)
      CHARACTER DUMMY
C
      DATA YELLOW / 1.0D0, 1.0D0, 0.0D0 /
      DATA ORANGE / 1.0D0, 0.8D0, 0.0D0 /
      DATA GRNYEL / 0.8D0, 1.0D0, 0.0D0 /
      DATA WHITE  / 1.0D0, 1.0D0, 1.0D0 /
      DATA GREY   / 1.0D0, 0.8D0, 0.9D0 /
      DATA BLUGRY / 0.6D0, 0.7D0, 1.0D0 /
      DATA GREBLU / 0.5D0, 0.5D0, 1.0D0 /
      DATA DULWHI / 0.2D0, 0.2D0, 0.2D0 /
      DATA PARMS  / .03 /
C
      BTN1=DOG(DCTRUE) ! creates and opens button1 group !
         CALL DGAO(DODIFC(DCRGB, YELLOW))
         CALL DGAO(DOSPCC(DCRGB, YELLOW))
         CALL DGAO(DOTC(DCRGB, YELLOW))
         CALL DGAO(B1) ! patch objects to make button1 !
      CALL DGCS()
C
      BTN2=DOG(DCTRUE) ! creates and opens button2 group !
         CALL DGAO(DODIFC(DCRGB, GRNYEL))
         CALL DGAO(DOSPCC(DCRGB, GRNYEL))
         CALL DGAO(DOTC(DCRGB, GRNYEL))
         CALL DGAO(B2) ! patch objects to make button2 !
      CALL DGCS()
C
      BTN3=DOG(DCTRUE) ! creates and opens button3 group!
         CALL DGAO(DODIFC(DCRGB, GRNYEL))
```

```
      CALL DGAO(DOSPCC(DCRGB, GRNYEL))
      CALL DGAO(DOTC(DCRGB, GRENYEL))
      CALL DGAO(B3) ! patch objects to make button3!
   CALL DGCS()
C
   BASE=DOG(DCTRUE) ! creates and opens base group!
      CALL DGAO(DODIFC(DCRGB, WHITE))
      CALL DGAO(DOSPCC(DCRGB, WHITE))
      CALL DGAO(BA) ! patch objects to make the base!
   CALL DGCS()
C
   CNTR=DOG(DCTRUE) ! creates and opens the counter group!
      CALL DGAO(DODIFC(DCRGB, GREY))
      CALL DGAO(DODIFI(0.8D0))
      CALL DGAO(DOREFS(DCON))
      CALL DGAO(DOSPCC(DCRGB, GREY))
      CALL DGAO(DOSPCI(1.0D0))
      CALL DGAO(DOSPCS(DCON))
      CALL DGAO(DOXLT(0.0D0, -3.5D0, 0.0D0)) ! lowers slab!
      CALL DGAO(DOSC(12.0D0, 1.0D0, 12.0D0)) ! stretches it !
      CALL DGAO(DOXLT(-0.5D0, -0.5D0, -0.5D0)) ! to origin !
      CALL DGAO(DOPMS(DCBOX))
   CALL DGCS()
C
   BLADE=DOG(DCTRUE) ! creates and opens the blade group !
      CALL DGAO(DODIFC(DCRGB, BLUGRY))
      CALL DGAO(DOSPCC(DCRGB, BLUGRY))
      CALL DGAO(BLD) ! patch objects to make the blades!
   CALL DGCS()
C
   CTR = DOG(DCTRUE) ! creates and opens cutter group !
      CALL DGAO(DODIFC(DCRGB, WHITE))
      CALL DGAO(DOSPCC(DCRGB, WHITE))
      CALL DGAO(DODIFI(1.0D0))
      CALL DGAO(DOSPCI(1.0D0))
      CALL DGAO(CUT) ! patch objects to make the cutter !
   CALL DGCS()
C
   PLNGR = DOG(DCTRUE) ! creates and opens plunger group!
      CALL DGAO(DODIFC(DCRGB, WHITE))
      CALL DGAO(DOSPCC(DCRGB, WHITE))
      CALL DGAO(PLUNG) ! patch objects to make the plunger !
   CALL DGCS()
C
   POT = DOG(DCTRUE) ! creates and opens the pot group !
      CALL DGAO(DOTS(DCON))
      CALL DGAO(DODIFC(DCRGB, WHITE))
      CALL DGAO(DOSPCC(DCRGB, WHITE))
      CALL DGAO(DOTC(DCRGB, WHITE))
      CALL DGAO(DODIFI(0.2D0))
      CALL DGAO(DOTI(0.7D0))
      CALL DGAO(PT) ! patch objects to make the pot !
   CALL DGCS()
C
   SHAFT = DOG(DCTRUE) ! creates and opens the shaft group !
      CALL DGAO(DODIFC(DCRGB, GREBLU))
```

Example
(continued)

```
      CALL DGAO(DOSPCC(DCRGB, GREBLU))
      CALL DGAO(SHFT) ! patch objects to make the shaft !
   CALL DGCS()
C
   PLATE = DOG(DCTRUE) ! creates and opens the plate group !
      CALL DGAO(DODIFC(DCRGB, DULWHI))
      CALL DGAO(DOSPCC(DCRGB, DULWHI))
      CALL DGAO(PLAT) ! patch objects to make the plate !
C
   OBJGRP = DOG(DCTRUE) ! creates and opens the object group !
      CALL DGAO(DOROT(DCYAX, -0.76D0))
      CALL DGAO(DOREPT(DCSURF))
      CALL DGAO(DOIT(SFSH))
      CALL DGAO(DOAMBS(DCON))
      CALL DGAO(DODIFS(DCON))
      CALL DGAO(DOSPCS(DCON))
      CALL DGAO(DOTS(DCON))
      CALL DGAO(DOSHAS(DCON))
      CALL DGAO(DOREFS(DCON))
      CALL DGAO(DOSDS(DCSDRL, PARMS))
      CALL DGAO(DOBFCS(DCON))
      CALL DGAO(DOBFC(DCON))
      CALL DGAO(CNTR)
      CALL DGAO(BASE)
      CALL DGAO(BTN1)
      CALL DGAO(BTN2)
      CALL DGAO(BTN3)
      CALL DGAO(BLADE)
      CALL DGAO(PLATE)
      CALL DGAO(CTR)
      CALL DGAO(SHAFT)
      CALL DGAO(PLNGR)
      CALL DGAO(DOBFCS(DCOFF))
      CALL DGAO(POT)
   CALL DGCS()
```

Chapter Summary

Chapter 6 describes *primitive attributes*, which are attributes used to affect how primitive objects look—for example, their surface properties, color, and display representation.

When a primitive object is executed, it uses the current value for each primitive attribute. Attribute stacks are pushed and popped at group boundaries. Within a group, primitive attribute objects modify the existing attribute value on the top of the stack. (Usually, the primitive attribute object *replaces* the previous value with a new value—for example, a *DoDiffuseColor* <*DODIFC*> with a value of *green* replaces the previous diffuse color value.)

You can also push and pop sets of attribute values using the functions *DoPushAtts* <*DOPUA*> and *DoPopAtts* <*DOPPA*> within a

group. It is usually preferable, however, to structure the Doré database so that the group structure causes attributes to push and pop "naturally."

An object's *display representation* is affected by the object's representation type, interpolation type, and its subdivision specification. An object's *representation type* can be specified as points, wireframe, polygonal outlines, or surfaces. Its *interpolation type* specifies the type of shading that results from the interpolation used when the object is rendered. The object's *subdivision specification* specifies how to subdivide primitive objects that are an approximation of an analytic surface.

An object's response to light has five basic components: ambient, diffuse, specular, transparent, and reflection. The *diffuse* lighting component refers to the response of a surface to incident (non-ambient) light, where the reflected light is scattered equally in all directions. The *ambient* lighting component refers to the object's response to ambient light, which is light that is everywhere in the scene, independent of light sources. The *specular* lighting component refers to the object's highlights. The *transparent* lighting component refers to the light transmitted through a surface. The transparent color is essentially the *filter* color of a particular surface.

Each of these surface lighting components has a primitive attribute specifying intensity (for example, *DoSpecularIntens* <*DOSPCI*>), one that specifies color (for example, *DoDiffuseColor* <*DODIFC*>), and one that specifies whether an object's surface properties include that component (for example, *DoDiffuseSwitch* <*DODIFS*>, *DoAmbientSwitch* <*DOAMBS*>).

GEOMETRIC TRANSFORMATIONS

CHAPTER SEVEN

This chapter introduces geometric transformation attribute objects, which can be used to affect subsequent primitive objects, as well as cameras and lights (see Chapter 9). The current transformation matrix stack is discussed, as well as the use of *DoPushMatrix* <DOPUMX> and *DoPopMatrix* <DOPPMX> to affect the top of stack. Concepts and terms introduced in this chapter include translating, scaling, rotating, and shearing objects; right-handed coordinate systems; relative coordinate systems; preconcatenation; modeling coordinates; and relative and absolute group definitions.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DoLookAtFrom <DOLAF>
DoPopMatrix <DOPPMX>
DoPushMatrix <DOPUMX>
DoRotate <DOROT>
DoScale <DOSC>
DoShear <DOSHR>
DoTransformMatrix <DOTMX>
DoTranslate <DOXLT>

Geometric Transformations

A *geometric transformation* attribute object affects the shape and positioning of primitive or studio objects in 3-dimensional space. *Scaling* a primitive object affects its relative size by shrinking or stretching it. *Translating* a primitive object moves its relative position in modeling space. *Rotating* a primitive object turns the object around a particular modeling axis. *Shearing* a primitive object displaces the coordinates in two dimensions proportional to their distance from one of the three major planes.

Modeling Coordinates

Each object is defined in its own *modeling space*, and you can use whatever coordinates are best suited to the particular object. (Another name for modeling coordinates is *local coordinates.*) One object might be defined in units from 0.0 to 1.0, and another might be defined in units from -112.56 to 112.56.

Geometric transformations combine with each other. For example, two consecutive rotations about the same axis add together. The cumulative effect of geometric transformations enables you to build hierarchies of modeling spaces. In the car wheel example in Chapter 4, the wheel group was defined in its own modeling space, then rotated and translated into the left and right wheel groups. The left and right wheel groups were then rotated and translated into position on the axle. If the example were extended further, the axle would next be rotated, scaled, and translated into position on the car body. Finally, the car body would be transformed into position in the final scene. This final scene, in which all scene elements are positioned in relation to each other, is said to be defined in terms of the *world coordinate system.*

Right-Handed Coordinate System

Doré uses right-handed coordinate systems. For example, numbers to the right, up, and forward (out of the screen) would be *positive,* and numbers to the left, down, and back (into the screen) would be *negative.* In a right-handed coordinate system, a positive rotation about a particular axis is found by pointing your right thumb in the positive direction of the axis and curling your fingers around the axis. The direction of the curl of your fingers is a positive rotation. In a right-handed system, a positive rotation about the x axis rotates an object that points up (in the positive y direction) out of the screen towards the viewer.

Current Transformation Matrix (CTM)

Geometric transformation attribute objects all modify the *current transformation matrix (CTM).* This four-by-four matrix is a compact way of representing transformations and allows arbitrary scaling and positioning of objects in 3D space. The graphics programming reference books cited in the Preface provide detailed explanations and examples of the transformation matrix. In general, however, you do not need to know about the form of this matrix; you need know only what transformations have gone into it, and in what order.

In Doré, the CTM is pushed and popped at group boundaries. Within a group, however, successive geometric transformations are *concatenated* together onto the CTM.

Ordering of Geometric Transformations

When a geometric transformation attribute object is executed, the associated matrix is preconcatenated onto the CTM. (The one current exception is *DoTransformMatrix* <*DOTMX*>, which can be preconcatenated or postconcatenated with the CTM, or can replace the CTM entirely.) **Transformations are preconcatenated, which means that the last transformation executed is the first one to affect following primitive objects.** You thus need to read backwards through the code from a given primitive object to see the actual order in which its transformations are *applied* to the object. (This process is consistent with other attributes in that attributes specified closest to the object bind most tightly.) The figures and examples below help you become accustomed to this process.

Relative Coordinate System

In Doré, the orientation and origin of the coordinate system are affected each time a geometric transformation takes place. For example

```
C code:

DoTranslate(2.0, 1.0, 0.0);
DoPrimSurf(DcSphere);


Fortran code:

CALL DOXLT(2.0D0, 1.0D0, 0.0D0)
CALL DOPMS(DCSPHR)
```

moves the sphere to the new location (2.0, 1.0, 0.0), as shown in Figure 7-1.

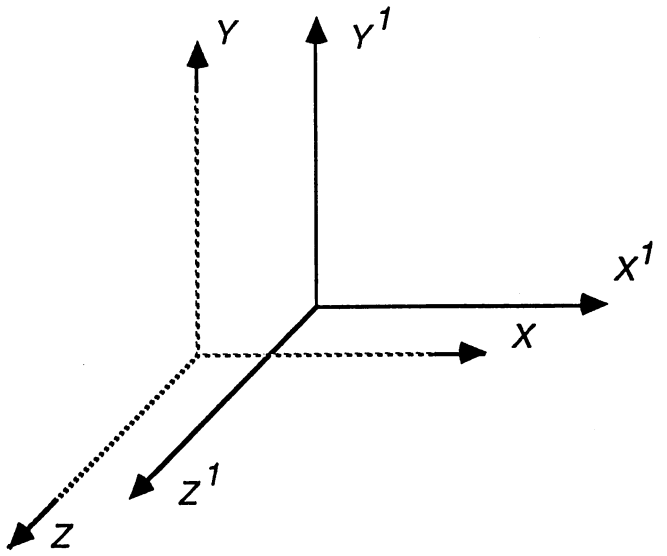


Figure 7-1. Translating a Primitive Object

```
C code:

DoTranslate(3.0, 3.0, 0.0);
DoTranslate(2.0, 1.0, 0.0);
DoPrimSurf(DcSphere);


Fortran code:

      CALL DOXLT(3.0D0, 3.0D0, 0.0D0)
      CALL DOXLT(2.0D0, 1.0D0, 0.0D0)
      CALL DOPMS(DCSPHR)
```

The next *DoTranslate* is made relative to that new position. If another translate object were added the sphere is now translated to location (2.0, 2.0, 0.0), then to a second location that is +3 units in the x and y directions. Actually, since translations are commutative, the two translations are simply added together and the sphere is moved to (5.0, 4.0, 0.0).

After each geometric transformation, the coordinate system is changed. For example, if the sphere is then rotated 45 degrees around the y axis, the coordinate system appears as shown in Figure 7-2.

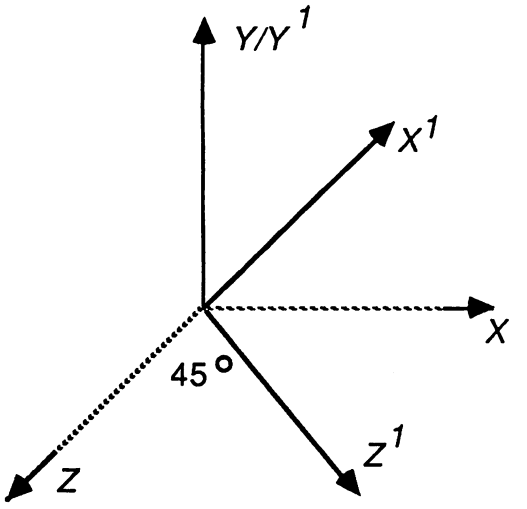


Figure 7-2. Rotating a Primitive Object

Example

The following example and figures illustrate how the matrices for successive geometric transformations are preconcatenated onto the CTM.

C code:

```
#define PI 3.14159265

cone_group=DoGroup(DcTrue);
     DgAddObj(DoTranslate(2.0, 2.0, -2.0));  /* moves the cone */
     DgAddObj(DoScale(1.0, 2.0, 1.0));       /* makes the cone taller */
     DgAddObj(DoRotate(DcXAxis, -PI/2));     /* stands the cone up */
     DgAddObj(DoPrimSurf(DcCone));
DgClose();
```

Fortran code:

```
      CONEGR = DOG(DCTRUE)
         CALL DGAO(DOXLT(2.0D0, 2.0D0, -2.0D0))
         CALL DGAO(DOSC(1.0D0, 2.0D0, 1.0D0))
         CALL DGAO(DOROT(DCXAX, -1.57D0))
         CALL DGAO(DOPMS(DCCONE))
      CALL DGCS()
```

Figure 7-3 shows the CTM when the cone group is entered. In the figure, "PTM" is simply the previous transformation matrix that is inherited by the group.

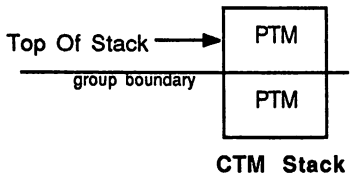


Figure 7-3. Entering the Cone Group

First, the translate object is executed (see Figure 7-4, below). **In the following figures, the transformations accumulated in the CTM are read from left to right to see the order in which their effects are felt.**

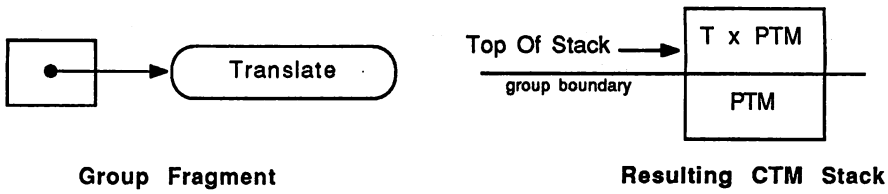


Figure 7-4. Preconcatenating the Translation Transformation

Next the scale object is executed. As shown in Figure 7-5 below, the scale transformation is preconcatenated with the current transformation matrix, which already includes the translation transformation.

Finally, the rotate object is executed and its associated matrix is preconcatenated with the current transformation matrix. See Figure 7-6.

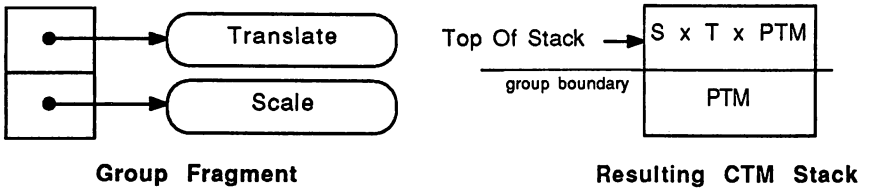


Figure 7-5. Preconcatenating the Scale Transformation

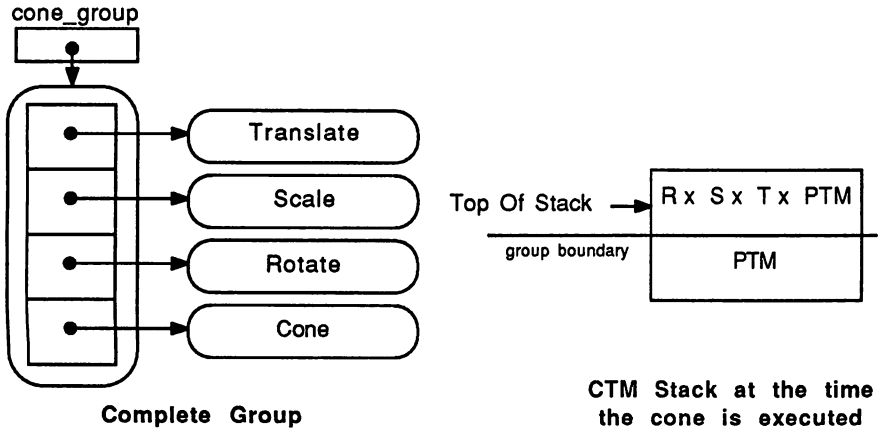


Figure 7-6. Preconcatenating the Rotation Transformation

When the cone group is exited, the top of the CTM stack is popped, and the original value is restored, as shown in Figure 7-7.

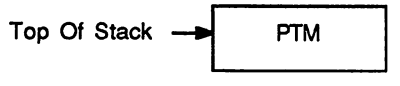


Figure 7-7. Exiting the Group and Popping the Stack

Ordering Transformations on the Stack

The order in which transformations are placed on the stack is significant. If you scaled the cone *before* you rotated it, instead of making the cone taller you would have stretched out its base from a circle into an ellipse. In addition, as shown in the figures above, as each new object is executed, its transformation is appended to the CTM in such a way that its effect is felt *first*. In this example, the cone is rotated, then scaled, then translated. The resulting cone is shown in Figure 7-8, below.

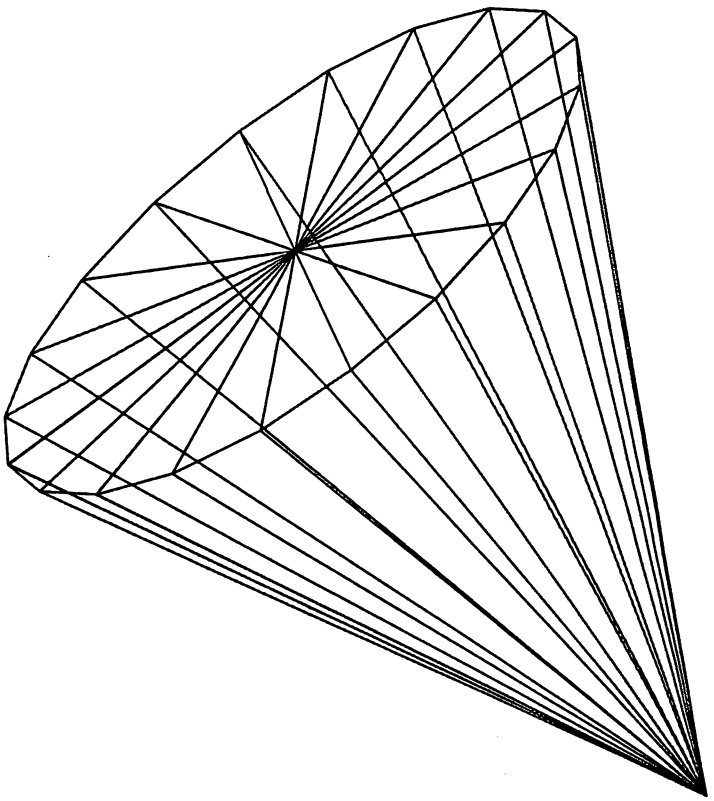


Figure 7-8. Cone after Geometric Transformations

Probably the best way to avoid confusion is to follow the general order of scaling closest to the object, then rotating, then translating. This way, the x, y, z axes used to model the original object will be used by the scale object. The order in your program would thus be

1. translate
2. rotate

3. scale
4. primitive

Pushing and Popping the CTM

Sometimes, however, you will want to control pushing and popping of the CTM stack *within* a group. The *DoPushMatrix* <*DOPUMX*> and *DoPopMatrix* <*DOPPMX*> functions provide you with this control. The car wheel example in Chapter 4 illustrates a simple use of this function pair.

C code:

```
wheel_group=DoGroup(DcTrue);
     DgAddObj(DoDiffuseColor(DcRGB, green));
     DgAddObj(DoPushMatrix());
          DgAddObj(DoRotate(DcXAxis, DTOR(90)));
          DgAddObj(DoTorus(1.0, 0.3));
     DgAddObj(DoPopMatrix());
     DgAddObj(DoScale(1.0, 1.0, 0.4));
     DgAddObj(DoTranslate(0.0, 0.0, -0.5));
     DgAddObj(DoPrimSurf(DcCylinder));
DgClose();
```

Fortran code:

```
      WHEEL=DOG(DCTRUE)
         CALL DGAO(DODIFC(DORGB, GREEN))
         CALL DGAO(DOPUMX())
            CALL DGAO(DOROT(DCXAX, (DEGRAD*90.0D0)))
            CALL DGAO(DOTOR(1.0D0, 0.3D0))
         CALL DGAO(DOPPMX())
         CALL DGAO(DOSC(1.0D0, 1.0D0, 0.4D0))
         CALL DGAO(DOXLT(0.0D0, 0.0D0, -0.5D0))
         CALL DGAO(DOPMS(DCCYL))
      CALL DGCS()
```

The addition of *DoPushMatrix* <*DOPUMX*> and *DoPopMatrix* <*DOPPMX*> affect the top of stack significantly, as shown below. First, the wheel group is entered, which causes the previous transformation matrix to be pushed onto the CTM stack. Then DoPushMatrix pushes this matrix again, as shown in Figure 7-9.

The rotate transformation is preconcatenated with the previous transformation matrix, and the torus is executed (Figure 7-10).

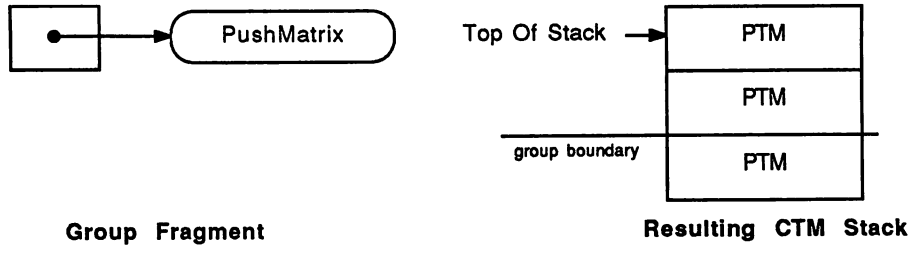


Figure 7-9. Entering the Wheel Group and Pushing the Matrix

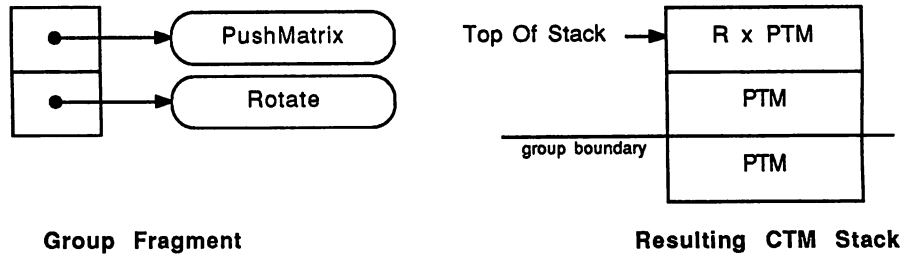


Figure 7-10. Preconcatenating the Rotation Transformation

Next the matrix is popped with *DoPopMatrix* (Figure 7-11).

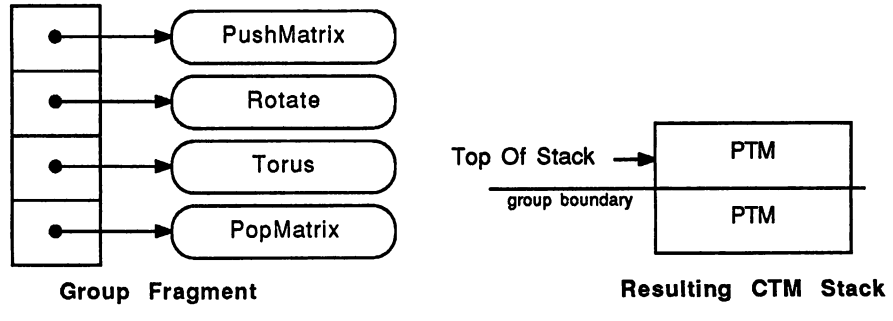


Figure 7-11. Popping the Matrix

The scale transformation is preconcatenated with the previous transformation matrix, then the translation (Figure 7-12). Then the cylinder is executed.

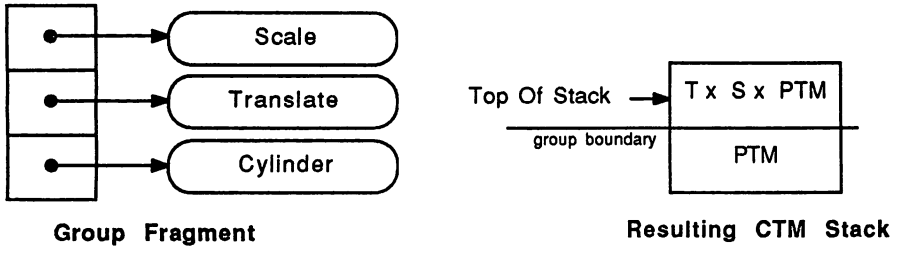


Figure 7-12. Preconcatenating the Scale and Translation Transformations

When the wheel group is exited, the top of the CTM stack is automatically popped again (Figure 7-13), returning it to the state before the wheel group was entered.

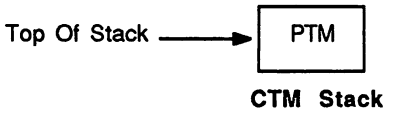


Figure 7-13. Exiting the Group and Popping the Matrix

DoLookAtFrom

Although *DoLookAtFrom* <*DOLAF*> is used most often to position cameras and lights, it is also a useful geometric transformation for positioning primitive objects relative to each other. For example, you could model an airplane with its nose at the origin and pointing into negative Z. Then it could be positioned in the scene with a single *DoLookAtFrom* <*DOLAF*> transformation. The plane

could also easily be moved along a path through the scene by using successive pairs of points along its intended flight path as "from" and "at" points used in the new *DoLookAtFrom* <*DOLAF*> objects created for each frame of the sequence.

The *DoLookAtFrom* will not stretch or shrink the cylinder to fit exactly; *DoLookAtFrom* positions and orients primitive objects, but it does not scale them.

Another use for *DoLookAtFrom* <*DOLAF*> would be to position cylindrical objects between arbitrary points in space in order to model the atomic bonds between spherical atoms of a molecule. The "at" and "from" parameters could be the *x*, *y*, *z* centers of the two atoms to be connected by the bond.

Origin of the Coordinate System

The placement of the *origin* of the coordinate system for a particular object is generally important in determining how easily you can rotate the object and position it in relation to other parts. For example, a bolt with the origin at its center could easily be positioned to the center top of a hole (using *DoTranslate* <*DOXLT*>), then rotated to the plane of the hole (using *DoRotate* <*DOROT*>). See Figure 7-14, below.

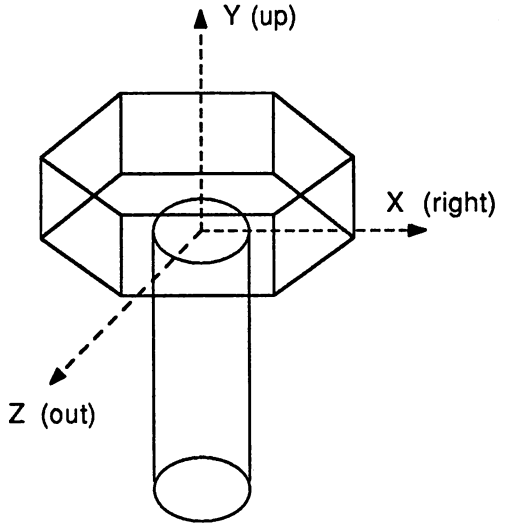


Figure 7-14. Bolt with Origin at the Center

Absolute vs. Relative Group Definition

The bolt shown in Figure 7-15 is defined by a cylinder for the threaded portion, polygons for the top and underside of the head, and six polygons (rectangles) for the faces of the head. The group could be created in two different ways, as shown below.

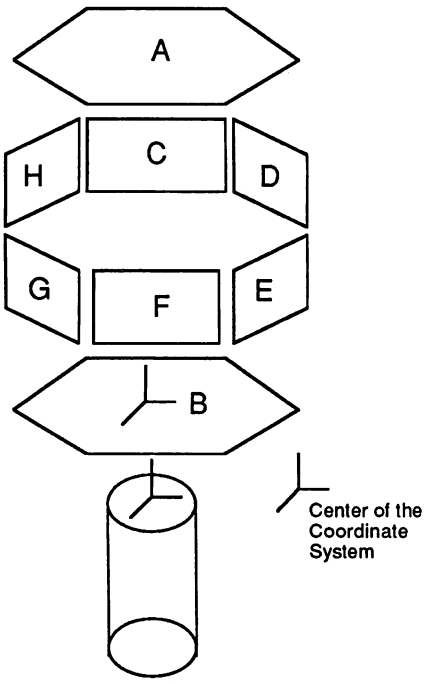


Figure 7-15. Polygons Forming the Bolt

Absolute Group Definition

The absolute definition of the bolt uses absolute coordinates for each of the polygons making up the bolt. Using this method, the group would appear as shown in Figure 7-16, below.

The food processor shown in Plate 7 was created with patches that used absolute coordinates. All components of the food processor were created in the same modeling space, just as the components of the previous bolt were.

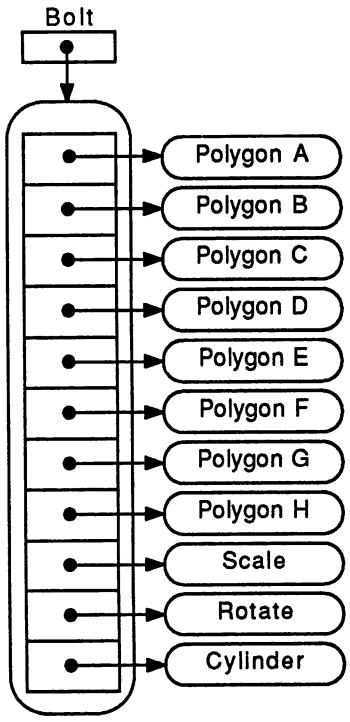


Figure 7-16. Absolute Definition of the Bolt Group

Relative Group Definition

The relative definition of the bolt is longer than the absolute definition and less efficient to render, but it requires only the relative position of each polygon and is thus easier to generate. See Figure 7-17, below.

Polygon A is used twice, first in its original position and then again in a translated position. A push and pop matrix pair is used to create a local geometric context for the top and bottom polygons of the bolt head.

The next push and pop matrix pair is used to isolate the effects of the rotations that position the side polygons so that these rotations do not affect the following definition of the bolt shaft. Because all the sides of the bolt head are simply rotations of each other, only one face is modeled (polygon C). This face is then instanced five more times in different positions. Notice also how successive rotate objects are inherited by the polygon objects later in the group. Finally, the original matrix is restored, and the bolt shaft cylinder is created.

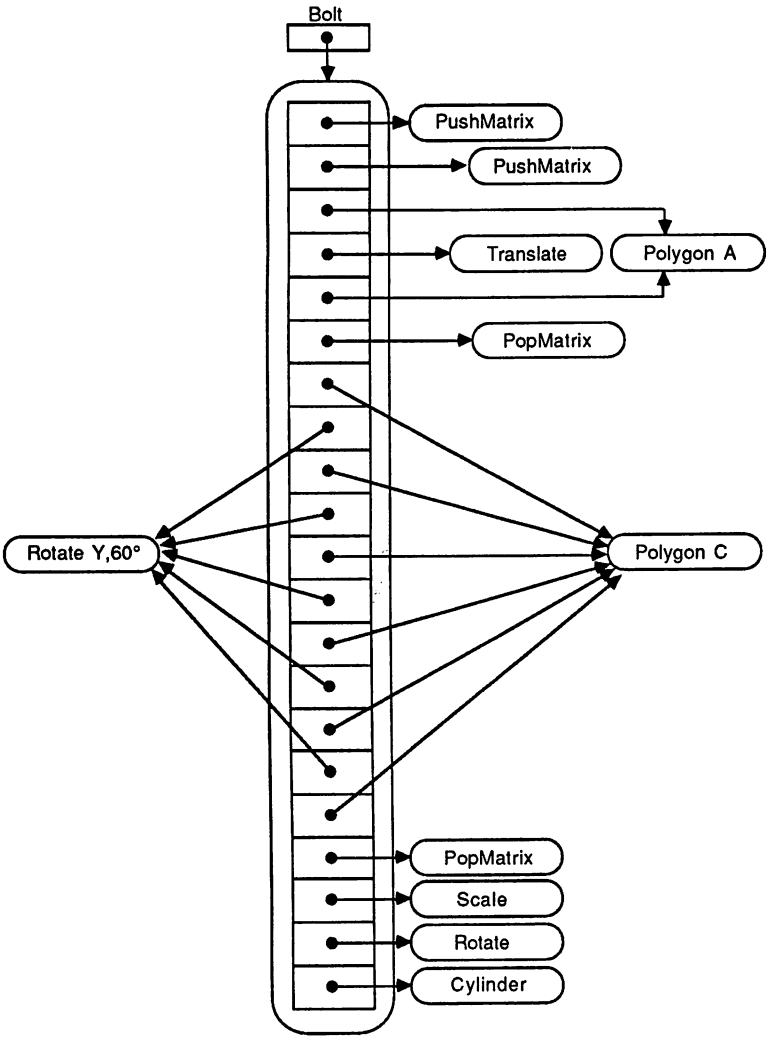


Figure 7-17. Relative Definition of the Bolt Group

Pros and Cons

For simple parts such as the bolt, which have multiple references to them, the absolute group definition is recommended because it executes faster than the relative definition. The relative group requires matrix multiplications for each translate, rotate, and scale, which are performed every time a bolt is used in the database. The absolute definition, on the other hand, contains only one scale and one rotate, which are required to size and position the cylinder. (As shown in the car wheel example in Chapter 4, primitive surfaces are defined with a fixed size and must then

be scaled, translated, and rotated into position.) Because the origin of the bolt and the origin of the cylinder are the same, a translate was not necessary.

The relative definition is easier to specify and to modify than the absolute definition. In addition, when a single object is referenced multiple times, any changes made in that object are reflected in all places it is used. If the object is large, it is also more space efficient to define it once and reference the object a number of times.

Chapter Summary

Chapter 7 describes the geometric transformation attributes that can be used to affect primitive objects, cameras, and lights. A geometric transformation attribute object affects the shape and positioning of primitive or studio objects in three-dimensional space, by *scaling, translating, rotating* or *shearing* the object.

Geometric transformation attribute objects all modify the *current transformation matrix* (CTM). This four-by-four matrix is a compact way of representing transformations and allows arbitrary scaling and positioning of objects in 3D space. The CTM is pushed and popped at group boundaries. Witinh a group, geometric transformations *modify* the CTM; they do not replace its current value. Geometric transformations operate in a manner similar to that of primitive attributes: the last transformation executed is the first one to affect subsequent primitive or studio objects.

It is often useful to control the pushing and popping of the CTM *within* a group, to localize the effects of certain geometric transformations. The *DoPushMatrix* <*DOPUMX*> and *DoPopMatrix* <*DOPPMX*> functions provide you with this control.

The coordinates used to define individual objects are termed *modeling coordinates.* (Another name for these coordinates is *local coordinates.*) Doré uses a *relative coordinate system,* so that the orientation and origin of the coordinate system are affected each time a geometric transformation takes place. The cumulative effect of geometric transformations enables you to build hierarchies of modeling spaces. When all the elements of a scene have been positioned in relation to each other, the scene is said to be defined in terms of *world coordinates.*

All coordinate systems in Doré are right-handed. For example, numbers to the right, up, and forward (out of the screen) would

be *positive*. A positive rotation about a particular axis is found by pointing your right thumb in the positive direction of the axis and curling your fingers around the axis. The direction of the curl of your fingers is a positive rotation.

This chapter also presents examples of absolute and relative group definition and discusses the pros and cons of each approach.

TEXT

CHAPTER EIGHT

This chapter describes the text primitives (*DoText <DOTXT>* and *DoAnnoText <DOANNT>*) and the text-specific attributes. The basic principles applying to groups, objects, and attribute stacking described in Chapters 4 through 7 also apply to text and text attributes. This chapter offers a few examples of text attributes, but you will probably also want to experiment with the wide variety of fonts and possibilities for sizing and orienting text that Doré offers.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DoAnnoText <DOANNT>
DoText <DOTXT>
DoTextAlign <DOTA>
DoTextExpFactor <DOTEF>
DoTextFont <DOTF>
DoTextHeight <DOTH>
DoTextPath <DOTPA>
DoTextPrecision <DOTPR>
DoTextSpace <DOTSP>
DoTextUpVector <DOTUV>

Text Primitives

Two primitive text objects included in Doré are

DoText <DOTXT>
specifies a text string to be drawn at an arbitrary position in an arbitrary plane in 3D space.

DoAnnoText <DOANNT>
specifies a text string to be drawn in a plane parallel to the *xy* plane in display space. The position of the string in this plane is determined by the final transformation of a given point.

Text Primitive Attributes

The images produced by the following three code fragments are shown in Figure 8-1, Figure 8-2, and Figure 8-3.

C code:

```
DtObject example[6];
static DtVector3 u = {1.0, 0.0, 0.0},
                 v = {0.0, 1.0, 0.0};
static DtPoint3 origin = {0.0, 0.0, 0.0};
static char name[] = "Ardent";

example [1] = DoGroup(DcTrue);
     DgAddObj(DoTextHeight(1.0));
     DgAddObj(DoTextUpVector(0.0, 1.0));
     DgAddObj(DoTextPath(DcTextPathRight));
     DgAddObj(DoTextAlign(DcTextHAlignCenter, DcTextVAlignHalf));
     DgAddObj(DoText(origin, u, v, name));
DgClose();

example [2] = DoGroup(DcTrue);
     DgAddObj(DoTextHeight(0.5));
     DgAddObj(DoTextUpVector(-1.0, 0.0));
     DgAddObj(DoTextPath(DcTextPathRight));
     DgAddObj(DoTextAlign(DcTextHAlignCenter, DcTextVAlignHalf));
     DgAddObj(DoText(origin, u, v, name));
DgClose();

example [3] = DoGroup(DcTrue);
     DgAddObj(DoTextHeight(0.5));
     DgAddObj(DoTextUpVector(0.0, 1.0));
     DgAddObj(DoTextPath(DcTextPathDown));
     DgAddObj(DoTextAlign(DcTextHAlignCenter, DcTextVAlignHalf));
     DgAddObj(DoText(origin, u, v, name));
DgClose();
```

Fortran code:

```
      INTEGER*4 EXAMPLE(6)
           REAL*8 U(3)
           REAL*8 V(3)
           REAL*8 ORIGIN (3)
           CHARACTER*6 NAME
           DATA NAME /'ARDENT'/
C
           U(1)=1.0D0
           U(2)=0.0D0
```

```
      U(3)=0.0D0
      V(1)=0.0D0
      V(2)=1.0D0
      V(3)=0.0D0
      ORIGIN(1)=0.0D0
      ORIGIN(2)=0.0D0
      ORIGIN(3)=0.0D0
C
      EXAMPLE(1)=DOG(DCTRUE)
         CALL DGAO(DOTH(1.0D0))
         CALL DGAO(DOTUV(0.0D0, 1.0D0))
         CALL DGAO(DOTPA(DCTPR))
         CALL DGAO(DOTA(DCTHAC, DCTVAH))
         CALL DGAO(DOTXT(ORIGIN, U, V, NAME))
      CALL DGCS()
C
      EXAMPLE(2)=DOG(DCTRUE)
         CALL DGAO(DOTH(0.5D0))
         CALL DGAO(DOTUV(-1.0D0, 0.0D0))
         CALL DGAO(DOTPA(DCTPR))
         CALL DGAO(DOTA(DCTHAC, DCTVAH))
         CALL DGAO(DOTXT(ORIGIN, U, V, NAME))
      CALL DGCS()
C
      EXAMPLE(3)=DOG(DCTRUE)
         CALL DGAO(DOTH(0.5D0))
         CALL DGAO(DOTUV(0.0D0, 1.0D0))
         CALL DGAO(DOTPA(DCTPD))
         CALL DGAO(DOTA(DCTHAC, DCTVAH))
         CALL DGAO(DOTXT(ORIGIN, U, V, NAME))
      CALL DGCS()
```

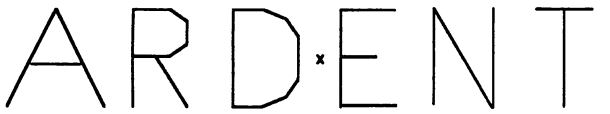


Figure 8-1. Example 1

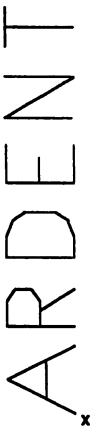


Figure 8-2. Example 2

Figure 8-3. Example 3

Example 1 uses defaults for text font, height, up vector, and path. The text alignment (*DoTextAlign* <*DOTA*>) is *DcTextHAlignCenter* <*DCTHAC*>, which means the text position specified in *DoText*

<*DOTXT*> is in the horizontal center of an imaginary box enclosing the text string, and *DcTextVAlignHalf* <*DCTVAH*>, which means that the text position is halfway between the top and bottom of an imaginary box enclosing the text string. (The text position is marked by an x in Figures 1 through 3.) Examples 1 through 3 use the default font, *DcPlainRoman* <*DCFPR*>, which has only capital letters.

Example 2, shown in Figure 8-2, changes the text height to 0.5, which makes the text half as large as the text in Example 1. The text up vector is specified as (-1.0, 0.0), which makes the "backbone" of the text point in the negative *x* direction.

Example 3, shown in Figure 8-3, uses the default up vector but changes the text path to *DcTextPathDown* <*DCTPD*>.

The images produced by the next three code fragments are shown in Figure 8-4, Figure 8-5, and Figure 8-6.

C code:

```
DtObject example[6];
static DtVector3 u = {1.0, 0.0, 0.0},
                 v = {0.0, 1.0, 0.0};
static DtPoint3 origin = {0.0, 0.0, 0.0};
static char name[] = "Ardent";

example [4] = DoGroup(DcTrue);
     DgAddObj(DoTextFont(DcGothicItalian));
     DgAddObj(DoTextExpFactor(1.0));
     DgAddObj(DoTextSpace(0.0));
     DgAddObj(DoTextAlign(DcTextHAlignCenter, DcTextVAlignHalf));
     DgAddObj(DoText(origin, u, v, name));
DgClose();

example [5] = DoGroup(DcTrue);
     DgAddObj(DoTextFont(DcGothicItalian));
     DgAddObj(DoTextExpFactor(0.5));
     DgAddObj(DoTextSpace(0.0));
     DgAddObj(DoTextAlign(DcTextHAlignCenter, DcTextVAlignHalf));
     DgAddObj(DoText(origin, u, v, name));
DgClose();

example [6] = DoGroup(DcTrue);
     DgAddObj(DoTextFont(DcGothicItalian));
     DgAddObj(DoTextExpFactor(1.0));
     DgAddObj(DoTextSpace(0.3));
     DgAddObj(DoTextAlign(DcTextHAlignCenter, DcTextVAlignHalf));
     DgAddObj(DoText(origin, u, v, name));
DgClose();
```

Fortran code:

```
   INTEGER*4 EXAMPLE(6)
      REAL*8 U(3), V(3), ORIGIN(3)
      CHARACTER*6 NAME
      DATA NAME /'ARDENT'/
C
      U(1)=1.0D0
      U(2)=0.0D0
      U(3)=0.0D0
      V(1)=0.0D0
      V(2)=1.0D0
      V(3)=0.0D0
      ORIGIN(1)=0.0D0
      ORIGIN(2)=0.0D0
      ORIGIN(3)=0.0D0
C
      EXAMPLE(4)=DOG(DCTRUE)
               CALL DGAO(DOTF(DCFGI))
               CALL DGAO(DOTEF(1.0D0))
               CALL DGAO(DOTSP(0.0D0))
               CALL DGAO(DOTA(DCTHAC, DCTVAH))
               CALL DGAO(DOTXT(ORIGIN, U, V, NAME))
      CALL DGCS()
C
      EXAMPLE(5)=DOG(DCTRUE)
               CALL DGAO(DOTF(DCFGI))
               CALL DGAO(DOTEF(0.5D0))
               CALL DGAO(DOTSP(0.0D0))
               CALL DGAO(DOTA(DCTHAL, DCTVAH))
               CALL DGAO(DOTXT(ORIGIN, U, V, NAME))
      CALL DGCS()
C
      EXAMPLE(6)=DOG(DCTRUE)
               CALL DGAO(DOTF(DCFGI))
               CALL DGAO(DOTEF(1.0D0))
               CALL DGAO(DOTSP(0.3D0))
               CALL DGAO(DOTA(DCTHAC, DCTVAH))
               CALL DGAO(DOTXT(ORIGIN, U, V, NAME))
      CALL DGCS()
```

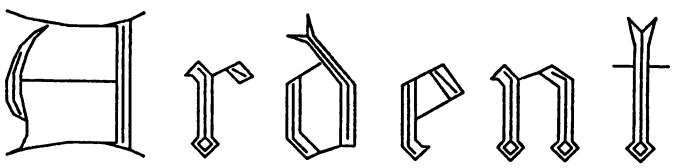


Figure 8-4. Example 4

Figure 8-5. Example 5

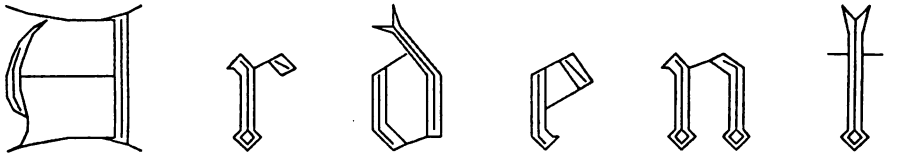


Figure 8-6. Example 6

Examples 4 through 6 use the Gothic Italian font, which has both capital and lowercase letters. Example 4 shows the default text expansion factor (1.0) and default text space (0.0).

Example 5 changes the text expansion factor to 0.5, which contracts the character width 50 percent. Example 6 changes the text spacing to 0.3, which increases the amount of space between adjacent characters by 0.3.

Chapter Summary

Chapter 8 describes the text primitives and the primitive attributes that apply specifically to text. Since text is a subset of primitive objects discussed in Chapter 5, and text attributes are a subset of primitive attribute objects discussed in Chapter 6, few new concepts are introduced in this chapter. Text attributes illustrated in this chapter include text height, font, expansion factor, alignment, path, up vector, and spacing.

CAMERAS AND LIGHTS

CHAPTER NINE

This chapter describes a new group of objects, *studio objects*, which comprises *cameras* and *lights*. *Studio attribute objects*, which modify studio objects, are also discussed. Cameras can use *parallel* or *perspective* projection or other more complex projections if you require them. Lights can be of a particular type, color, and intensity. The chapter example shows the use of six different camera groups to view the same object, a house. Concepts and terms introduced in this chapter include the camera projection matrix, camera coordinates, parallel and perspective projection, hither and yon clipping planes, global rendering attributes, and stereo viewing.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DoCamera <DOCM>
DoCameraMatrix <DOCMX>
DoGlbRendMaxObjs <DOGRMO>
DoGlbRendMaxSub <DOGRMS>
DoGlbRendRayLevel <DOGRRL>
DoLight <DOLT>
DoLightAttenuation <DOLTA>
DoLightColor <DOLC>
DoLightIntens <DOLI>
DoLightSpreadAngles <DOLTSA>
DoLightSpreadExp <DOLSE>
DoLightType <DOLTT>
DoParallel <DOPAR>
DoPerspective <DOPER>
DoProjection <DOPRJ>
DoStereo <DOSTER>
DoStereoSwitch <DOSTES>

Studio Objects

Chapters 5 through 8 focused on primitive objects and the primitive attribute objects that modify them. You saw how primitive objects are first defined in their own modeling space, and then are scaled, rotated, and translated into position relative to each other in *world space*. World space uses whatever *world coordinates* are convenient to define all objects that currently exist in the graphics "world" you are depicting.

Next, you must choose a position and an angle from which to view this "world." Because this procedure is analogous to taking a picture of a scene in real life, Doré allows you to create a *camera* for a particular scene. You position the camera in space, specify where to point it, indicate whether this "camera" is a *perspective* camera that forms images as our eyes do, or whether it forms a different kind of image, and set the aperture. You can have any number of cameras in your scene, but you can take only one picture at a time, as described in more detail later in this chapter.

Just as a photographer in a studio can control the position, type, and color of the lights on the subject matter, you can create different kinds of lights and position them relative to the scene you are about to photograph with your camera.

Studio Objects and Their Attribute Objects

Studio attributes are bound to their studio objects (cameras and lights) in the same manner as primitive objects are bound to their attribute objects. For example, camera attribute objects include *DoParallel* <*DOPAR*>, *DoPerspective* <*DOPER*>, and *DoStereo* <*DOSTER*>. Attribute objects that affect lights include *DoLightColor* <*DOLC*>, *DoLightIntens* <*DOLI*>, and *DoLightType* <*DOLTT*>.

Geometric Transformations and Studio Objects

Geometric transformation attribute objects are bound to studio objects just as they are bound to primitive objects. The *DoLookAtFrom* <*DOLAF*> function is commonly used to position lights and cameras in a scene.

Cameras and Their Attributes

A camera is created with *DoCamera <DOCM>* and positioned in relation to the scene. One of the following camera attribute objects is then used to describe the type of camera projection:

DoParallel <DOPAR>
forms a parallel projection. With this projection, parallel lines in the display objects remain parallel in the image.

DoPerspective <DOPER>
forms a perspective projection. With this projection, foreshortening occurs and distant objects appear smaller than near objects.

DoProjection <DOPRJ>
forms orthographic or oblique projections. Use *DoProjection <DOPRJ>* when you are unable to obtain the desired effects with *DoParallel <DOPAR>* or *DoPerspective <DOPER>*.

DoCameraMatrix <DOCMX>
specifies an arbitrary four-by-four matrix for the camera projection matrix attribute. Since *DoCameraMatrix <DOCMX>* is the most complex way to specify camera projection, it should be used only when you cannot obtain the desired effect using any of the other three camera projection functions.

Clipping at View Boundaries

Primitive objects are clipped to the view boundaries when the camera projection attribute is applied (*DoParallel <DOPAR>*, *DoPerspective <DOPER>*, *DoProjection <DOPRJ>*, or *DoCameraMatrix <DOCMX>*). *Clipping* refers to the process of checking to see if any part of a primitive object falls outside of the viewing area; whatever part lies outside is cut.

For all camera types, primitive objects are clipped in the *z* direction at the *hither* and *yon* clipping planes. For parallel cameras, objects are also clipped at the four planes that define the *xy* window. For perspective cameras, objects are clipped at the four planes that define the field of view.

Hither and Yon Clipping Planes

Figure 9-1 and Figure 9-2 show the parameters used for parallel and perspective projections. Objects are clipped in the z direction against the *hither* and *yon* clipping planes for both types of projections.

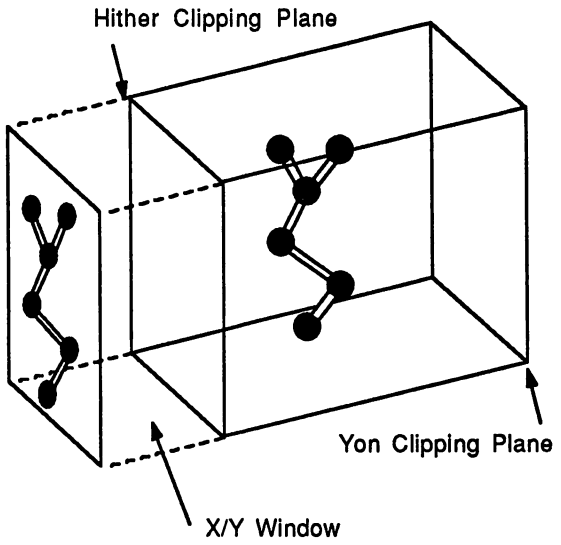


Figure 9-1. Parallel Projection

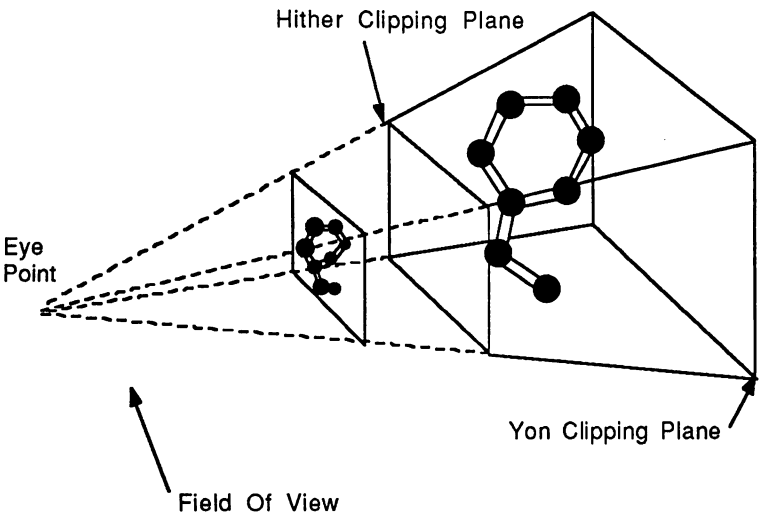


Figure 9-2. Perspective Projection

Camera Projection Matrix

The camera projection matrix attribute specifies the camera projection used for all subsequently executed camera objects. Each camera projection function (*DoParallel* <*DOPAR*>, *DoPerspective* <*DOPER*>, *DoProjection* <*DOPRJ*>, and *DoCameraMatrix* <*DOCMX*>) specifies a *replacement* for the camera projection matrix attribute. **Unlike the geometric transformation attribute objects, these camera attribute objects are not additive.** (This is also probably intuitively obvious to you, since one camera would not be able to function in two different ways at the same time.)

Example

Figure 9-3 shows the same object, a house, added to six different views, each with a different camera. The top three views use perspective cameras, and the bottom three views use parallel cameras. All cameras are positioned at the same point in the scene with *DoLookAtFrom* <*DOLAF*>. In the parallel projections (bottom row), as the window grows (from 15x15 to 40x40 to 65x65), the object becomes smaller. In the perspective projections, as the field of view angle widens, the object takes up less of the field of view. With a very narrow field of view (25 degrees), you can see only part of the house. A large field of view (100 degrees), like a wide angle lens, causes the house to appear far away because it takes up such a small portion of the field of view.

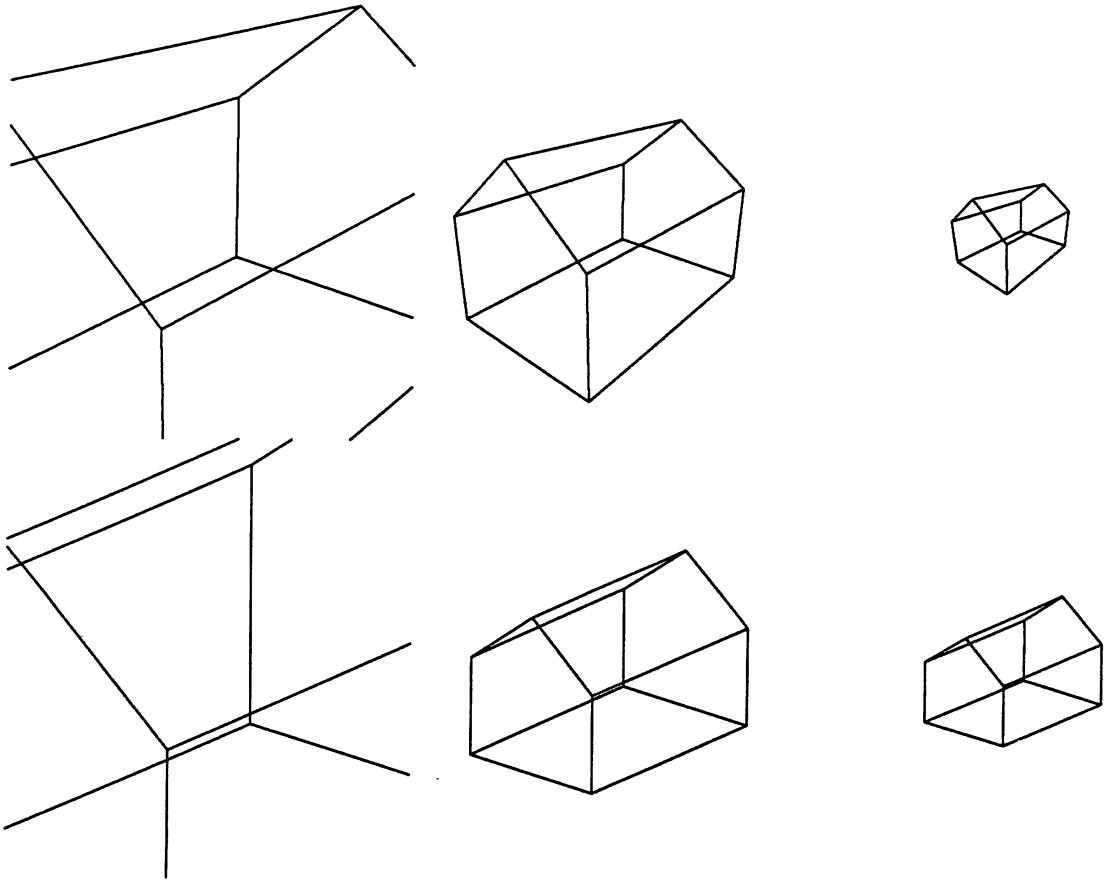


Figure 9-3. Examples of Perspective and Parallel Camera Projections

Fragments of the code for the cameras and lights for this figure are shown below.

C code:

```
static DtPoint3 origin = {0.0, 0.0, 0.0};
static DtPoint3 camera_from = {25.0, 15.0, -30.0};
static DtPoint3 light_from = {0.0, 0.0, -10.0};
static DtVector3 up = {0.0, 1.0, 0.0};
DtObject camera1, camera2, camera3, camera4, camera5, camera6,
      light;

camera1 = DoGroup(DcTrue);     /* creates the bottom left camera group */
     DgAddObj(DoParallel(15.0, -0.01, -100.0));    /* smallest window */
```

```
     DgAddObj(DoLookAtFrom(origin, camera_from, up));
     DgAddObj(DoCamera());
DgClose();

camera2 = DoGroup(DcTrue);     /* creates the bottom middle camera group */
     DgAddObj(DoParallel(40.0, -0.01, -100.0));    /* larger window */
     DgAddObj(DoLookAtFrom(origin, camera_from, up));
     DgAddObj(DoCamera());
DgClose();

camera3 = DoGroup(DcTrue);     /* creates the bottom right camera group */
     DgAddObj(DoParallel(65.0, -0.01, -100.0));    /* largest window */
     DgAddObj(DoLookAtFrom(origin, camera_from, up));
     DgAddObj(DoCamera());
DgClose();

camera4 = DoGroup(DcTrue);     /* creates the top left camera group */
     DgAddObj(DoPerspective(25.0, -0.01, -100.0));
     DgAddObj(DoLookAtFrom(origin, camera_from, up));
     DgAddObj(DoCamera());
DgClose();

camera5 = DoGroup(DcTrue);     /* creates the top middle camera group */
     DgAddObj(DoPerspective(50.0, -0.01, -100.0));
     DgAddObj(DoLookAtFrom(origin, camera_from, up));
     DgAddObj(DoCamera());
DgClose();

camera6 = DoGroup(DcTrue);     /* creates the top right camera group */
     DgAddObj(DoPerspective(100.0, -0.01, -100.0));
     DgAddObj(DoLookAtFrom(origin, camera_from, up));
     DgAddObj(DoCamera());
DgClose();

light = DoGroup(DcTrue);
     DgAddObj(DoLightIntens(1.0));
     DgAddObj(DoLookAtFrom(origin, light_from, up));
     DgAddObj(DoLight());
DgClose();
```

Fortran code:

```
     REAL*8 ORIGIN(3), CAMERA_FROM(3)
     REAL*8 LIGHT_FROM(3), UP(3)
     INTEGER*4 CAMERA1, CAMERA2, CAMERA3, CAMERA4, CAMERA5,
    1CAMERA6, LIGHT
C
     ORIGIN(1)=0.0D0
     ORIGIN(2)=0.0D0
     ORIGIN(3)=0.0D0
     CAMERA_FROM(1)=25.0D0
     CAMERA_FROM(2)=15.0D0
     CAMERA_FROM(3)=-30.0D0
     LIGHT_FROM(1)=0.0D0
```

Example
(continued)

```
      LIGHT_FROM(2)=0.0D0
      LIGHT_FROM(3)=-10.0D0
      UP(1)=0.0D0
      UP(2)=1.0D0
      UP(3)=0.0D0
C
      CAMERA1=DOG(DCTRUE) !CREATES THE BOTTOM LEFT CAMERA GROUP!
           CALL DGAO(DOPAR(15.0D0, -0.01D0, -100.0D0))
           CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
           CALL DGAO(DOCM())
      CALL DGCS()
C
      CAMERA2=DOG(DCTRUE) !CREATES BOTTOM MIDDLE CAMERA GROUP!
           CALL DGAO(DOPAR(40.0D0, -0.01D0, -100.0D0))
           CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
           CALL DGAO(DOCM())
      CALL DGCS()
C
      CAMERA3=DOG(DCTRUE) !BOTTOM RIGHT CAMERA GROUP!
           CALL DGAO(DOPAR(65.0D0, -0.01D0, -100.0D0))
           CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
           CALL DGAO(DOCM())
      CALL DGCS()
C
      CAMERA4=DOG(DCTRUE) !TOP LEFT CAMERA GROUP!
           CALL DGAO(DOPER(25.0D0, -0.01D0, -100.0D0))
           CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
           CALL DGAO(DOCM())
      CALL DGCS()
C
      CAMERA5=DOG(DCTRUE) !TOP MIDDLE CAMERA GROUP!
           CALL DGAO(DOPER(50.0D0, -0.01D0, -100.0D0))
           CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
           CALL DGAO(DOCM())
      CALL DGCS()
C
      CAMERA6=DOG(DCTRUE) !TOP RIGHT CAMERA GROUP!
           CALL DGAO(DOPER(100.0D0, -0.01D0, -100.0D0))
           CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
           CALL DGAO(DOCM())
      CALL DGCS()
C
      LIGHT=DOG(DCTRUE)
           CALL DGAO(DOLI(1.0D0))
           CALL DGAO(DOLAF(ORIGIN, LIGHT_FROM, UP))
           CALL DGAO(DOLT())
      CALL DGCS()
```

See also the example in Chapter 10, "Views, Frames, and Devices," which shows four different cameras added to four different views of the same object.

DoPushMatrix and DoPopMatrix

The example above uses groups to push and pop attributes for each camera. If it is more convenient to put all cameras for a particular view into one group, you can use *DoPushMatrix* <*DOPUMX*> and *DoPopMatrix* <*DOPPMX*> to explicitly push and pop the attributes for each camera and light. For example:

C code:

```
DtObject camera, camera1, camera2, camera3;

camera = DoGroup(DcTrue);
     DgAddObj(DoPerspective(100.0, -0.01, -100.0));
     DgAddObj(DoPushMatrix());
          DgAddObj(DoLookAtFrom(origin, camera_from, up));
          DgAddObj(camera1=DoCamera());
     DgAddObj(DoPopMatrix());
     DgAddObj(DoPushMatrix());
          /* then add camera2 and its characteristics */
     DgAddObj(DoPopMatrix());
     DgAddObj(DoPushMatrix());
          /* then add camera3 and its characteristics */
     DgAddObj(DoPopMatrix());
DgClose();
```

Fortran code:

```
    INTEGER*4 CAMERA1, CAMERA2, CAMERA3
C
     CAMERA=DOG(DCTRUE)
          CALL DGAO(DOPER(100.0D0, -0.01D0, -100.0D0))
          CALL DGAO(DOPUMX())
               CALL DGAO(DOLAF(ORIGIN, CAMERA_FROM, UP))
               CALL DGAO(CAMERA1=DOCAMERA())
          CALL DGAO(DOPPMX())
          CALL DGAO(DOPUMX())
C    THEN ADD CAMERA2 AND ITS CHARACTERISTICS
          CALL DGAO(DOPPMX())
          CALL DGAO(DOPUMX())
C    THEN ADD CAMERA3 AND ITS CHARACTERISTICS
          CALL DGAO(DOPPMX())
     CALL DGCS()
```

The Active Camera

As mentioned earlier, many cameras can be defined for a particular view, but only one camera is designated as the *active* camera for a particular view at a given time. Use *DvSetActiveCamera* <*DVSAC*> to specify the active camera for rendering a view. *DvInqActiveCamera* <*DVQAC*> returns the active camera for a

view. In the previous example, we might have used *DvSetActiveCamera (view, camera1).*

Coordinate Systems

The following list outlines the basic transformations you have encountered to this point.

(1) During rendering initialization, the cameras and lights in definition groups transform *world coordinates* to *camera coordinates.*

(2) During rendering, objects in display groups, which are first described in their own *local coordinates,* are then positioned relative to each other in *world coordinates.*

(3) Eventually, when the scene is projected onto the camera lens, the world coordinates are transformed to *camera projection coordinates.*

(4) This entire projected scene is then placed in a view, which is defined by *view coordinates.* The view coordinate system is identical to the *frame coordinate system.* Default frame coordinates are (0.0, 0.0, 0.0) for the back lower left corner to (1.0, 1.0, 1.0) for the front upper right corner.

Chapter 10, "Views, Frames, and Devices," will trace this path to its final destination, the device, and explain the relationship between views and frames in more detail.

DoStereo and DoStereoSwitch

A stereo device is required to make use of *DoStereo* and *DoStereoSwitch.*

DoStereo <*DOSTER*> creates a camera with two viewpoints, one for each eye, which gives a more realistic 3D effect to a scene. First, you specify the *eye separation* distance, which is the distance from the normal camera position to each of the two stereo camera positions. The total distance between the left and right camera viewpoints is thus 2 times the camera separation distance specified. Second, you specify the *focal* distance, which is the distance from the original camera position to the point along the camera direction vector at which both stereo camera directions are centered. This controls the "focus" point, which becomes important when objects are close. The ratio between the eye separation distance and the focal distance is generally about 1 to 10.

DoStereoSwitch <*DOSTES*> specifies whether a camera is a stereo camera. If on, the camera is translated to two different viewpoints, one for the left eye and one for the right eye.

Ray Tracing and Cameras

When standard ray tracing is used to render a scene, three global rendering attributes that are bound to the camera are used:

DoGlbRendMaxSub <DOGRMS>
specifies the maximum level of spatial subdivisions for ray tracing. The default is 10.

DoGlbRendMaxObjs <DOGRMO>
specifies the maximum number of objects per spatial subdivision for ray tracing. The default is 1.

DoGlbRendRayLevel <DOGRRL>
specifies the maximum number of ray bounces. The default is 3.

Doré's standard ray tracer is used when *DvSetRendStyle* <*DVSRS*> is set to *DcProductionTime* <*DCPRTM*>. See Chapter 12, "Rendering," and the *Doré Reference Manual.*

Lights and Their Attributes

Light objects, like camera objects, are part of a scene but are not visible themselves. In contrast to cameras, more than one light can be used in a scene at one time. A light created with *DoLight* <*DOLT*> is, by default, a white light source at infinity, oriented along the positive z axis, facing the negative z direction, with an intensity of 1.0.

Use the various *light attribute* object functions to create lights with other properties. Important light attributes are

Type
DoLightType <*DOLTT*> specifies the primary type of subsequent light objects: ambient light, light source at infinity, spot light, attenuated spot light, point light, and attenuated point light. (See the sections below for more description of these light types.) Not all renderers can make use of all the available light types.

Intensity

DoLightIntens <DOLI> specifies the overall intensity of subsequent light objects. Intensities are normally between 0.0 and 1.0.

Color

DoLightColor <DOLC> specifies the color of subsequent light objects.

Ambient Light

An ambient light source distributes light equally throughout the scene. Since ambient light is nondirectional, its exact position in the scene is not important. Ambient light produces the inherent "glow" of an object and does not cast shadows. The ambient light component is the sum of all the ambient lights in the scene. The illumination from each ambient light is determined by multiplying the light color times the light intensity:

```
DoLightColor * DoLightIntens
```

See Figure 9-4.

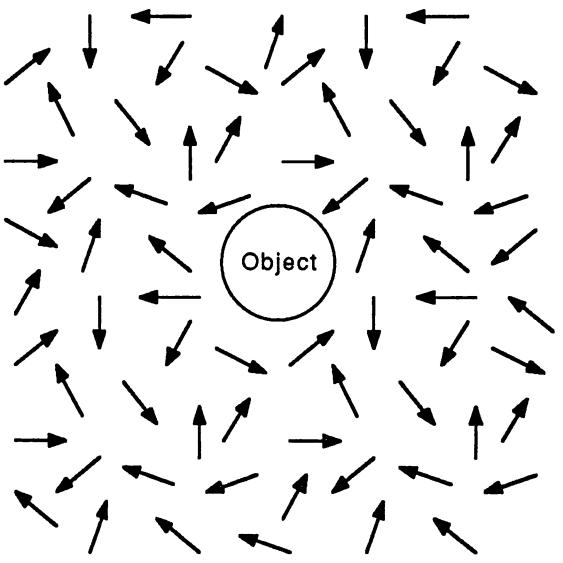


Figure 9-4. Ambient Light Source

Light Source at Infinity

Like sunlight, light rays produced by a light source at infinity can be represented by a series of parallel vectors, as shown in Figure 9-5. A light source at infinity causes primitive objects in its path to cast shadows on their nonilluminated sides. The distance of an object from a light source at infinity is not important, but the direction is. For lights at infinity, as for ambient lights, the light color is multiplied by the light intensity to determine the illumination from each light:

```
DoLightColor * DoLightIntens
```

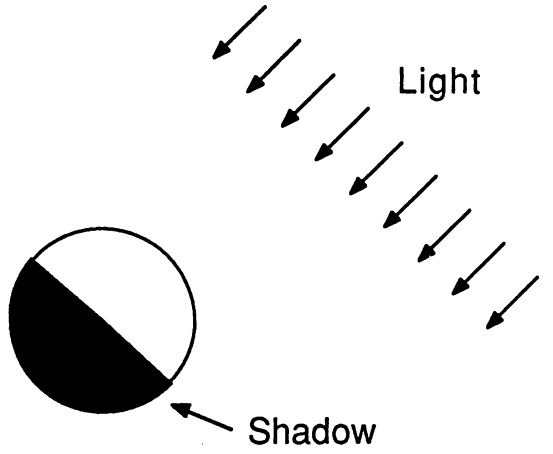


Figure 9-5. Light Source at Infinity

Point Lights

A point light source radiates light outward uniformly in all directions (see Figure 9-6). This light source is analogous to a light bulb or a match, although it is not actually visible.

Point light sources can be either standard point lights, specified by *DcLightPoint* <*DCLTPT*> or attenuated point lights, specified by *DcLightPointAttn* <*DCLTPA*>. Attenuated point lights have an additional attribute that describes how the light intensity falls off as the distance from the light increases (see the following section).

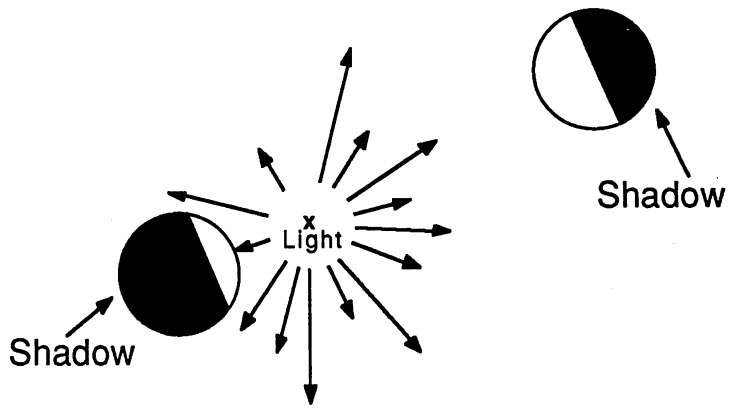


Figure 9-6. Point Light Source

The positions of point light sources are important. As this light source gets closer to the object, a larger proportion of the object will be in shadow. For standard point lights, the illumination from each light is again the product of the light color and the light intensity:

```
DoLightColor * DoLightIntens
```

Attenuated Point Lights

Attenuated point lights (and also attenuated spot lights, described below), are affected by the studio attribute *DoLightAttenuation* <*DOLTA*>, which specifies the falloff of light as the distance from the light source increases. This falloff is described by the following equation (*c1* and *c2* are parameters to *DoLightAttenuation*):

```
lightattenuation = 1./(c1 + c2 * distance_from_lgt)
```

In this equation, *1/c1* defines the initial value of light attenuation. *c2* defines how fast the intensity falls off. The larger *c2* is, the faster the light intensity drops off from the initial value to 0.

The illumination from an attenuated point light at a particular point in space can be represented as

```
DoLightColor * DoLightIntens * lightattenuation
```

Spot Lights

A spot light is a directed light source that radiates light outward from the light position in the light direction and whose light intensity decreases as a function of the angle from the light direction (see Figure 9-7). This drop in light intensity can be anywhere from an abrupt drop, which produces sharp edges between full light and darkness, to a gradual drop in light intensity, which produces soft edges to the cone of light (see next section). Like point lights, spot lights can be either standard (*DcLightSpot <DCLTSP>*) or attenuated (*DcLightSpotAttn <DCLTSA>*).

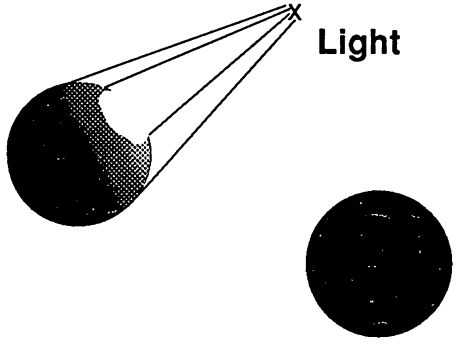


Figure 9-7. Spot Light Source

A *spread angle* and a *spread exponent* can be defined for either type of spot light. In the following equation, the effect of the spread angle is represented by *lightspread1*. The effect of the spread exponent is represented by *lightspread2*. For standard spot lights, the illumination from each light is represented as

```
DoLightColor * DoLightIntens * lightspread1 * lightspread2
```

lightspread1 and *lightspread2* are described in the following sections on "Spread Angle" and "Spread Exponent".

Spread Angle

The *DoLightSpreadAngles <DOLTSA>* function defines the size of the light cone for both standard spot lights and attenuated spot lights. Actually, as shown in Figure 9-8, this function defines two cones of light. Anywhere within the center cone, *lightspread1* is full intensity. This center cone is specified as *(total-delta)*. On the outside of this cone is a secondary cone, specified as *total*.

Between these two cones, *lightspread1* falls off from full intensity (1) to no intensity (0). The larger the value of *delta* is, the softer and more gradual the decline in *lightspread1*.

This effect can be represented as

```
if (alpha < (total-delta)) then lightspread1 = 1.0
else if (alpha > total) then lightspread1 = 0.0
else lightspread1 = 1 - ((alpha-(total-delta))/delta)
```

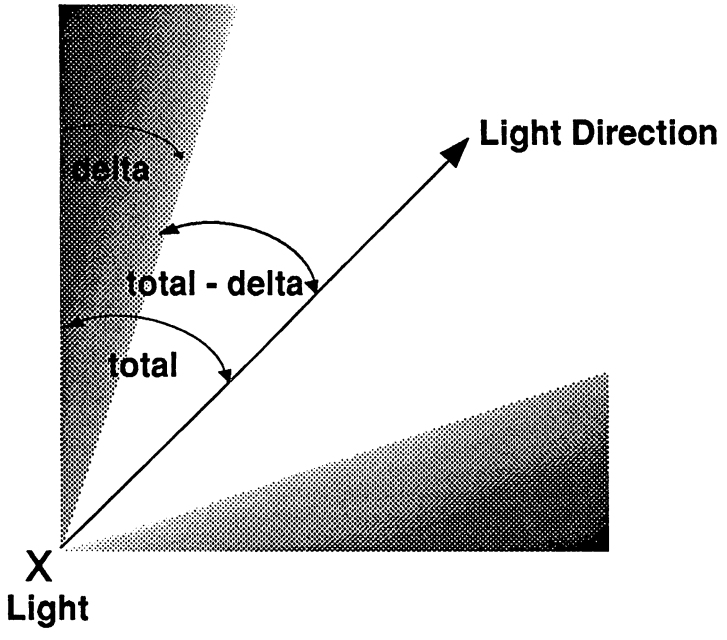


Figure 9-8. Parameters for Light Spread Angles

Spread Exponent

The *DoLightSpreadExp <DOLSE>* function defines another factor affecting light intensity of spot lights. For any point *p*, a line can be drawn from *p* to the light position, as shown in Figure 9-9. The cosine of the angle formed by this line and the light direction vector (angle alpha in Figure 9-9), is raised to the power *exponent* specified in *DoLightSpreadExp <DOLSE>*:

```
lightspread2 = cos (alpha) ** exponent
```

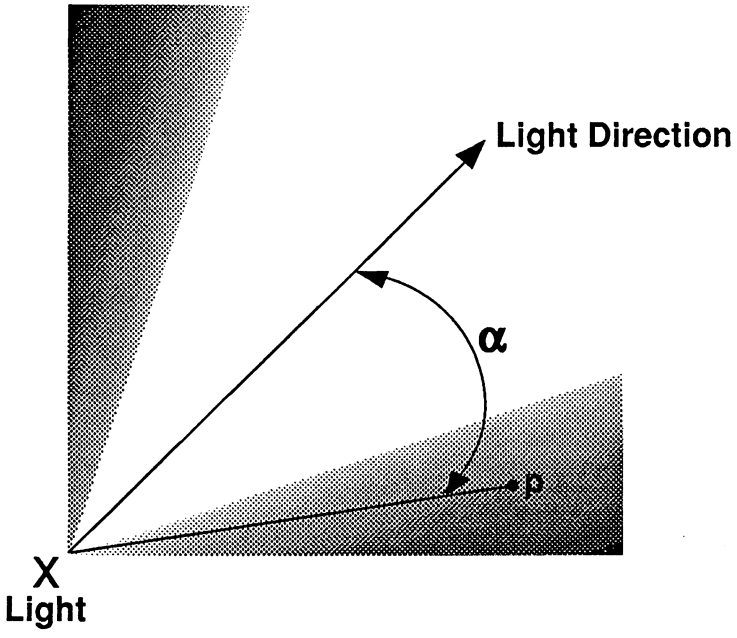


Figure 9-9. Angle Alpha Used with the Light Spread Exponent

Figure 9-10 shows how four different light spread exponents affect the light intensities for angles from -90 degrees to 90 degrees.

Attenuated Spot Lights

Like attenuated point lights, attenuated spot lights are affected by the studio attribute *DoLightAttenuation* <*DOLTA*>. The fall-off of light as the distance from the light source increases is the same as described in the section on attenuated point lights. The illumination from each attenuated spot light can be represented as

```
DoLightColor * DoLightIntens * lightspread1 * lightspread2 * lightattenuation
```

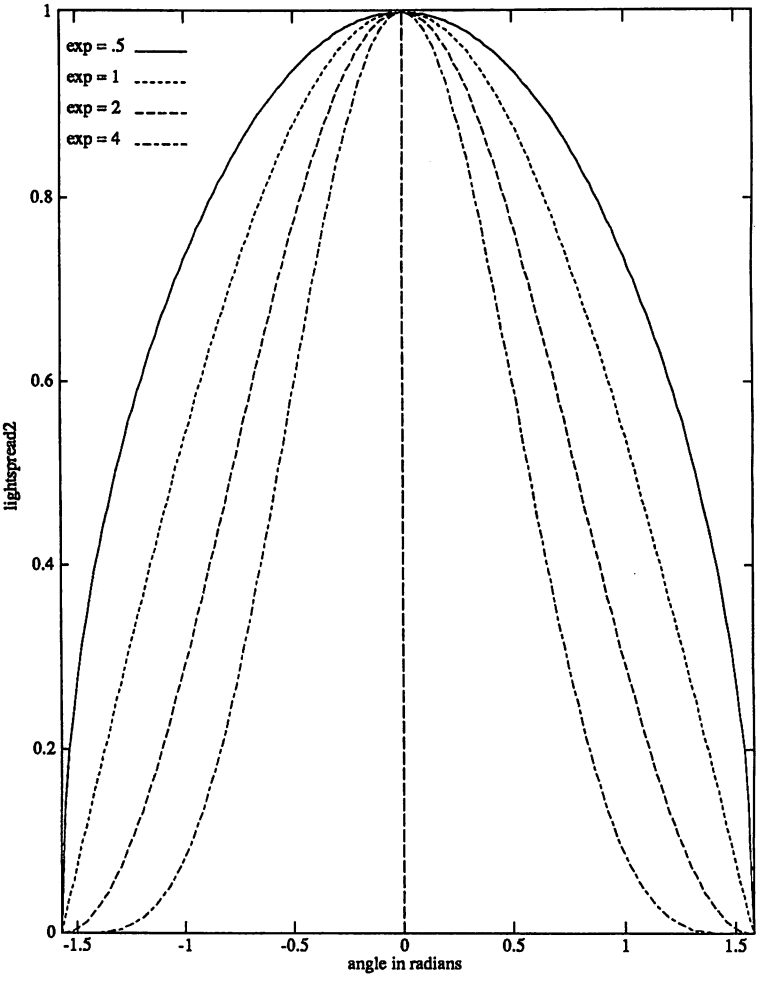


Figure 9-10. Effect of Different Light Spread Exponents

Light Switch

The function *DoLightSwitch* <*DOLTS*> creates a primitive attribute object that specifies whether subsequent primitive objects will be illuminated by a particular light object. If the switch value is *DcOff*, subsequent primitive objects will not be lit by the light source. If it is the only light source, those objects will not be seen. If the switch value for a particular light is *DcOn* (the default), subsequent objects will reflect light from that light source.

Shadows

Shadows can be produced when high-quality rendering is used. The following function must be enabled for shadows to occur:

DoShadowSwitch <DOSHAS>
creates a *primitive attribute* object that specifies whether subsequent primitive objects will be displayed with shadows on their surfaces.

Chapter Summary

Chapter 9 describes cameras and lights, which form a new class of objects called *studio objects*. Studio objects are modified by *studio attribute objects*, which include attribute objects affecting light type, color, and intensity, and the type of camera projection. Certain global rendering attributes used during ray tracing are bound to the camera as well.

Studio attributes are bound to cameras and lights just as primitive attributes are bound to primitive objects. The same principles of attribute stacking apply to both studio and primitive attributes. Geometric transformations are bound to studio objects in the same manner that they are bound to primitive objects.

You can position any number of cameras in a scene, but only one camera is active at a time. Simple camera projections are the *parallel* and *perspective* projections. The *DoProjection <DOPRJ>* function can be used to obtain other types of projections. For the most complex projections, an arbitrary camera projection matrix can be specified with *DoCameraMatrix <DOCMX>*.

The *camera projection matrix* specifies the camera projection used for all subsequently executed camera objects. The camera projection functions specify a *replacement* for the camera projection matrix attribute. (This contrasts to the geometric transformation attribute objects, whose effects on the CTM are cumulative within a group.)

Primitive objects are *clipped* to the view boundaries as defined by the camera projection attribute. In addition, primitive objects are clipped in the z direction at the *hither* and *yon* clipping planes.

VIEWS, FRAMES, AND DEVICES

CHAPTER TEN

This chapter describes the organizational objects: *views*, *frames*, and *devices*. *Display groups* and *definition groups* are discussed. Two chapter examples show the same display group added to four different views, each with a different definition group. A third example shows combining display objects and definition objects into the same group. Mapping from frame coordinates to device coordinates is also explained. Examples and a discussion of pseudocolor devices are provided at the end of the chapter. Concepts and terms introduced in this chapter include adding objects and views; adding views to frames; updating views, frames, and devices; the device viewport; clipping; and pseudocolor.

Related Functions

Boldface type indicates that the function is used in the chapter examples.

DdInqColorEntries <DDQCE>
DdInqColorTableSize <DDQCTS>
DdInqExtent <DDQE>
DdInqFonts <DDQFT>
DdInqFrame <DDQFR>
DdInqNumFonts <DDQNF>
DdInqShadeMode <DDQSM>
DdInqShadeRanges <DDQSR>
DdInqViewport <DDQV>
DdInqVisualType <DDQVT>
DdSetColorEntries <DDSCE>
DdSetFrame <DDSF>
DdSetShadeMode <DDSSM>
DdSetShadeRanges <DDSSR>
DdSetViewport <DDSDV>
DdUpdate <DDU>
DfInqBoundary <DFQB>
DfInqJust <DFQJ>

Related Functions
(continued)

DfInqViewGroup <DFQVG>
DfSetBoundary <DFSB>
DfSetJust <DFSJ>
DfUpdate <DFU>
DoDevice <DOD>
DoFrame <DOFR>
DoView <DOVW>
DvInqActiveCamera <DVQAC>
DvInqBackgroundColor <DVQBC>
DvInqClearFlag <DVQCF>
DvInqDefinitionGroup <DVQDG>
DvInqDisplayGroup <DVQIG>
DvInqRendStyle <DVQRS>
DvInqBoundary <DVQB>
DvInqUpdateType <DVQUT>
DvSetActiveCamera <DVSAC>
DvSetBackgroundColor <DVSBC>
DvSetBoundary <DVSB>
DvSetClearFlag <DVSCF>
DvSetRendStyle <DVSRS>
DvSetShadeIndex <DVSSI>
DvSetUpdateType <DVSUT>
DvUpdate <DVU>

Organizational Objects

Three organizational objects in Doré include views, frames, and devices. A *view* describes a scene. It is used to collect groups containing *primitive objects* and their attributes with groups containing the *viewing parameters* for those objects (cameras and lights, along with their attributes). Views are collected into *frames*. As shown in Figure 10-1, multiple views can be collected into one frame, much like papers are posted on a bulletin board.

A frame is used to define a virtual image. The frame is assigned, or *set*, to one or more devices. A *device* is an output mechanism used to display a frame. Examples of devices are the video display, an X-window, an image file, a hardcopy device, or a section of memory. A given frame can be set to more than one device. The figure below shows the same frame being set to both an X-window and a section of memory. Note that a device could be a separate physical entity, such as a plotter, or it could be one of several X-windows being displayed on the same screen. **A device can have only one frame attached to it at a time.** (See Figure 10-1.)

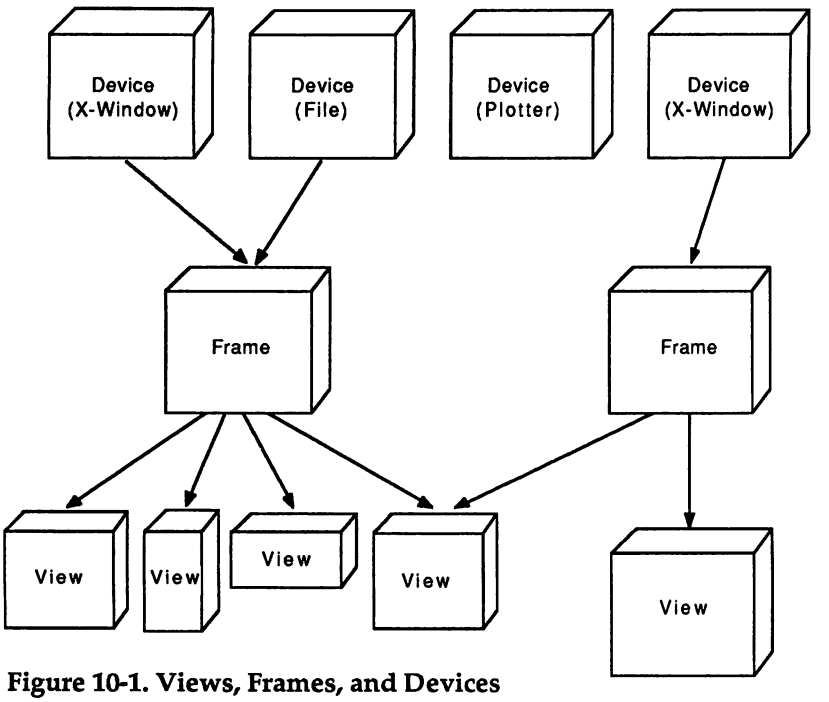


Figure 10-1. Views, Frames, and Devices

Display Groups and Definition Groups

Group objects can be added to a view as either definition objects or display objects. When a view is created with *DoView <DOVW>*, it has an empty display group and an empty definition group. As their names suggest, the *display group* contains the objects that are to be displayed when the view is updated (primitive objects, primitive attribute objects, or groups of those objects). The *definition group* contains the cameras and lights for the view, along with their attributes.

To add display objects to the display group, use *DvInqDisplayGroup <DVQIG>* to obtain the handle to the specified view's display group. For example, to add sphere_group to the display group of view1, the code would be:

```
C code:

DgAddObjToGroup(DvInqDisplayGroup(view1), sphere_group);


Fortran code:

CALL DGAOG(DVQIG(VIEW1), SPHERE);
```

Adding Objects to Views

Definition objects are executed during *rendering initialization*. They include the objects that define the viewing parameters for the scene: the studio objects described in Chapter 9, along with their associated attributes and geometric transformations.

Display objects are executed during *rendering*. They include primitive objects, along with their attributes and geometric transformations.

Usually, you add display objects and definition objects in separate groups (see Examples 2 and 3, below). In certain cases, however, you will want to combine display objects and definition objects into the same group. Example 4, below, shows a spaceship/camera group moving through space. Placing the spaceship and the camera in the same group allows the same geometric transformations to be applied to both the primitive object and the studio object.

Overview of Creating Views, Frames, and Devices

This section presents a brief overview of the steps involved in creating views, frames, and devices. Examples and more detailed explanations are offered later in this chapter.

To render a scene, you will generally complete the following steps. This list is provided mainly as an instructional aid. (You can revise the order of these steps. See Example 1, below, for instance, in which the device is created first. Objects must, however, be created before things are added to them, and the update must occur last.)

(1) Create a view (*DoView <DOVW>*).

(2) Add display group(s) and definition group(s) to the view (*DgAddObjToGroup <DGAOG>*).

(3) Set various view features, such as rendering style, clear flag, background color, view boundary (*Dvxxx* functions).

(4) Create a frame (*DoFrame <DOFR>*).

(5) Add the view to a frame (*DgAddObjToGroup <DGAOG>*).

(6) Set various frame features, such as frame boundary, frame justification (*Dfxxx* functions).

(7) Create a device (*DoDevice* <*DOD*>).

(8) Set device viewport, if necessary (*DdSetViewport* <*DDSDV*>).

(9) Assign frame to a device (*DdSetFrame* <*DDSF*>).

(10) Update the device.

View Groups

When a frame is created with *DoFrame* <*DOFR*>, it has an empty view group. Once you obtain the handle to a frame's view group (with *DfInqViewGroup* <*DFQVG*>), you can add one or more views to the frame. For example, to add the solar_system view and the spaceship view to the same frame (frame1), the code would read:

```
C code:

DgAddObjToGroup(DfInqViewGroup(frame1), solar_system);
DgAddObjToGroup(DfInqViewGroup(frame1), spaceship);


Fortran code:

CALL DGAOG(DFQVG(FRAME1), SOLAR)
CALL DGAOG(DFQVG(FRAME1), SHIP)
```

Example 1

A simple method for creating views, frames, and devices is shown in Example 1 below. The device, a Stardent X11+-window, is created with *DoDevice* <*DOD*>. Then *DdInqExtent* <*DDQE*> is used to obtain the volume range available to that particular device. This volume is used for the frame and for the view, so that they overlap completely.

```
C code:

DtObject device, frame, view;
DtVolume volume;

device = DoDevice("ardentx11","-geometry =640x512+0+0");/* create the device */
DdInqExtent(device, &volume);      /* return device's volume */

frame = DoFrame();                  /* create the frame */
DdSetFrame(device, frame);          /* assign frame to device */
DfSetBoundary(frame, &volume); /* set frame boundary to dev. volume */
```

Example 1
(continued)

```
view = DoView();                    /* create the view */
DvSetClearFlag(view, DcFalse);
DvSetRendStyle(view, DcRealTime);
DgAddObjToGroup(DfInqViewGroup(frame), view);
DvSetBoundary(view, &volume);  /* set view boundary to dev. volume */

/*      .
        .
        .
 create the objects, add the group objects to the view
        .
        .
        .
*/

DdUpdate(device);   /* display the view */


Fortran code:

      INTEGER*4 DEVICE, FRAME, VIEW
      REAL*8 VOLUME(3)
C
      DEVICE=DOD('ardentx11',9,'-geometry=640x512+0+0',22)
C
      CALL DDQE(DEVICE, VOLUME)      !RETURNS DEVICE'S VOLUME!
C
      FRAME=DOFR()    !CREATES THE FRAME!
      CALL DDSF(DEVICE, FRAME) !ASSIGNS FRAME TO THE DEVICE!
      CALL DFSB(FRAME, VOLUME) !SETS FRAME BOUNDARY TO DEV. VOL.!
C
      VIEW=DOVW()
      CALL DVSCF(VIEW, DCFALS)
      CALL DVSRS(VIEW, DCRLTM)
      CALL DGAOG(DFQVG(FRAME), VIEW)
      CALL DVSB(VIEW, VOLUME)  !SETS VIEW BOUNDARY TO DEV. VOL.!
C     CREATE THE OBJECTS; ADD GROUP OBJECTS TO THE VIEW
      CALL DDU(DEVICE)
```

Coordinate Systems

The final transformation from one coordinate system to another involves the clipping, perspective divide, and workstation transformations that transform *frame coordinates* into *device coordinates*. Each of these transformations is described briefly below. Default frame coordinates are from (0.0, 0.0, 0.0) to (1.0, 1.0, 1.0), but they can be set to any convenient volume using *DfSetBoundary <DFSB>*. *Device coordinates* are floating point values in pixels that relate directly to the display device.

Clipping

Clipping actually occurs several times. The first time, discussed in Chapter 9, occurs when the primitive objects are clipped to the view boundary with the camera projection attribute object (*DoParallel* <*DOPAR*>, *DoPerspective* <*DOPER*>, *DoProjection* <*DOPRJ*>, or *DoCameraMatrix* <*DOCMX*>). Clipping occurs a second time when a view is placed within a frame. Any areas of the view that fall outside of the frame boundary are clipped (see Figure 10-2). The third time clipping occurs is when the device viewport is clipped to the device volume.

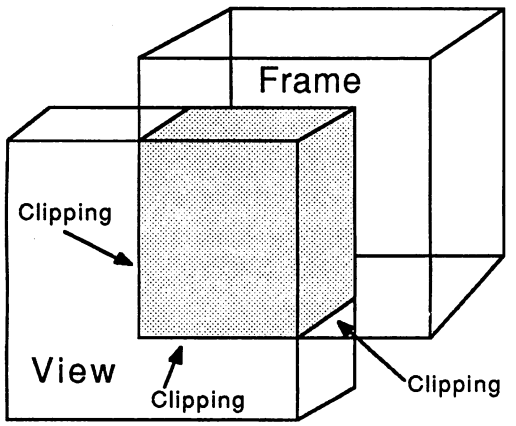


Figure 10-2. Clipping a View to the Frame Boundary

The *device viewport* specifies the portion of the display device that is to be used to display the frame assigned to it. The default viewport uses the entire device volume. If the device extent changes, the device viewport automatically resizes to fill the new device volume. Use *DdSetViewport* <*DDSDV*> to explicitly set the device viewport. When a device is created with *DoDevice* <*DOD*>, you can specify *-noautosize* to turn off autosizing.

Positioning the Frame in the Device Viewport

When a frame is mapped into the device viewport, the frame is sized as large as possible, while preserving its aspect ratio. In cases where the new volume and the device viewport have different aspect ratios, there will be extra white space in the device viewport. Use *DfSetJust* <*DFSJ*> to specify where to put this white space.

Other Transformations

If the camera projection is a perspective projection, another transformation is defined by the camera matrix, called the *z- divide* or *perspective divide.* The transformation itself is actually transparent to the user, but its effect is the *foreshortening* we normally associate with a perspective view (objects closer to the viewer appear larger than objects farther away). The z-divide specifies that each *x,y* point on an object is divided by a value proportional to the distance in *z* between the viewer and the object. You can readily see that when the distance is small, the value of *z* is small. As the distance increases, the *z* value increases, and the *x, y* values become smaller. The parallel camera projection uses *x, y* values and disregards the *z* values (it is therefore the simplest possible projection).

Workstation transformations refers to the transformation of the arbitrary frame coordinate system into the device coordinate system, which consists of floating point pixels directly related to the device extent. Device coordinates range from (0.0, 0.0, 0.0) in the back lower left corner to the device extent in the front upper right corner (for example, 1024.0, 1280.0, 65,536.0).

Priorities

Objects within a group are rendered in the order in which they are added to the group. Similarly, *views* are rendered in the order in which they are added to a frame. If views overlap, the last view to be rendered will still be visible.

Updating Views, Frames, and Devices

DvUpdate <*DVU*> is used to redisplay the specified view. Each view has an update type, set with *DvSetUpdateType* <*DVSUT*>. *DcUpdateAll* <*DCUALL*> updates both the display objects and the definition objects. Use this type if cameras and lights have been modified since the previous update of the view. Use *DcUpdateDisplay* <*DCUDIS*> to update only the display objects. This type is specified if the cameras and lights have *not* been changed since the previous update. Since camera and light groups are not traversed at all, increased efficiency results. *DfUpdate* <*DFU*> updates all views associated with a particular frame. *DdUpdate* <*DDU*> updates the frame associated with a particular device. Use *DvInqUpdateType* <*DVQUT*> to query the view's update type.

Several other important view features are listed below. **Each of these features has a related inquiry function, beginning DvInq-<DVQ-> to inquire the current value.** (See Figure 10-3.)

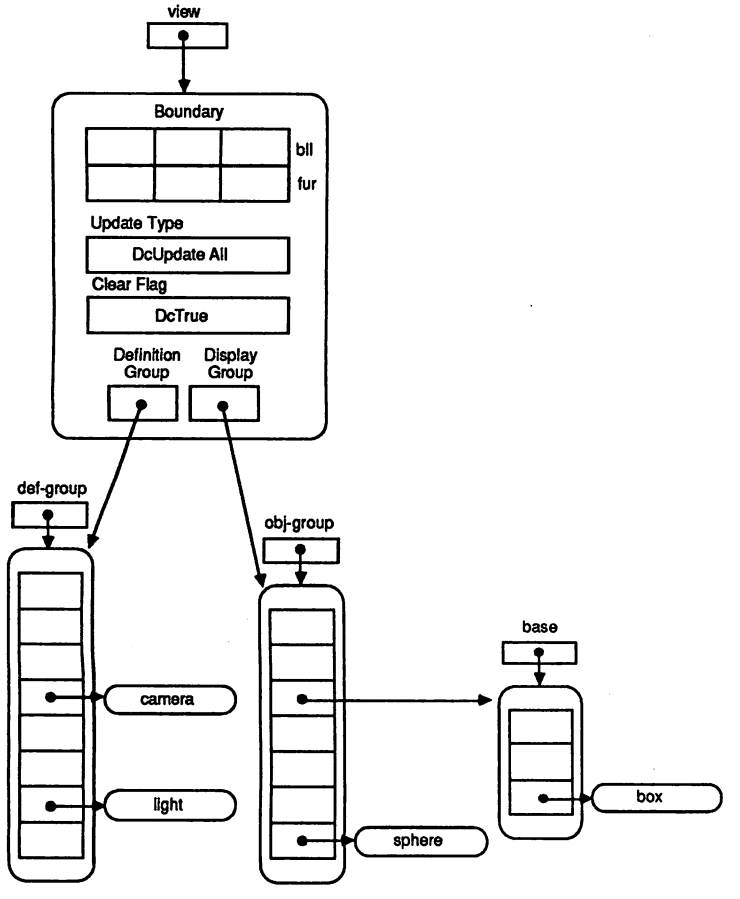


Figure 10-3. Important View Features

Rendering Style

DvSetRendStyle <*DVSRS*> selects the renderer. Most Doré configurations include two renderers: *real time,* for fast, dynamic rendering; or *production time,* for the most realistic rendering, which takes the most time. The default render style is *DcRealTime* <*DCRLTM*>. See Appendex E of the Doré*Reference*Manual" for a list of renderers available for your configuration.

Clear Flag

DvSetClearFlag <*DVSCF*> sets the clear flag of a view. If the clear flag is on, then the view is first cleared to its background color each time the view is updated.

Background Color

DvSetBackgroundColor <*DVSBC*> sets the background color of a view.

View Boundary

DvSetBoundary <*DVSB*> sets the view boundary. The default view boundary is (0.0, 0.0, 0.0) to (1.0, 1.0, 1.0) in view coordinates (which are the same as *frame coordinates*). The default view boundary is identical to the default *frame* boundary. Examples 2 and 3 add four different views to the same frame. Each view has boundaries set with *DvSetBoundary*. The frames in both examples use the default boundaries.

Example 2: Adding an Object to Four Different Views

The device in this example is the Stardent X11+ window. Four views are added to a frame that uses the default boundary, from (0.0, 0.0, 0.0) to (1.0, 1.0, 1.0). Mapping from views to frames is easy because they both use the same (frame) coordinates. Each view in Figure 10-4 uses one-fourth of the frame volume. Figure 10-5 shows the general orientation of the image in each of the four views. (The actual Doré image is shown in Plate 10.)

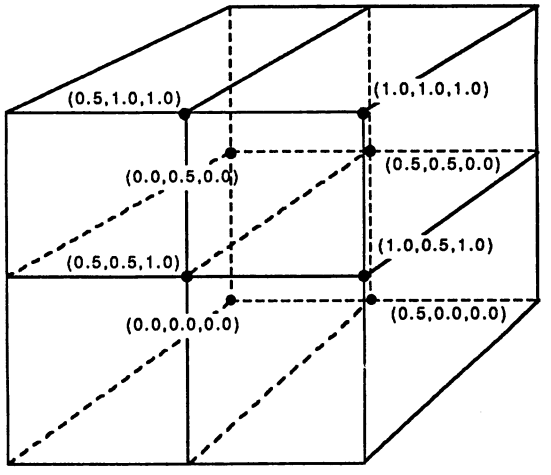


Figure 10-4. Adding Four Views to a Frame

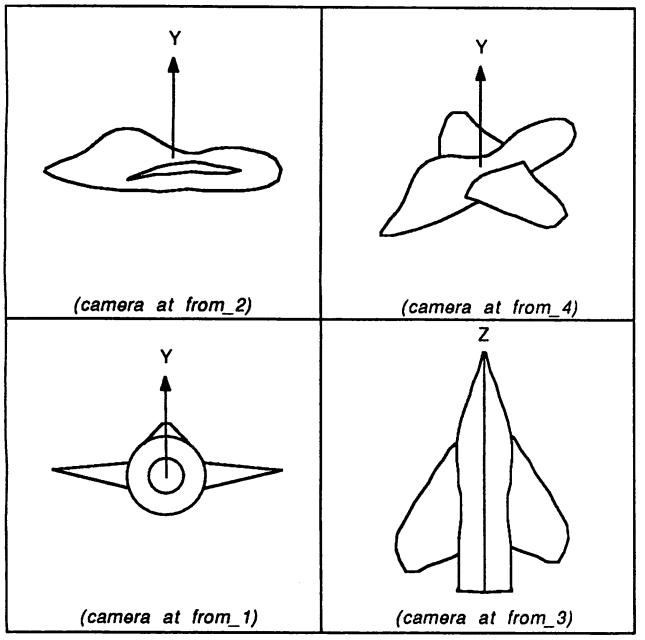


Figure 10-5. Sample Output for Example 2

C code:

```
DtObject plane;      /* patches to make the airplane */
                     /* not included here */
DtObject camera1, camera2, camera3, camera4, view1, view2, view3,
     view4, device, frame;
DtVolume volume;
static DtPoint3
     origin = {0.0, 0.0, 0.0},
     from1 = {0.0, 0.0, 4.0},       /* front view */
     from2 = {4.0, 0.0, 0.0},       /* side view */
     from3 = {0.0, 4.0, 0.0},       /* top view */
     from4 = {4.0, 4.0, 4.0};       /* perspective view */
static DtVector3
     y_dir = {0.0, 1.0, 0.0},
     z_dir = {0.0, 0.0, 1.0};
static DtVolume
     vol_1 = {{0.0, 0.0, 0.0}, {0.5, 0.5, 1.0}},
     vol_2 = {{0.0, 0.5, 0.0}, {0.5, 1.0, 1.0}},
     vol_3 = {{0.5, 0.0, 0.0}, {1.0, 0.5, 1.0}},
     vol_4 = {{0.5, 0.5, 0.0}, {1.0, 1.0, 1.0}};

device = DoDevice("ardentx11", "-geometry =640x512+0+0");
frame = DoFrame();             /* creates a frame with default boundaries */
```

Example 2: Adding an Object to Four Different Views
(continued)

```
DdSetFrame(device, frame);

camera1 = DoGroup(DcTrue);     /* creates first camera group */
     DgAddObj(DoParallel(4.0, -0.1, -100.0));
     DgAddObj(DoLookAtFrom(origin, from1, y_dir));
     DgAddObj(DoCamera());
     DgAddObj(DoLight());
DgClose();

camera2 = DoGroup(DcTrue);     /* creates second camera group */
     DgAddObj(DoParallel(4.0, -0.1, -100.0));
     DgAddObj(DoLookAtFrom(origin, from2, y_dir));
     DgAddObj(DoCamera());
     DgAddObj(DoLight());
DgClose();

camera3 = DoGroup(DcTrue);     /* creates third camera group */
     DgAddObj(DoParallel(4.0, -0.1, -100.0));
     DgAddObj(DoLookAtFrom(origin, from3, z_dir));
     DgAddObj(DoCamera());
     DgAddObj(DoLight());
DgClose();

camera4 = DoGroup(DcTrue);     /* creates fourth camera group */
     DgAddObj(DoPerspective(90.0, -0.1, -100.0));
     DgAddObj(DoLookAtFrom(origin, from4, y_dir));
     DgAddObj(DoCamera());
     DgAddObj(DoLight());
DgClose();

view1 = DoView();
DgAddObjToGroup(DfInqViewGroup(frame), view1);
DvSetBoundary(view1, &vol_1);   /* bottom left quadrant */
DgAddObjToGroup(DvInqDisplayGroup(view1), plane); /* add airplane group */
DgAddObjToGroup(DvInqDefinitionGroup(view1), camera1); /* add camera 1 */

view2 = DoView();
DgAddObjToGroup(DfInqViewGroup(frame), view2);
DvSetBoundary(view2, &vol_2);   /* top left quadrant */
DgAddObjToGroup(DvInqDisplayGroup(view2), plane); /* add airplane group */
DgAddObjToGroup(DvInqDefinitionGroup(view2), camera2); /* add camera 2 */

view3 = DoView();
DgAddObjToGroup(DfInqViewGroup(frame), view3);
DvSetBoundary(view3, &vol_3);   /* bottom right quadrant */
DgAddObjToGroup(DvInqDisplayGroup(view3), plane); /* add airplane group */
DgAddObjToGroup(DvInqDefinitionGroup(view3), camera3); /* add camera 3 */

view4 = DoView();
DgAddObjToGroup(DfInqViewGroup(frame), view4);
DvSetBoundary(view4, &vol_4);   /* top right quadrant */
DgAddObjToGroup(DvInqDisplayGroup(view4), plane); /* add airplane group */
DgAddObjToGroup(DvInqDefinitionGroup(view4), camera4); /* add camera 4 */


Fortran code:
```

```
      INTEGER*4 PLANE ! PATCHES TO MAKE THE AIRPLANE !
      INTEGER*4 CAMERA1, CAMERA2, CAMERA3, CAMERA4
      INTEGER*4 DEVICE, FRAME, VIEW1, VIEW2, VIEW3, VIEW4
      REAL*8 VOLUME (3)
      REAL*8 ORIGIN(3), FROM1(3), FROM2(3), FROM3(3), FROM4(3)
      REAL*8 YDIR(3), ZDIR(3)
      REAL*8 VOL1, VOL2, VOL3, VOL4
      DIMENSION VOL1 (3,2), VOL2(3,2), VOL3(3,2), VOL4(3,2)
C
      DATA ORIGIN / 0.0D0, 0.0D0, 0.0D0 /
      DATA FROM1  / 0.0D0, 0.0D0, 4.0D0 / ! front view !
      DATA FROM2  / 4.0D0, 0.0D0, 0.0D0 / ! side view !
      DATA FROM3  / 0.0D0, 4.0D0, 0.0D0 / ! top view !
      DATA FROM4  / 4.0D0, 4.0D0, 4.0D0 / ! perspective view !
      DATA YDIR   / 0.0D0, 1.0D0, 0.0D0 /
      DATA ZDIR   / 0.0D0, 0.0D0, 1.0D0 /
      DATA VOL1   / 0.0D0, 0.0D0, 0.0D0, 0.5D0, 0.5D0, 1.0D0 /
      DATA VOL2   / 0.0D0, 0.5D0, 0.0D0, 0.5D0, 1.0D0, 1.0D0 /
      DATA VOL3   / 0.5D0, 0.0D0, 0.0D0, 1.0D0, 0.5D0, 1.0D0 /
      DATA VOL4   / 0.5D0, 0.5D0, 0.0D0, 1.0D0, 1.0D0, 1.0D0 /
C
      DEVICE = DOD ('ardentx11',9,'-geometry=640x512+0+0',22)
      FRAME=DOFR() ! creates a frame with default boundaries !
      CALL DDSF (DEVICE, FRAME)
C
      CAMERA1 = DOG (DCTRUE) ! creates first camera group!
         CALL DGAO(DOPAR(4.0D0, -0.1D0, -100.0D0))
         CALL DGAO(DOLAF(ORIGIN, FROM1, YDIR))
         CALL DGAO(DOCM())
         CALL DGAO(DOLT())
      CALL DGCS()
C
      CAMERA2 = DOG(DCTRUE)  ! creates second camera group !
         CALL DGAO(DOPAR(4.0D0, -0.1D0, -100.0D0))
         CALL DGAO(DOLAF(ORIGIN, FROM2, YDIR))
         CALL DGAO(DOCM())
         CALL DGAO(DOLT())
      CALL DGCS()
C
      CAMERA3 = DOG(DCTRUE)  ! creates third camera group !
         CALL DGAO(DOPAR(4.0D0, -0.1D0, -100.0D0))
         CALL DGAO(DOLAF(ORIGIN, FROM3, ZDIR))
         CALL DGAO(DOCM())
         CALL DGAO(DOLT())
      CALL DGCS()
C
      CAMERA4 = DOG(DCTRUE)  ! creates fourth camera group !
         CALL DGAO(DOPER(90.0D0, -0.1D0, -100.0D0))
         CALL DGAO(DOLAF(ORIGIN, FROM4, YDIR))
         CALL DGAO(DOCM())
         CALL DGAO(DOLT())
      CALL DGCS()
C
      VIEW1=DOVW()
         CALL DGAOG(DFQVG(FRAME), VIEW1)
         CALL DVSB(VIEW1, VOL1) ! bottom left quadrant !
```

Example 2: Adding an Object to Four Different Views (continued)

```
            CALL DGAOG(DVQIG(VIEW1), PLANE) ! add airplane group !
            CALL DGAOG(DVQDG(VIEW1), CAMERA1) ! add camera 1!
C
      VIEW2=DOVW()
            CALL DGAOG(DFQVG(FRAME), VIEW2)
            CALL DVSB(VIEW2, VOL2) ! top left quadrant !
            CALL DGAOG(DVQIG(VIEW2), PLANE) ! add airplane group !
            CALL DGAOG(DVQDG(VIEW2), CAMERA1) ! add camera 2!
C
      VIEW3=DOVW()
            CALL DGAOG(DFQVG(FRAME), VIEW3)
            CALL DVSB(VIEW3, VOL3) ! bottom right quadrant !
            CALL DGAOG(DVQIG(VIEW3), PLANE) ! add airplane group !
            CALL DGAOG(DVQDG(VIEW3), CAMERA1) ! add camera 3 !
C
      VIEW4=DOVW()
            CALL DGAOG(DFQVG(FRAME), VIEW4)
            CALL DVSB(VIEW4, VOL4) ! top right quadrant !
            CALL DGAOG(DVQIG(VIEW4), PLANE) ! add airplane group !
            CALL DGAOG(DVQDG(VIEW4), CAMERA1) ! add camera 4 !
```

Example 3

Example 3 is identical to Example 2, except that the boundaries for the four views are specified with relative references rather than absolute references, for added flexibility.

C code:

```
DtObject plane_obj, light_group, camera_groups[2][2];
DtObject view, frame, device;
DtVolume volume;
DtReal Width, Height, Xoff, Yoff, Xstep, Ystep;
DtInt i, j;

device = DoDevice("ardentx11", "-geometry =640x512+0+0");
DdInqViewport(device, &volume);
Width  = volume.fur[0] - volume.bll[0];
Height = volume.fur[1] - volume.bll[1];
Xoff = volume.bll[0];
Yoff = volume.bll[1];
Xstep = Width  / 2.0;
Ystep = Height / 2.0;
frame = DoFrame();
DdSetFrame(device, frame);
DfSetBoundary(frame, &volume); /* to match device viewport */

/* define the light group and the 2x2 array of camera groups here */

/* make views and add them to the frame */
for(i=0; i<2; i++)
          for(j=0; j<2; j++) {
               view = DoView();          /* create a new view */
               DgAddObjToGroup(DfInqViewGroup(frame), view);
```

```
            volume.bll[0] = i * Xstep + Xoff;
            volume.bll[1] = j * Ystep + Yoff;
            volume.fur[0] = (i + 1) * Xstep + Xoff;
            volume.fur[1] = (j + 1) * Ystep + Yoff;
            DvSetBoundary(view, &volume);
            DgAddObjToGroup(DvInqDefinitionGroup(view),
                 camera_groups[i][j]);
            DgAddObjToGroup(DvInqDefinitionGroup(view), light_group);
            DgAddObjToGroup(DvInqDisplayGroup(view), plane_obj);
        }
```

Fortran code:

```
      INTEGER*4 PLNOBJ, LGTGRP, CAMGRP(2,2), VIEW FRAME, DEVICE
      REAL*8 VOLUME (3,2),WIDTH, HEIGHT, XOFF, YOFF, XSTEP, YSTEP
      INTEGER*4 I,J
      include '/usr/include/fortran/DORE'
C
      DEVICE = DOD('ardentx11',9,'-geometry =640x512+0+0'=22)
      CALL DDQV (DEVICE, VOLUME)
      WIDTH=VOLUME(1,2)-VOLUME(1,1)
      HEIGHT=VOLUME(2,2)-VOLUME(2,1)
      XOFF=VOLUME(1,1)
      YOFF=VOLUME(2,1)
      XSTEP=WIDTH/2.0
      YSTEP=HEIGHT/2.0
      FRAME=DOFR()
      CALL DDSF(DEVICE, FRAME)
      CALL DFSB(FRAME, VOLUME)    ! to match device viewport !
C     DEFINE THE LIGHT GROUP AND THE 2X2 ARRAY OF CAMERA GROUPS
C
C     MAKE VIEWS AND ADD THEM TO THE FRAME
      DO 30 I=0,1
            DO 20 J=0,1
                  VIEW=DOVW()
                  CALL DGAOG(DFQVG(FRAME),VIEW)
                  VOLUME(1,1)=I*XSTEP+XOFF
                  VOLUME(2,1)=J*YSTEP+YOFF
                  VOLUME(1,2)=(I+1)*XSTEP+XOFF
                  VOLUME(2,2)=(J+1)*YSTEP+YOFF
                  CALL DVSB(VIEW,VOLUME)
                  CALL DGAOG(DVQIG(VIEW),CAMGRP(J+1,I+1))
                  CALL DGAOG(DVQIG(VIEW),LGTGRP)
                  CALL DGAOG(DVQDG(VIEW),PLNOBJ)
20          CONTINUE
30    CONTINUE
```

Example 4: Combining Definition and Display Groups

Example 4 shows adding a group twice to the same view—once as a definition object, where the studio objects and attributes, along with their geometric transformations, will execute, and a second time as a display object, where the primitive objects and their attributes, and the same geometric transformations, will execute. Primitive objects and primitive attribute objects in a definition group are ignored. Similarly, studio objects and studio attribute objects in a display group are ignored. Figure 10-6 shows the spaceship/camera group orbiting the earth and the observer/camera group located on the surface of the planet.

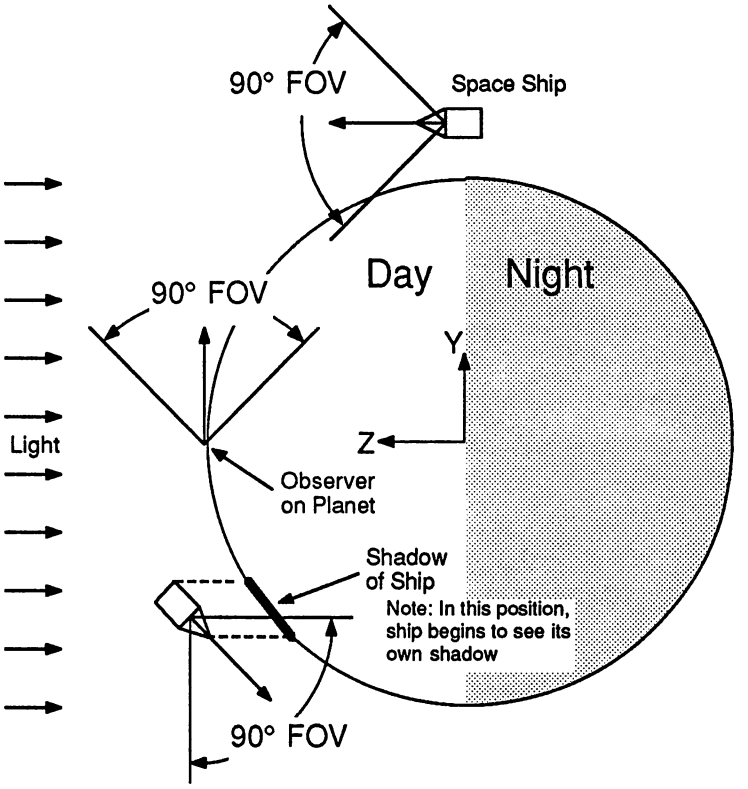


Figure 10-6. Spaceship and Observer Camera Groups

C code:

```
#include "dore.h"

#define DTOR(degrees) (.017453292 * (degrees))
```

```
main()
{
     DtObject device, frame, observer_view, ship_view,
            solar_system, observer_camera, ship_camera;
     static DtVolume
          device_volume,
          frame_volume    =       {{0.0, 0.0, 0.0}, {1.0, 1.0, 1.0}},
          observer_view_volume = {{0.0, 0.0, 0.0}, {0.5, 0.5, 1.0}},
          ship_view_volume     = {{0.5, 0.5, 0.0}, {1.0, 1.0, 1.0}};
     static DtPoint3
          origin          = {0.0,  0.0, 0.0},
          ship_from       = {0.0, 12.0, 0.0},
          ship_at         = {0.0, 12.0, 1.0},
          observer_from   = {0.0,  0.0, 10.1},
          observer_at     = {0.0,  1.0, 10.1},
          light_from      = {0.0,  0.0, 15.0};
     static DtVector3
          y_dir = {0.0, 1.0, 0.0},
          z_dir = {0.0, 0.0, 1.0};
     static DtReal
          red[]     = {1.0, 0.0, 0.0},
          yellow[]  = {1.0, 1.0, 0.0};
     DtInt i;


     DsInitializeSystem(0);

     solar_system = DoGroup(DcTrue);
          /* specify some camera and primitive attributes */
          DgAddObj(DoPerspective(90.0, -0.01, -400.0));
          DgAddObj(DoRepType(DcSurface));
          DgAddObj(DoBackfaceCullable(DcTrue));
          DgAddObj(DoAmbientSwitch(DcOff)); /* outer space look */

          /* make the planet */
          DgAddObj(DoPushMatrix());
               DgAddObj(DoDiffuseColor(DcRGB, red));
               DgAddObj(DoScale(10.0, 10.0, 10.0));
               DgAddObj(DoPrimSurf(DcSphere));
          DgAddObj(DoPopMatrix());

          /* position camera on planet's surface */
          DgAddObj(DoPushMatrix());
               DgAddObj(DoLookAtFrom(observer_from, observer_at, z_dir));
               DgAddObj(observer_camera = DoCamera());
          DgAddObj(DoPopMatrix());

          /* make light source from the sun */
          DgAddObj(DoPushMatrix());
               DgAddObj(DoLookAtFrom(origin, light_from, y_dir));
               DgAddObj(DoLightIntens(1.0));
               DgAddObj(DoLight());
          DgAddObj(DoPopMatrix());

          /* position the spaceship and its onboard camera */
          DgAddObj(DoLabel(1));/* label to find following rotate object */
```

Example 4: Combining Definition and Display Groups
(continued)

```
        DgAddObj(DoRotate(DcXAxis, 0.0));  /* dummy rotate at first */
        DgAddObj(DoTranslate(0.0, 0.0, 12.0));

        /* make the spaceship model */
        DgAddObj(DoPushMatrix());
            DgAddObj(DoDiffuseColor(DcRGB, yellow));
            DgAddObj(DoScale(0.5, 0.5, 1.0)); /* make it slender */
            DgAddObj(DoPrimSurf(DcCone));   /* nose cone */
            DgAddObj(DoTranslate(0.0, 0.0, -1.0));
            DgAddObj(DoPrimSurf(DcCylinder));    /* body */
        DgAddObj(DoPopMatrix());

        /* position camera onboard spaceship looking forward */
        DgAddObj(DoPushMatrix());
            DgAddObj(DoLookAtFrom(ship_from, ship_at, y_dir));
            DgAddObj(ship_camera = DoCamera());
        DgAddObj(DoPopMatrix());
        DgSetElePtrRelLabel(1, 1); /* prepare to replace rotate object */
    DgClose();

    /* set up display environment */
    device = DoDevice("ardentx11", "-geometry =640x512+0+0");
    frame = DoFrame();
    DdSetFrame(device, frame);
    DfSetBoundary(frame, &frame_volume);
    observer_view = DoView();
    ship_view = DoView();
    DvSetBoundary(observer_view, &observer_view_volume);
    DvSetBoundary(ship_view, &ship_view_volume);
    DgAddObjToGroup(DfInqViewGroup(frame), observer_view);
    DgAddObjToGroup(DfInqViewGroup(frame), ship_view);
    DvSetActiveCamera(observer_view, observer_camera);
    DvSetActiveCamera(ship_view, ship_camera);
    DgAddObjToGroup(DvInqDisplayGroup(observer_view), solar_system);
    DgAddObjToGroup(DvInqDefinitionGroup(observer_view), solar_system);
    DgAddObjToGroup(DvInqDisplayGroup(ship_view), solar_system);
    DgAddObjToGroup(DvInqDefinitionGroup(ship_view), solar_system);

    for(i=0; i<720; i++) { /* orbit the planet twice */
        DdUpdate(device);
        DgReplaceObjInGroup(solar_system, DoRotate(DcXAxis, DTOR(i)));
    }

    DsReleaseObj(device);
    DsTerminateSystem();
}


Fortran code:

      PROGRAM MAIN
C
      IMPLICIT NONE
      INCLUDE '/usr/include/DORE'
C
      INTEGER*4 DEVICE, FRAME, OBSVW, SHIPVW, I
```

```
      INTEGER*4 SOLSYS, OBSCAM, SHPCAM
      REAL*8 DEVVOL, FRMVOL(3,2), OBSVOL(3,2), SHPVOL(3,2)
      REAL*8 ORIGIN(3), SHPFRM(3), SHPAT(3)
      REAL*8 OBSFRM(3), OBSAT(3), LTFROM(3)
      REAL*8 YDIR(3), ZDIR(3), RED(3), YELLOW(3)
      PARAMETER (DEGRAD = 0.0174533D0)
C
      DATA FRMVOL / 0.0D0, 0.0D0, 0.0D0,   1.0D0, 1.0D0, 1.0D0 /
      DATA OBSVOL / 0.0D0, 0.0D0, 0.0D0,   0.5D0, 0.5D0, 1.0D0 /
      DATA SHPVOL / 0.5D0, 0.5D0, 0.0D0,   1.0D0, 1.0D0, 1.0D0 /
      DATA ORIGIN / 0.0D0, 0.0D0, 0.0D0 /
      DATA SHPFRM / 0.0D0, 12.0D0, 0.0D0 /
      DATA SHPAT / 0.0D0, 12.0D0, 1.0D0 /
      DATA OBSFRM / 0.0D0, 0.0D0, 10.1D0 /
      DATA OBSAT / 0.0D0, 1.0D0, 10.1D0 /
      DATA LTFROM / 0.0D0, 0.0D0, 15.0D0 /
      DATA YDIR / 0.0D0, 1.0D0, 0.0D0 /
      DATA ZDIR / 0.0D0, 0.0D0, 1.0D0 /
      DATA RED / 1.0D0, 0.0D0, 0.0D0 /
      DATA YELLOW / 1.0D0, 1.0D0, 0.0D0 /
C
      CALL DSINIT(0)
C
      SOLSYS=DOG(DCTRUE)
C     SPECIFY SOME CAMERA AND PRIMITIVE ATTRIBUTES
         CALL DGAO(DOPER(90.0D0, -0.01D0, -400.0D0))
         CALL DGAO(DOREPT(DCSURF))
         CALL DGAO(DOBFC(DCTRUE))
         CALL DGAO(DOAMBS(DCOFF)) !OUTER SPACE LOOK!
C     MAKE THE PLANET
      CALL DGAO(DOPUMX())
         CALL DGAO(DODIFC(DCRGB, RED))
         CALL DGAO(DOSC(10.0D0, 10.0D0, 10.0D0))
         CALL DGAO(DOPMS(DCSPHR))
      CALL DGAO(DOPPMX())
C     POSITION CAMERA ON PLANET'S SURFACE
      CALL DGAO(DOPUMX())
         CALL DGAO(DOLAF(OBSFRM, OBSAT, ZDIR))
         CALL DGAO(OBSCAM=DOCM())
      CALL DGAO(DOPPMX())
C     MAKE LIGHT SOURCE FROM THE SUN
      CALL DGAO (DOPUMX())
         CALL DGAO(DOLAF(ORIGIN, LTFRM, YDIR))
         CALL DGAO(DOLI(1.0D0))
         CALL DGAO(DOLT())
      CALL DGAO(DOPPMX())
C     POSITION THE SPACESHIP AND ITS ONBOARD CAMERA
      CALL DGAO(DOLL(1)) ! label to find following rotate object !
      CALL DGAO(DOROT(DCXAX, 0.0D0))
      CALL DGAO(DOXLT(0.0D0, 0.0D0, 12.0D0))
C     MAKE THE SPACESHIP MODEL
      CALL DGAO(DOPUMX())
         CALL DGAO(DODIFC(DCRGB, YELLOW))
         CALL DGAO(DOSC(0.5D0, 0.5D0, 1.0D0)) ! make it slender !
         CALL DGAO(DOPMS(DCCONE)) ! nose cone!
         CALL DGAO(DOXLT(0.0D0, 0.0D0, -1.0D0))
```

Example 4: Combining Definition and Display Groups
(continued)

```
         CALL DGAO(DOPMS(DCCYL))
      CALL DGAO(DOPPMX())
C
      POSITION CAMERA ONBOARD SPACESHIP LOOKING FORWARD
      CALL DGAO(DOPUMX())
         CALL DGAO(DOLAF(SHPFRM, SHPAT, YDIR))
         CALL DGAO(SHPCAM=DOCM())


      CALL DGAO(DOPPMX())
      CALL DGSEPL(1,1) ! prepare to replace rotate object !
      CALL DGCS()
C     SET UP DISPLAY ENVIRONMENT
      DEVICE=DOD('ardentx11',9,'-geometry =640x512+0+0',22)
      FRAME=DOFR()
      CALL DDSF(DEVICE, FRAME)
      CALL DFSB(FRAME, FRMVOL)
      OBSVW=DOVW()
      CALL DVSB(OBSVW, OBSVOL)
      CALL DVSB(SHPVW, SHPVOL)
      CALL DGAOG(DFQVG(FRAME), OBSVW)
      CALL DGAOG(DFQVG(FRAME), SHPVW)
      CALL DVSAC(OBSVW, OBSCAM)
      CALL DVSAC(SHPVW, SHPCAM)
      CALL DGAOG(DVQIG(OBSVW), SOLSYS)
      CALL DGAOG(DVQDG(OBSVW), SOLSYS)
      CALL DGAOG(DVQIG(SHPVW), SOLSYS)
      CALL DGAOG(DVQDG(SHPVW), SOLSYS)
C
      DO 90 I=0, 719
         CALL DDU(DEVICE)
         CALL DGROG(SOLSYS, DOROT(DCXAX, I*DEGRAD))
   90 CONTINUE
C
      CALL DSRO(DEVICE)
      CALL DSTERM()
```

Pseudocolor

The following discussion and examples introduce the use of pseudocolor devices. This section is provided for the advanced Doré user who needs to install a customized color map for pseudocolor. Because a given pseudocolor device may or may not set up a default color map, the application should *always* set up the color map for a pseudocolor device.

The Doré renderer generates all images in full color. If the output is sent to a pseudocolor device, use *DoDevice* <*DOD*> to indicate that the device uses pseudocolor (include the - *visualtype* parameter). With pseudocolor, the full-color image is converted into a pseudocolor image by one of two methods: *bit compression* or *range intensity mapping*. The *DdSetShadeMode* <*DDSSM*> function

indicates the mode selected: *DcComponent <DCCOMP>* for bit compression, or *DcRange <DCRNG>* for range intensity mapping.

Bit Compression

With bit compression, the full-color value is reduced to an 8-bit value by taking the top three significant bits for red from the full-color value, the top three bits for green, and the top two bits for blue. (See Figure 10-7.) This 8-bit value produces a number between 0 and 255 which is used as an index into the color table.

The following example shows how to set up an RGB color table for bit compression mode. Figure 10-8 illustrates the color table setup.

C code:

```
DtReal entries [256*3]; *p;
p = entries;
for (i=0; i<256; i++) {
     *p++ = (DtReal)(i>>5)/7.;           /* red */
     *p++ = (DtReal)((i>>2) & 0x7)/7.;   /* green */
     *p++ = (DtReal)(i & 0x3)/3.;        /* blue */
}
device = DoDevice("ardentx11","-visualtype DcPseudoColor");

DdSetColorEntries(device, DcRGB, 0, 256, entries);/* sets up color table */

DdSetShadeMode (device, DcComponent);  /* specifies bit compression mode */
```

Fortran code:

```
      REAL*8 ENTRYS(3,256)
      INTEGER*4 DEVICE
C
      DO 30 I=0,255
      ENTRYS(1,I+1)=DFLOAT(I/32)/7.0D0          ! red !
      ENTRYS(2,I+1)=DFLOAT(MOD(I/4),8)/7.0D0    ! green !
      ENTRYS(3,I+1)=DFLOAT(MOD(I,4))/3.0D0      ! blue  !
30    CONTINUE
      DEVICE=DOD('ardentx11',9,'-visualtype DcPseudocolor',25)
      CALL DDSCE(DEVICE,DCRGB,0,256,ENTRYS) ! sets up color table !
      CALL(DDSSM(DEVICE, DCCOMP)    ! specifies bit compression mode !
```

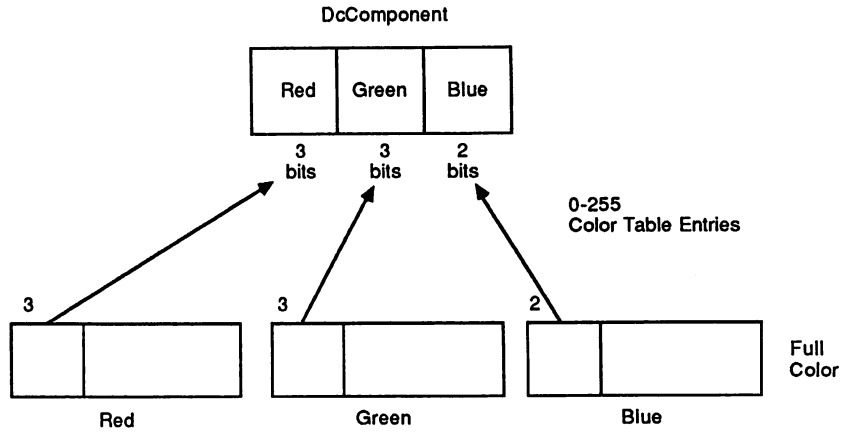


Figure 10-7. Using Bit Compression for Pseudocolor

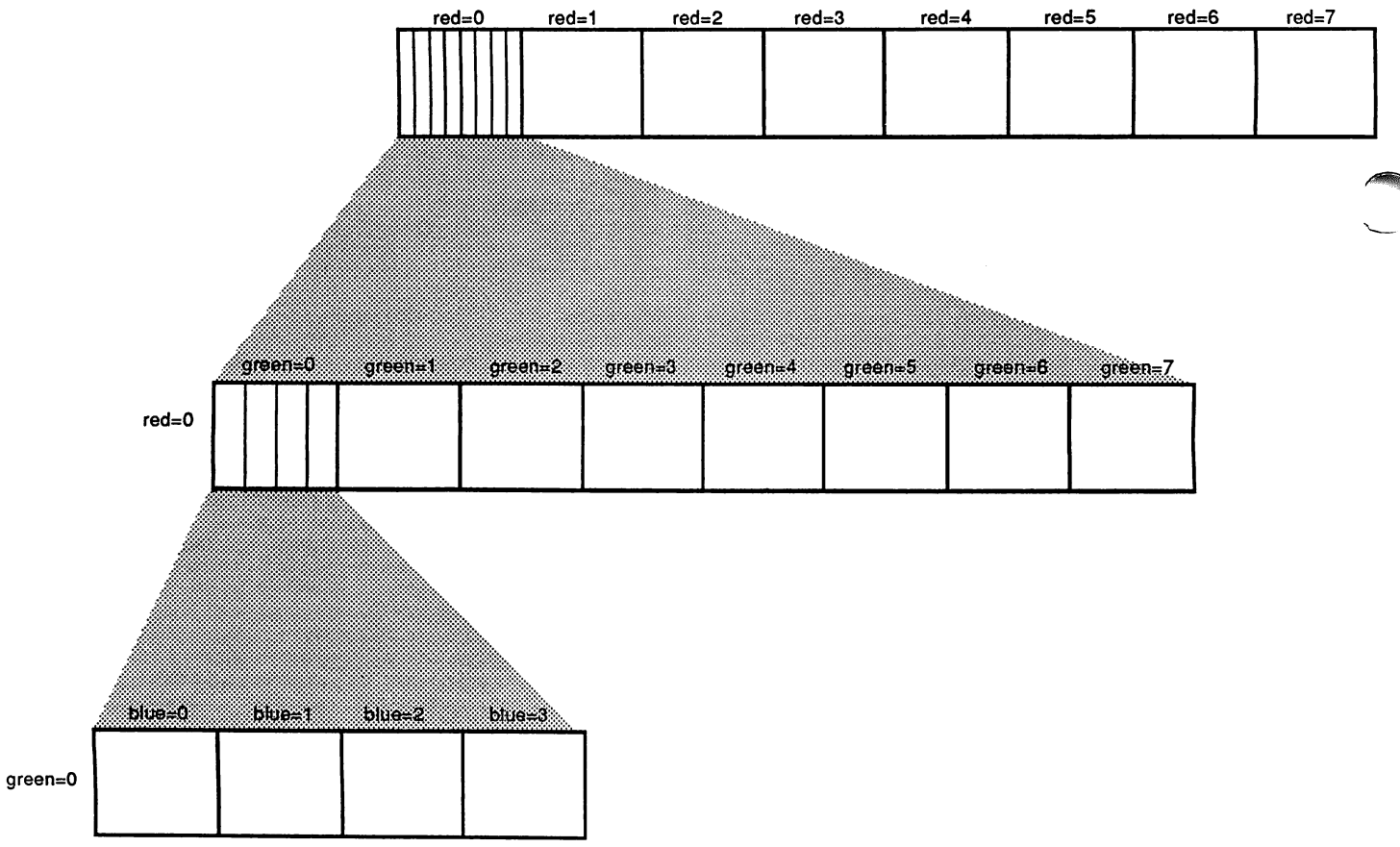


Figure 10-8. Color Table Using RGB Bit Compression

Grey Scale with Bit Compression

The color map created in the example above does not result in a true grey scale (see Figure 10-9). Since red and green values are in sevenths, and blue values are in thirds, it is not possible to get an even value of red/green/blue at any point. (A true grey scale represents a straight line between the black and white corners of the color cube. Here, points in the color cube do not follow that line.)

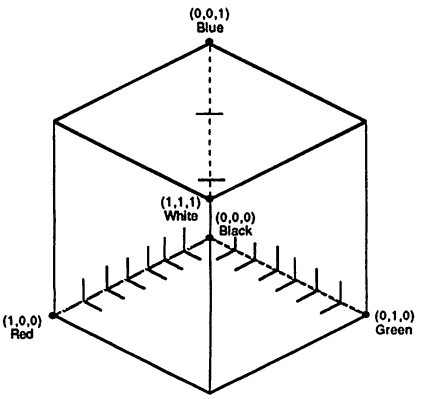


Figure 10-9. Color Cube Using 3-3-2 Bit Compression

The following algorithm would give us a true grey scale, but here we would not have true black, or fully saturated red, green, or blue (those three corners of the cube would be sliced off).

C code:

```
(DtReal)((i>>5) + 1)/8.;
(DtReal)(((i>>2) & 0x7) + 1)/8.;
(DtReal)((i& 0x3) + 1)/4.;
```

Fortran code:

```
        (I/32+1)/8.0D0
        (MOD(I/4,8)+1)/8.0D0
        (MOD(I,4)+1)/4.0D0
```

Range Intensity Mapping

With range intensity mapping, you partition the color map for the device into a series of color ranges. Within each range, the color intensity increases from zero to the maximum intensity for that color. When this mode is used, the renderer converts the full-color value to an intensity value that is mapped to a particular shade range, and then converted to an index into the color table.

For example, in Figure 10-10, the color table is divided into eight shade ranges (0 to 7). Shade range 3 extends from 96 to 127. (Color table entry 96 contains none of this color; entry 127 contains the full intensity of this color.) The renderer converts the original full-color value it computed for the image to an intensity value of *I*, which is between 0 and 1. This value is then multiplied by the number of entries in that color range (here, 31) and then added to the beginning of the range (here, 96) to generate the index into the color table.

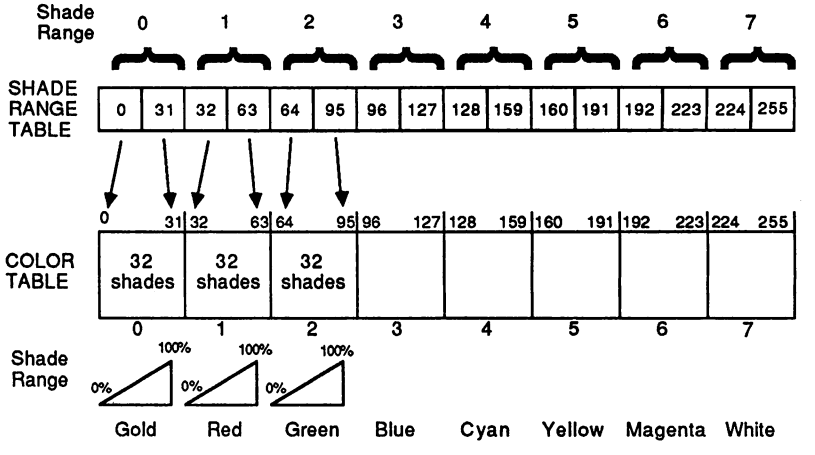


Figure 10-10. Color Table Using Shade Ranges

color table index = I * (127-96) + 96

With range intensity mapping, the display group needs to include a call to *DoShadeIndex <DOSI>*, which specifies the shade range to use for subsequent primitive objects.

The following code illustrates setting up the color table for eight shade ranges, each with 32 values.

C code:

```
DtReal entries [256*3];
DtReal colors [8][3] = {.8, .498, .196, /* gold */
                        1., 0., 0.,     /* red */
                        0., 1., 0.,     /* green */
                        0., 0., 1.,     /* blue */
                        0., 1., 1.,     /* cyan */
                        1., 1., 0.,     /* yellow */
                        1., 0., 1.,     /* magenta */
                        1., 1., 1.};    /* white */
DtInt range [16];
p = entries;
    for(i=0; i<8; i++) {
       for (j=0; j<32; j++){
           *p++ = colors[i][0]*(j/31.);
           *p++ = colors[i][1]*(j/31.);
           *p++ = colors[i][2]*(j/31.);
       }
       range [2*i] = i*32;
       range [2*i +1] = (i+1)*32-1;
    }

device = DoDevice("ardentx11","-visualtype DcPseudoColor");

DdSetColorEntries(device, DcRGB, 0, 256, entries);/* sets up color table */
DdSetShadeMode (device, DcRange);  /* range intensity mapping */
DdSetShadeRanges (device, 0, 8, ranges); /* sets up 8 shade ranges */


DgAddObj(DoShadeIndex(n));    /* tells which shade index to use */
/*
        .
        .
display objects here...
        .
        .
 */
```

Fortran code:

```
      REAL*8 ENTRYS(3,256), COLORS(3,8)
      INTEGER*4 RANGE(16), DEVICE
C
      DATA COLORS / .8D0, .498D0, .196D0,     ! gold !
                    1.D0, 0.D0, 0.D0,     !red !
                    0.D0, 1.D0, 0.D0,     !green !
                    0.D0, 0.D0, 1.D0,     !blue !
                    0.D0, 1.D0, 1.D0,     !cyan !
                    1.D0, 1.D0, 0.D0,     !yellow !
                    1.D0, 0.D0, 1.D0,     !magenta !
                    1.D0, 1.D0, 1.D0 /    !white !
C
      DO 40 I=0,7
      DO 30 J=0,31
```

```
      K=1+32*I+J
      DO 20 L=1,3
20    ENTRYS(L,K)=COLORS(L,I+1)*(J/31.0)
30    CONTINUE
      RANGE(2*I+1)=I*32
40    RANGE(2*I+2)=(I+1)*32-1
C
      DEVICE=DOD('ardentx11',9,'-visualtype DcPseudoColor',25)
      CALL DDSCE(DEVICE, DCRGB,0,256,ENTRYS) ! sets up color table !
      CALL DDSSM(DEVICE, DCRNG) ! range intensity mapping !
      CALL DDSSR(DEVICE,0,8,RANGE) ! sets up 8 shade ranges !
C
      CALL DGAO (DOSI(n))  ! tells which shade range to use !
C     DISPLAY OBJECTS HERE
```

Pros and Cons

There are tradeoffs to consider before choosing one of the two pseudocolor modes. With bit compression, you have a wide spread across the color spectrum but a coarse sampling of shades within each color. With range intensity mapping, you have a smaller selection of colors, but for each color, you can have a full range of intensities. Bit compression is useful for widely varied applications that require many different colors. But if the application requires only a limited number of colors, range intensity mapping produces the best simulation of true color, since each color can have a full range of intensities. Range intensity mapping also allows you to set up a single grey scale, from 0 to 255, for an intensity map that ranges from black to white. With bit compression, vertex shading produces color banding because the samples of color are far apart. To avoid this effect, specify constant shading for the interpolation type.

Chapter Summary

Chapter 10 describes how primitive objects, studio objects, and their attributes are organized into *views* and *frames* and, finally, assigned to a particular *device.*

A *view* is used to combine primitive objects and their attributes with the viewing parameters for those objects (cameras, lights, and their attributes). The *definition group* for a view consists of the cameras, lights, and their attributes. The *display group* for a view consists of the primitive objects and their attributes. View features that can be specified include rendering style, update type, active camera, clear flag, background color, and view boundary.

A *frame* defines a virtual image, which consists of one or more views. Multiple views can be collected into one frame. Views are

rendered in the order in which they are added to a frame. If the views overlap, the last one to be rendered will still be visible. Views and frames both use *frame coordinates*. When a view is placed within a frame, it is *clipped* to the frame boundary.

A *device* is an output mechanism used to display a frame. Examples of devices include the video display, an X-window, an image file, a hardcopy device, or a section of memory. A device can have only *one* frame attached to it at a time. A given frame, however, can be set to more than one device. If extra white space results when a frame is mapped to a device, the frame can be *justified* inside the device viewport to allocate this white space. Device coordinates are three-dimensional floating point values in pixels that relate directly to the display device. For *pseudocolor* devices, you should always set up a color table. With pseudocolor, the full-color image is converted into a pseudocolor image by one of two methods: *bit compression* or *range intensity mapping*.

When a view, frame, or device is *updated*, it is redisplayed. The view *update type* indicates whether all objects within a view should be updated or only the display objects should be updated.

CONDITIONALS

CHAPTER ELEVEN

This chapter discusses the use of conditional elements to affect execution of the Doré database. *DoCallback <DOCB>* causes Doré to call out to user-written functions during execution and to execute the current method on all elements referenced in the callback function (via *DsExecuteObj <DSEO>*). *DsExecutionAbort <DSEA>* and *DsExecutionReturn <DSER>* also affect execution of the Doré database. *DoBoundingVol <DOBV>* allows you to specify a simpler alternate object if a particular object falls below a minimum measurement. The *executability set* allows you to "freeze" certain attributes at their current values so that subsequent attribute objects of particular types do not execute. *Name sets* and *filters* provide high-level control over the invisibility and pickability switches. Concepts and terms introduced in this chapter include the executability set; callback objects; name sets and filters; and bounding volumes.

Related Functions

Boldface type indicates that the function is used in the chapter examples.

DoBoundingVol <DOBV>
DoCallback <DOCB>
DoExecSet <DOES>
DoFilter <DOFL>
DoInvisSwitch <DOINVS>
DoNameSet <DONS>
DoPickSwitch <DOPS>
DsExecuteObj <DSEO>
DsExecutionAbort <DSEA>
DsExecutionReturn <DSER>

Executability Set

DoExecSet <*DOES*> modifies the executability set attribute. This attribute is actually a set consisting of all object types in Doré that are currently eligible to be executed. By default, the executability set consists of all currently defined Doré object types.

Switch Attributes vs. the Executability Set

The difference between switch attributes and the executability set is an important one. When a switch attribute is turned off, its related attribute still pushes and pops its values at group boundaries, but subsequent primitive objects do not *use* this attribute.

The *DoExecSet* <*DOES*> function, on the other hand, actually stops the execution of any object whose type is not included in its set. Subsequent primitive objects still have all attributes and *use* all attributes. But objects whose type is omitted from the executability set *do not execute*. If, for instance, *DoRepType* <*DOREPT*> is omitted from the executability set, the current representation type attribute is frozen at its state when *DoRepType* <*DOREPT*> was deleted from the executability set. If you delete the patch type from the executability set, patches will not execute and thus will not render. Similarly, if *DoInvisSwitch* <*DOINVS*> is turned off, then omitted from the executability set, all primitive objects below this point in the display tree are visible, regardless of subsequent *DoInvisSwitch* <*DOINVS*> attribute objects.

Callbacks

The *DoCallback* <*DOCB*> function allows you to affect the shape of the definition or display tree into which it is added. *DoCallback* <*DOCB*> takes a pointer to a user-written function and a pointer to user data. When *DoCallback* <*DOCB*> is executed, the user function is called and is passed the data. If the callback is used as a conditional, the user function can call *DsExecuteObj* <*DSEO*> with an object. Whatever method was executed on the callback function is executed on any such objects. For example, if the rendering method were executing on the callback function, the rendering method will be executed on the objects executed by the callback function.

Other Functions Affecting Execution

Callback functions can call two others functions that affect execution of the Doré database: *DsExecutionAbort <DSEA>* and *DsExecutionReturn <DSER>*. *DsExecutionAbort <DSEA>* terminates execution of the database. *DsExecutionReturn <DSER>* effectively "prunes" the display tree. This function acts as if it is the last object in its group. The rest of the objects in the group are ignored, and execution of the database continues below that group in the display tree. *DsExecutionReturn <DSER>* can be used by pick callbacks to speed up picking.

Example

The following simple example uses a callback to construct a blinking 3D cursor. The group cursor_obj contains a cursor modeled using three cylinders. This cursor can be moved interactively through the scene. The callback function, illustrated in Figure 11-1, simply executes the cursor every other time the view is rendered, which creates the effect of a blinking cursor.

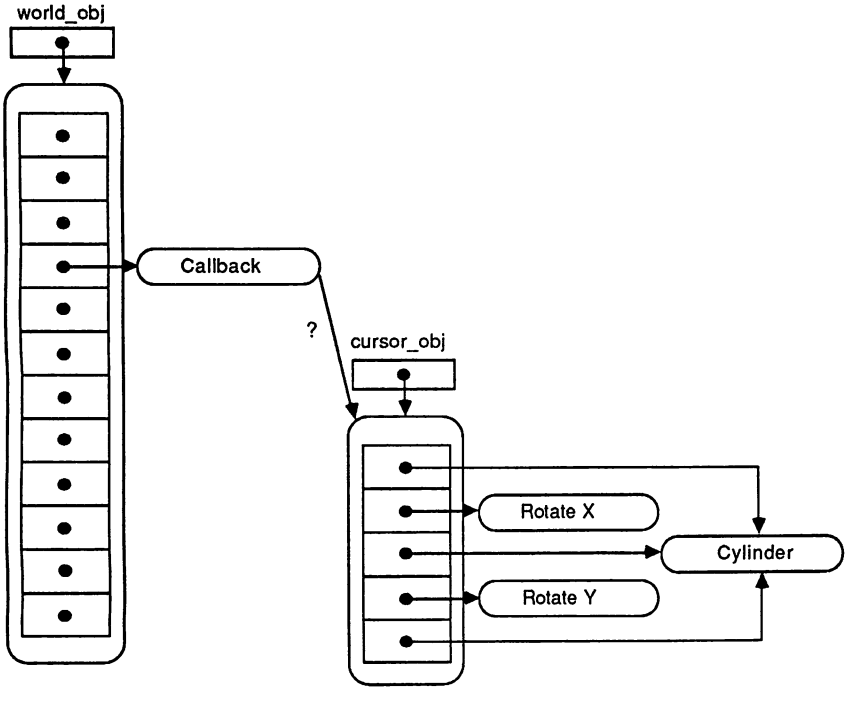


Figure 11-1. Display Tree Including a Callback Object

Example
(continued)

C code:

```
cursor_callback_function(data)
Dt32bits data;
{
      static int blink=1;


      if(blink)
           DsExecuteObj(cursor_obj);
      blink = 1-blink;
}

DtObject make_world()
{
      DoGroup(DcTrue);
        .
        .
 /* add objects */
        .
        .
         DgAddObj(DoCallback(cursor_callback_function,
                  DcNullObject));
        .
        .
      return(DgClose());
}
```

Fortran code:

```
       INTEGER FUNCTION CURSCB(IDATA)
       INTEGER IDATA,BLINK,CRSOBJ
       COMMON /BCTRL/BLINK,CRSOBJ
       DATA /BCTRL/1
C
C       THIS FUNCTION IS USED AS A CALLBACK FUNCTION
C
       IF(BLINK.EQ.1)CALL DSEO(CRSOBJ)
       BLINK=1-BLINK
       RETURN
       END
C
C function using this as callback object
       INTEGER FUNCTION MKWRLD
       EXTERNAL CRSOBJ
C
       IX=DOG(DCTRUE)
          ! ADD OBJECTS !
          .
          .
          CALL DGAO(DOCB(CURSCB,DCNULL))
```

Example
(continued)

```
    .
    .
    ! ADD OBJECTS !
MKWRLD=DGCS()
RETURN
END
```

Name Sets and Filters

Name sets and filters provide high-level control over the invisibility and pickability switches. If you use name sets and filters, then do not also use the low-level switch attribute objects *DoInvisSwitch <DOINV>* and *DoPickSwitch <DOPS>*. When *invisibility* is enabled, primitive objects are not displayed. When *pickability* is enabled, displayed primitive objects are eligible to be picked (see Chapter 12, "Methods").

Invisible Objects

Invisible objects are not pickable. Thus if an object has invisibility enabled, it will not be pickable, regardless of the state of the pickability attribute.

Name Sets

The current name set for a particular point in a scene hierarchy enables you to give symbolic names to the following subtree. You can define the meaning of up to 256 members in the current name set. It may be useful to assign members to indicate certain attributes—for example, objects that have right- and left- handedness, objects made of stainless steel, objects weighing less than 5 grams. It may also be useful to assign members to indicate certain parts of the database—parts belonging to the wheel group, parts of the axle group, the door group, the window group, and so on.

Filters

There are four filters in Doré:

- Invisibility exclusion and invisibility inclusion
- Pickability exclusion and pickability inclusion

The filter attribute object, created with *DoFilter <DOFL>*, is intended to be used at the top of the database tree, so that it affects everything that follows it, but you can change the attribute lower down in the tree if you wish.

Example: Invisibility Filter

This example uses the invisibility filter, but the principles apply to the pickability filter as well. The combination of the two invisibility filters (*inclusion* and *exclusion*) and the current name set indicates whether or not invisibility is enabled.

Invisibility is enabled if

(1) At least one member in the current name set is contained in the *invisibility inclusion* filter **and**

(2) *No* members in the current name set are contained in the *invisibility exclusion* filter

In the following example, the invisibility inclusion filter (at the bottom of the code) contains the hardware, sheets, and equipment members. The invisibility exclusion filter is empty (the default). The result is that the hardware, sheets, and equipment are invisible. Since the hardware group is invisible, objects below it in the tree inherit the invisibility (the cables, pulleys, and cleats). The elements of the sheets group (long and short sheets) are also invisible. The visible parts of the database are the hull and the sails. Figure 11-2 shows the display tree for the sailboat objects. (For the sake of brevity, the example assumes that the main_sail, jib, long_sheets, short_sheets, cables, pulleys, cleats, anchor, life_jackets, deck, keel, and rudder objects have already been created.)

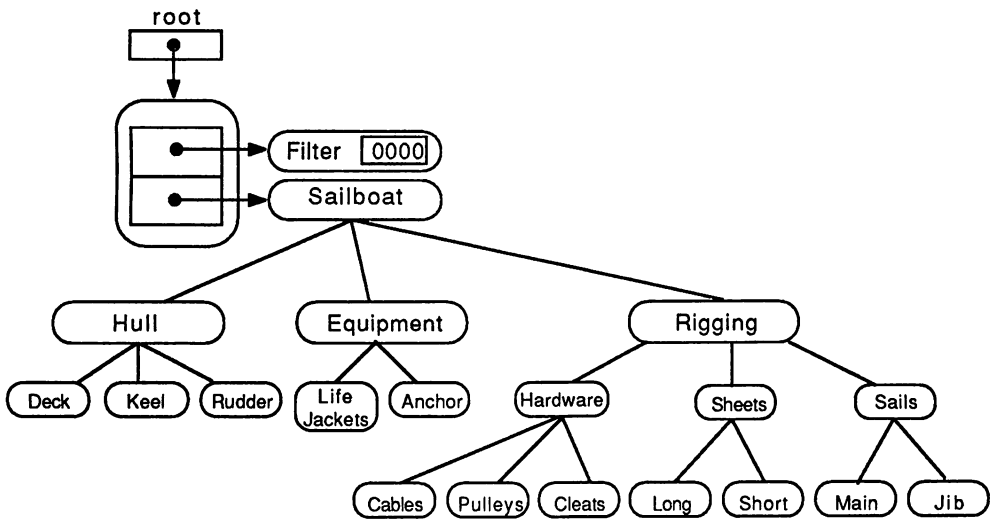


Figure 11-2. Sailboat Display Tree

```
C code:

DtObject equipment, hardware, hull, inv_group, rigging,
         sailboat, sheets, sails;
DtInt temp[7];

temp[0] = SAILS;
sails = DoGroup(DcTrue);
     DgAddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(main_sail);
     DgAddObj(jib);
DgClose();

temp[0] = SHEETS;
sheets = DoGroup(DcTrue);
     DgAddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(long_sheets);
     DgAddObj(short_sheets);
DgClose();

temp[0] = HARDWARE;
hardware = DoGroup(DcTrue);
     DgAddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(cables);
     DgAddObj(pulleys);
     DgAddObj(cleats);
DgClose():

temp[0] = RIGGING;
```

```
rigging = DoGroup(DcTrue);
     DgAddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(hardware);
     DgAddObj(sheets);
     DgAddObj(sails);
DgClose();

temp[0] = EQUIPMENT;
equipment = DoGroup(DcTrue);
     DgAddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(anchor);
     DgAddObj(life_jackets);
DgClose();

temp[0] = HULL;
hull = DoGroup(DcTrue);
     AddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(deck);
     DgAddObj(keel);
     DgAddObj(rudder);
DgClose();

temp[0] = SAILBOAT;
sailboat = DoGroup(DcTrue);
     DgAddObj(DoNameSet(1, temp, DcSetAdd));
     DgAddObj(hull);
     DgAddObj(equipment);
     DgAddObj(rigging);
DgClose();

temp[0] = HARDWARE,
temp[1] = SHEETS,
temp[2] = EQUIPMENT;

inv_group = DoGroup(DcTrue);
     DoFilter(DcInvisibilityInclusion, 3, temp, DcSetAdd);
     DgAddObj(sailboat);
DgClose();
```

Fortran code:

```
      PARAMETER (ISAILS=0, ISHEETS=1, IHARDW=2, IRIG=3, IEQUIP=4
     1           IHULL=5, ISLBT=6)
      INTEGER*4 SAILS, SHEETS, HARDW, RIG, EQUIP, HULL, SLBT
      INTEGER*4 TEMP(0:6)
C
      TEMP(1) = ISAILS
      SAILS = DOG(DCTRUE)
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(MAINS)
         CALL DGAO(JIB)
      CALL DGCS()
C
      TEMP(1) = ISHEETS
      SHEETS = DOG(DCTRUE)
```

```
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(LONGS)
         CALL DGAO(SHORTS)
      CALL DGCS()
C
      TEMP(1) = IHARDW
      HARDW = DOG(DCTRUE)
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(CABLES)
         CALL DGAO(PULLEYS)
         CALL DGAO(CLEATS)
      CALL DGCS()
C
      TEMP(1) = IRIG
      RIG = DOG(DCTRUE)
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(HARDW)
         CALL DGAO(SHEETS)
         CALL DGAO(SAILS)
      CALL DGCS()
C
      TEMP(1) = IEQUIP
      EQUIP = DOG(DCTRUE)
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(ANCHOR)
         CALL DGAO(LIFEJAC)
      CALL DGCS()
C
      TEMP(1) = IHULL
      HULL = DOG(DCTRUE)
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(DECK)
         CALL DGAO(KEEL)
         CALL DGAO(RUDDER)
      CALL DGCS()
C
      TEMP(1) = ISLBT
      SLBT = DOG(DCTRUE)
         CALL DGAO(DONS(1, TEMP, DCSADD))
         CALL DGAO(HULL)
         CALL DGAO(EQUIP)
         CALL DGAO(RIG)
      CALL DGCS()
C
      TEMP(1) = IHARDW
      TEMP(2) = ISHEETS
      TEMP(3) = IEQUIP
C
      INVGRP = DOG(DCTRUE)
         CALL DOFL(DCINVI, 3, TEMP, DCSADD))
         CALL DGAO(SLBT)
      CALL DGCS()
```

Bounding Volumes

A *bounding volume* is an object that specifies a rectangular volume assumed to enclose all objects below that object in the display tree. Bounding volumes can be used to speed up rendering traversal (see Chapter 12, "Methods") and to affect how an object is rendered. This section discusses how a bounding volume conditionally affects the display tree.

A bounding volume object, created with *DoBoundingVol <DOBV>*, specifies how big an arbitrary object is. If the bounding volume switch, specified by *DoBoundingVolSwitch <DOBVS>*, is on (the default), Doré first calculates whether the volume is within the device viewport at all. If it is not, then execution of that portion of the display tree stops, since the objects are not in the viewport.

If the bounding volume is within the device viewport but it falls below a minimum measurement, you can specify a simpler *alternate object* to be substituted for the original, more detailed object. The alternate object should be created the same size as the original object, but with less detail. For example, a detailed aircraft image might acceptably be rendered as a simple rectangle when viewed from a great distance. The minimum measurement, called the *minimum bounding extension,* is specified with *DoMinBoundingVolExt <DOMBVE>*. The default minimum bounding extension is 2 pixels.

```
C code:

obj = DoGroup(DcTrue);
     DgAddObj(DoBoundingVol(&vol, box_obj));
     DgAddObj(airplane_object);
DgClose();

Fortran code:

      OBJ=DOG(DCTRUE)
         CALL DGAO(DOBV(VOL, BOX))
         CALL DGAO(PLNOBJ)
      DGCS()
```

You can also nest levels of alternate objects that have increasing levels of simplicity, as shown in the following code fragment. Here, if complex_plane is below the first minimum bounding extension (15.0), then the alternate object is the plane2 group. Simple_plane, which has less detail than complex_plane, will be drawn if the object is greater than the next minimum bounding extension (5.0). But if the object is smaller than this second minimum size, the alternate object for this second group, the box,

will be drawn instead.

C code:

```
DtObject plane1, plane2, box, complex_plane, simple_plane;

box = DoGroup(DcTrue);
     DgAddObj(DoScale(0.5, 0.5, 0.25));
     DgAddObj(DoPrimSurf(DcBox));
DgClose();

plane2 = DoGroup(DcTrue);
     DgAddObj(DoMinBoundingVolExt(5.0));
     DgAddObj(DoBoundingVol(&vol, box));
     DgAddObj(simple_plane);
DgClose();

plane1 = DoGroup(DcTrue);
     DgAddObj(DoMinBoundingVolExt(15.0));
     DgAddObj(DoBoundingVol(&vol, plane2));
     DgAddObj(complex_plane);
DgClose();
```

Fortran code:

```
      INTEGER*4 PLANE1, PLANE2, BOX, CMPXPN, SIMPN
C
      BOX = DOG(DCTRUE)
         CALL DGAO(DOSC(0.5D0, 0.5D0, 0.25D0))
         CALL DGAO(DOPMS(DCBOX))
      CALL DGCS()
C
      PLANE2 = DOG(DCTRUE)
         CALL DGAO(DOMBVE(5.0D0))
         CALL DGAO(DOBV(VOL, BOX))
         CALL DGAO(SIMPN)
      CALL DGCS()
C
      PLANE1 = DOG(DCTRUE)
         CALL DGAO(DOMBVE(15.0D0))
         CALL DGAO(DOBV(VOL, PLANE2))
         CALL DGAO(COMPXPN)
      CALL DGCS()
```

Chapter Summary

Chapter 11 describes how certain conditionals can be used to affect traversal of the Doré database. The *DoCallback <DOCB>* function takes a pointer to a user-written function and user data. If the callback is used as a conditional, the user function can call *DsExecuteObject <DSEO>* with an object. Such objects are executed with the same method that was used to execute the callback

function itself. Callback functions can also call two other functions that affect traversal of the Doré database: *DsExecutionAbort* <*DSEA*> and *DsExecutionReturn* <*DSER*>.

The *executability set* includes all object types in Doré that are eligible to be executed. If object types are omitted from the executability set, their objects do not execute; they are *frozen* at their state when that object type was eliminated from the executability set.

Name sets and *filters* provide high-level control over the invisibility and pickability switches. Name sets are used to identify portions of the scene hierarchy. Filters are generally used at the top of the database tree. Doré includes four filters: invisibility exclusion, invisibility inclusion, pickability exclusion, and pickability inclusion.

Bounding volume objects are used to speed up rendering traversal of complex scenes. They allow for the dynamic substitution of a simpler version of an object when the image of the complex one would fall below a minimum size.

METHODS

CHAPTER TWELVE

This chapter describes the key execution methods to be used on a Doré database: rendering, picking, and computing bounding volumes. In addition, it describes *immediate mode* execution, in which objects are created and subsequently executed without being stored at all. Concepts and terms introduced in this chapter include hidden surfaces, picking, pick aperture, pick path, hits, pick callbacks, and computing bounding volumes.

Related Functions

Boldface type indicates that the function is used in the chapter examples.

DdInqPickAperture <DDQPA>
DdInqPickCallback <DDQPC>
DdInqPickPathOrder <DDQPPO>
DdPickObjs <DDPO>
DdSetPickAperture <DDSPA>
DdSetPickCallback <DDSPCB>
DdSetPickPathOrder <DDSPPO>
DdUpdate <DDU>
DfUpdate <DFU>
DoBackfaceCullable <DOBFC>
DoBackfaceCullSwitch <DOBFCS>
DoBoundingVol <DOBV>
DoBoundingVolSwitch <DOBVS>
DoHiddenSurfSwitch <DOHSS>
DoInterpType <DOIT>
DoPickID <DOPID>
DoPickSwitch <DOPS>
DoRepType <DOREPT>
DsCompBoundingVol <DSCBV>
DsPrintObj <DSPO>
DvInqRendStyle <DVQRS>

Related Functions
(continued)

DvSetRendStyle <DVSRS>
DvSetUpdateType <DVSUT>
DvUpdate <DVU>

Methods

An execution *method* is a function that performs a specific task relative to an object or a group structure. Two methods, rendering and rendering initialization, have already been discussed in previous chapters. During *rendering initialization*, studio objects and studio attribute objects added to definition groups are executed to set up the scene viewing parameters. During *rendering*, primitive objects added to display groups are executed using the current value for each primitive attribute. All methods are defined for all objects, but in many cases, the method has no effect—for example, when a camera is placed in a display group, its rendering method has no effect. Similarly, the only method for label objects that has an effect is the print method.

Other methods include

- picking (discussed in this chapter)
- computing a bounding volume (discussed in this chapter)
- printing

When you initiate a method on a group hierarchy, it propagates through the hierarchy. Groups execute the current method on their elements. For most methods (with the current exception of printing), when a method is executed on a regular group, the following sequence occurs:

(1) Attributes are pushed when the group is entered.

(2) Elements of the group are executed, in order.

(3) Attributes are popped when the group is exited.

Executing a Stored Database vs. Immediate Execution

There are essentially two modes for traversing, or executing, a database. You should be aware of the advantages and disadvantages of both modes and select the mode best suited to your needs. As described below, the modes are not mutually exclusive, and they can easily be intermixed.

Executing a Stored Database

Executing a stored database involves first creating the objects that make up the database tree. After this database is in place, methods, such as rendering and picking, are executed and the database tree is traversed.

Immediate Execution

With immediate execution, objects are created and then immediately executed. The objects exist only during database traversal. This mode uses *DsExecuteObj* <*DSEO*> to execute objects created on-the-fly which are then automatically deleted. Since object creation is time-consuming, immediate mode is slower than executing a stored database.

Normally, you will have both an application database and a Doré database. Since storing two databases can consume a great deal of memory, you might want to use immediate mode to create pieces of the Doré database when you need them, then discard them as they are used. Another alternative would be to store frequently used parts of the Doré database and create other parts on-the- fly.

Rendering

Since many primitive attribute objects, such as *DoRepType* <*DOREPT*>, *DoInterpType* <*DOIT*>, and *DoSubDivSpec* <*DOSDS*>, affect how a primitive object is rendered, much of the material in this section has already been covered in earlier chapters. (See the section "Affecting an Object's Display Representation" in Chapter 6, "Primitive Attributes.") The following paragraphs on rendering serve mainly as a summary of what you have already learned about rendering.

Update Functions

Rendering is triggered by one of the update functions: *DdUpdate* <*DDU*>, *DfUpdate* <*DFU*>, or *DvUpdate* <*DVU*>. When *DdUpdate* <*DDU*> is called, if the device has a frame attached to it, all views attached to that frame are updated. *DfUpdate* <*DFU*> causes the specified frame to be updated on all devices to which it is attached. *DvUpdate* <*DVU*> causes the specified view to be updated. Each view has an update type, set with *DvSetUpdateType* <*DVSUT*>. The type is either

DcUpdateAll <DCUALL>
which causes all objects, including both definition objects and display objects, to be updated, or

DcUpdateDisplay <DCUDIS>
which updates only the display objects. Specify this type if camera and light information is the same as on previous updates. Use of this mode results in increased efficiency, since only the display groups are traversed at update time.

Rendering Styles

There are currently two renderers provided in standard Doré:

DcRealTime <DCRLTM>
is for fast, interactive display and rendering.

DcProductionTime <DCPRTM>
is the most realistic renderer. This renderer is slower than the *DcRealTime* renderer.

Renderings in which different elements of the scene require information about the scene as a whole—such as shadows and environmental reflections—require the *DcProductionTime* renderer.

Appendix E of the *Doré Reference Manual* lists the renderers available for your configuration.

Rendering Efficiency Measures

You can employ various efficiency measures to speed up the rendering process. A few of these efficiency measures include

- backface culling
- hidden surface switch
- bounding volumes (see the section on "How to Use Bounding Volumes," below)
- use of *DcUpdateDisplay <DCUDIS>* view update type whenever possible (see above under "Update Functions")

Backface Culling

Backface culling, discussed in Chapter 5, is an efficiency measure in which surfaces whose geometric normals point away from the viewer are not drawn at all. When the backface culling switch

(*DoBackfaceCullSwitch <DOBFCS>*) is enabled, and the primitive object is designated as backface cullable (with *DoBackfaceCullable <DOBFC>*), the object will be backface culled before shading and rendering occur.

Hidden Surfaces

If *DoHiddenSurfSwitch <DOHSS>* is on, primitive objects will not render any of their parts that are hidden from the viewer by other objects in the scene. If this switch is off, each object is displayed in group order, and the groups are displayed according to their priority, without regard to which parts are obscured from the viewer by other objects.

Computing a Bounding Volume

Computing a bounding volume is another execution method requiring traversal of the graphics database. *DsCompBoundingVol <DSCBV>* tells an application how big an arbitrary object is by computing a tight volume that encloses a primitive object or a group. This value can then be used for the *volume* parameter of *DoBoundingVol <DOBV>* (see also Chapter 11, "Conditionals," which includes an example of *DoBoundingVol <DOBV>*).

Example

The following example shows the use of *DsCompBoundingVol <DSCBV)* to find out the center of a volume containing a particular object so that you can move the object to the origin and rotate it about its center. The object is scaled so that its maximum dimension is equal to 1.

```
C code:

#define MAX(x,y)  (((x)>(y))?(x):(y))

DtVolume vol;
DtReal depth, width, height, max_dim, scale_fac;
DtObject our_group;

DsCompBoundingVol(&vol, obj);
printf("volume of object = (%f, %f, %f) to (%f, %f, %f)\n",
     vol.bll[0], vol.bll[1], vol.bll[2], vol.fur[0], vol.fur[1],
     vol.fur[2]);

depth = vol.fur[2] - vol.bll[2];
width = vol.fur[0] - vol.bll[0];
height = vol.fur[1] - vol.bll[1];
max_dim = MAX(depth, MAX(width, height));
```

```
scale_fac = 1.0/max_dim;

our_group = DoGroup(DcTrue);
     DgAddObj(DoScale(scale_fac, scale_fac, scale_fac));
     DgAddObj(DoTranslate(-width/2.0, -height/2.0, -depth/2.0));
     DgAddObj(obj);
DgClose();
```

Fortran code:

```
      INTEGER*4 OBJJ, OURGP
      REAL*8 VOL(3,2)
      REAL*8 DEPTH, WIDTH, HEIGHT, MAXDIM, SCFAC
C
      CALL DSCBV(VOL, OBJ)
C
      WRITE(6,99) VOL
99    FORMAT('VOLUME OF OBJECT FROM(',2(D13.6,','),
     X  D13.6') TO (',2(D13.6,','),D13.6,')')
C
      DEPTH=VOL(3,2)-VOL(3,1)
      WIDTH=VOL(1,2)-VOL(1,1)
      HEIGHT=VOL(2,2)-VOL(2,1)
C
      MAXDIM=DMAX(DEPTH, WIDTH, HEIGHT)
      SCFAC=1.0/MAXDIM
C
      OURGP=DOG(DCTRUE)
            CALL DGAO(DOSC(SCFAC, SCFAC, SCFAC))
            CALL DGAO(DOXLT(-WIDTH/2.0D0, -HEIGHT/2.0D0,
     X                -DEPTH/2.0D0))
            CALL DGAO(OBJ)
      CALL DGCS
```

How to Use Bounding Volumes

Bounding volumes are used to provide Doré with a sense of spatial coherence—what objects are *always* near each other, and the extent to which you are interested in the details of a particular object. When used in a scene, bounding volumes can help speed up rendering traversal. If the bounding volume is not within the device viewport, traversal of that part of the database will be halted before the object itself is executed at all. Or, if a particular object or group is less than the minimum bounding extension, a simpler alternate object can be substituted (see Chapter 11, "Conditionals").

To use a bounding volume, make a special group to contain the bounding volume and the object and add the group to the view's display group, as shown below. Then, before Doré executes the object itself, it evaluates the bounding volume to decide (a) if it is

within the device viewport, and if so (b) whether it exceeds the minimum bounding extension.

```
C code:

earth_group = DoGroup(DcTrue);
     DgAddObj(DoMinBoundingVolExt(6.0));
     DgAddObj(DoBoundingVol(DsCompBoundingVol(&volume, earth)),
          DoPrimSurf(DcSphere));
     DgAddObj(earth);
DgClose();

Fortran code:

      EARTH=DOG(DCTRUE)
         CALL DGAO(DOMBVE(6.0D0))
         CALL DGAO(DOBV(DSCBV(VOL, ERTH)),DOPMS(DCSPHR))
         CALL DGAO(ERTH)
      DGCS()
```

DoBoundingVolSwitch

If the bounding volume switch is off, bounding volume calculations will be skipped and bounding volumes will be ignored. The default bounding volume switch is *on*.

Picking

Picking is a method that identifies the drawable primitive objects that are contained in a specified volume (the *pick aperture*) of a Doré device. In applications, a common sequence is to *pick* an object at a particular point using an input device such as a mouse, to highlight that object, and then to reposition it in the scene.

The picking method is triggered by the *DdPickObjs <DDPO>* function, which returns information about what it finds in the pick aperture. During picking, geometric attribute objects are executed just as they are during rendering. Other attribute objects, such as those relating to color, highlights, and shadows, have no effect during picking. All coordinate transformations are performed, and primitive objects are decomposed into points, lines, and triangles which are then transformed into their final values in x, y, and z device coordinates.

If any part of the object is contained within the pick aperture, and if the current *pickability switch* attribute is *on*, the primitive is considered to be *hit*. This process continues for all objects in the scene.

Pick Path

A *pick path* consists of the hit object, plus information about the lineage of the hit object so that the application can uniquely identify it in the display hierarchy. For example, a complex molecule might be made up of many atoms that are all made from the same sphere primitive. If any one atom in the molecule—the iron atom, for example—were *hit*, the pick path for the iron atom would list all the groups along the path to the iron atom, as well as information on *where* in each group to branch to the next lower object in the display tree.

Pick Path Elements

A detailed explanation of the pick path is presented in the following paragraphs.

Each pick path element is a *node* in the display tree along the path to the hit object. Figure 12-1 is a conceptual diagram of a display tree containing a pick path. Note that the sphere object is instanced twice in Group G3, but the pick path specifies the sphere object that was translated.

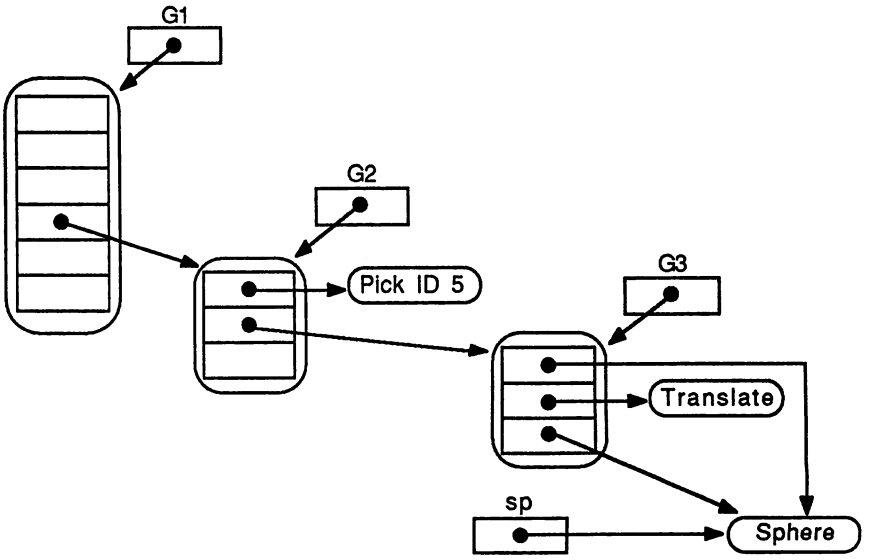


Figure 12-1. Conceptual View of a Pick Path

Each pick path element is actually an integer triple containing the following information, as shown in Figure 12-2:

(1) object handle

(2) pick ID

(3) number of the group element pointing to the next object in the pick path

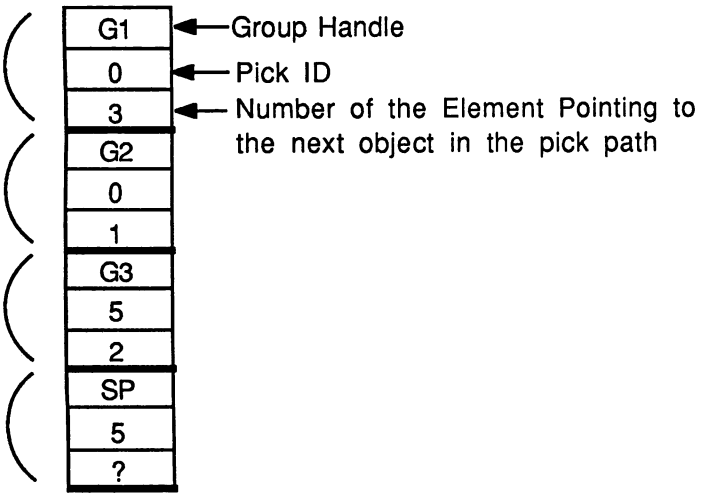


Figure 12-2. Pick Path for the Sphere Object

In Figure 12-2, the first pick path element is comprised of the following information:

(1) The root of the display tree for the hit sphere object is in Group **G1**.

(2) The pick ID is **0** (the default).

(3) Group element **3** (actually the fourth element in the group since group numbering is zero-based) references the next group, G2.

Hit List and Index

DdPickObjs <*DDPO*> writes into the array *hit_list* the pick path for each of the hit objects (see Figure 12-3). You specify the size of this array with the *list_size* parameter. *DdPickObjs* <*DDPO*> also writes the indexes into the hit list into the array *index*. You specify the size of the index array with the *index_size* parameter. *Index* is an array into which *DdPickObjs* <*DDPO*> writes the indices into *hit_list* for the first elements of each pick path. In our example (Figure 12-3), the pick path for the first object begins with element 0 in the hit list. The pick path for the second hit object begins with element 9. The third hit object begins with element 15.

The last pick path element is always one element before the beginning of the next pick path. The first object thus *ends* at index[1] - 1, or 8. The second object ends at index[2] - 1, or 14, and the third object ends at index[3] -1, or 17 (see Figure 12-3). **The index size is thus always at least one greater than the number of hit objects.**

If *index_size* or *list_size* is not large enough to hold information on all hit objects, this fact is recorded in the space pointed to by *error_word*. In this case, you can allocate more memory or make the pick aperture smaller so that less information is returned. In one of these overflow cases, you can also choose to use the hits recorded before the overflow occurred. Partial paths are not included, but *hit_count* always counts the good hits returned.

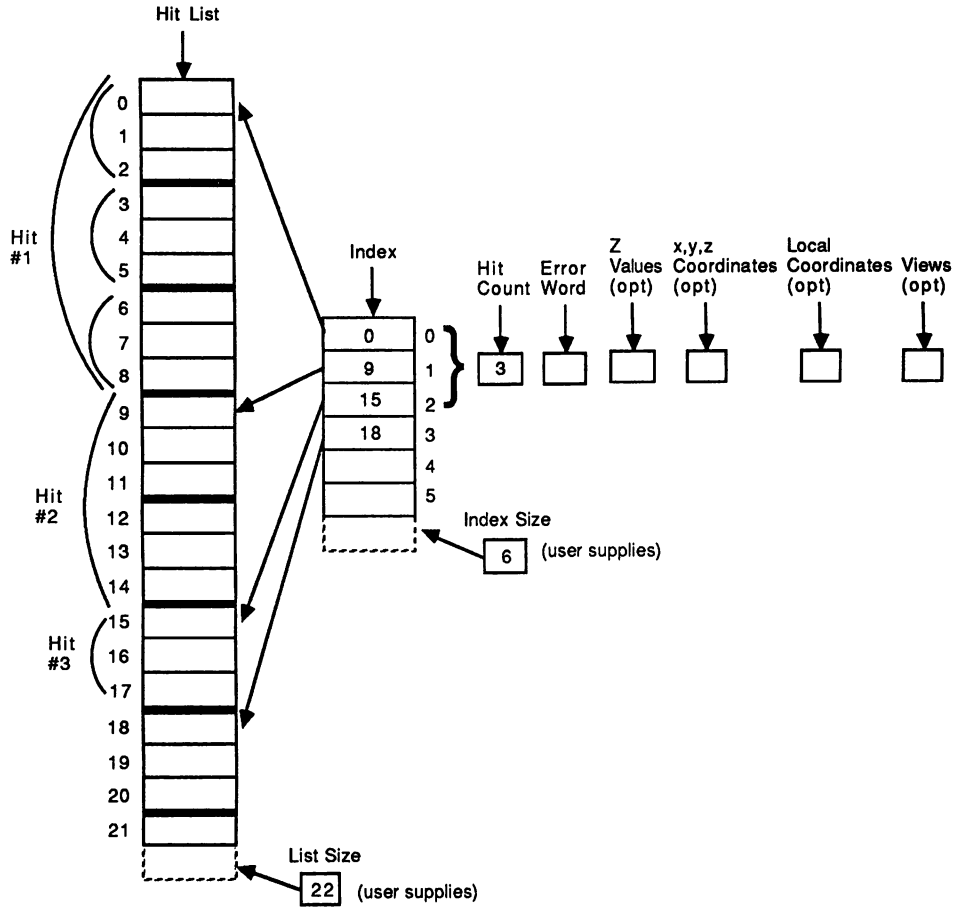


Figure 12-3. Parameters of *DdPickObjs*

Pick Path Order

Each device has a pick path order associated with it, which can be set to either *DcTopFirst* <DCTOPF> or *DcBottomFirst* <DCBOTF> with *DdSetPickPathOrder* <DDSPPO>. The default, *DcTopFirst* <DCTOPF>, is shown in Figure 12-2, which starts at the top of the display tree and ends with the hit primitive object. *DcBottomFirst* <DCBOTF> specifies to order the pick path elements beginning with the hit primitive object and ending with the root. Note that the order of the triples *within* each pick path element is not affected by *DdSetPickPathOrder* <DDSPPO>.

Z Values

DdPickObjs <DDPO> can also return an array of real numbers that lists the closest z value, in picking coordinates, for each of the hit objects in the pick aperture. Picking coordinates are the intersection of the pick volume and the view volume, where 0 is the front of the volume in z, and -1 is the back of the volume. This option is

useful for sorting the hits relative to the viewer. Use *DcNullPtr* <*DCNULL*> for the *z_values* parameter if you do not want information on depth values.

X, Y, Z World Coordinates

World coordinate values is another optional *DdPickObjs* <*DDPO*> pointer to the *x, y, z* world coordinate position for each of the hit objects. This array of DtReals <REAL*8s> should be three times the size of *index*. Use *DcNullPtr* <*DCNULL*> if you do not want this information returned.

Local Coordinates

Local coordinate values is an optional *DdPickObjs* <*DDPO*> pointer to the *x, y, z* local modeling coordinates for each of the hit objects. This array of DtReals <REAL*8s> should be three times the size of *index*. Use *DcNullPtr* <*DCNULL*> if you do not want this information returned.

Views

Views is another optional *DdPickObjs* <*DDPO*> pointer that points to an array of view objects that correspond to the views in which each hit object is found. Use *DcNullPtr* <*DCNULL*> if you do not want information on views here.

Pick ID Attribute

The pick ID attribute is an integer *name* that can be used to tag all objects below it in the display hierarchy. In large databases, it is often useful to use this method to identify different parts of the database—for example, parts belonging to the spaceship, parts belonging to the solar system, and so on. The pick ID is the second piece of each pick path element. An application might want to look at this ID only, not at the objects themselves.

Picking Example

The following example creates three objects, a sphere, a box, and a cylinder, as well as a parallel camera and a light. A pick is initiated with *DdPickObjs* <*DDPO*>, and the pick point is determined by three numbers typed at the keyboard. Information is printed on the screen regarding how many objects were hit. In addition, for each hit, the screen shows the z value, world coordinate position, local coordinate position, which primitive objects were hit,

and their pick IDs.

C code:

```
#include "dore.h"

main()
{
     DtObject device, frame, view, camera_group, object_group, obj, stu,
          sphere_obj, box_obj, cylinder_obj;
     DtVolume volume;
     static DtPoint3
          origin = {0.0, 0.0, 0.0},
          camera_from = {0.0, 0.0, 8.0}, /* positive z */
          light_from = {-8.0, 8.0, 8.0}; /* upper left */
     static DtVector3 up = {0.0, 1.0, 0.0};
     DtPoint3 pick_point;
     DtInt hit_count, index[4], hit_list[100], error, i;

     DtReal z_values[4];
     DtReal wc_values[4*3];
     DtReal lc_values[4*3];
     DtInt j;
     DtInt *Path;
     DtInt Size;
     double dval;

     DsInitializeSystem(0);

     /* make the studio objects */
     stu = DoGroup(DcTrue);
          DgAddObj(DoParallel(6.0, -0.01, -400.0));
          DgAddObj(DoPushMatrix());
               DgAddObj(DoLookAtFrom(origin, camera_from, up));
               DgAddObj(DoCamera());
          DgAddObj(DoPopMatrix());
          DgAddObj(DoPushMatrix());
               DgAddObj(DoLookAtFrom(origin, light_from, up));
               DgAddObj(DoLightIntens(1.0));
               DgAddObj(DoLight());
          DgAddObj(DoPopMatrix());
     DgClose());

     /* make the display objects */
     obj = DoGroup(DcTrue);
          DgAddObj(DoPickSwitch(DcOn));
          DgAddObj(DoPickId(1));
          DgAddObj(sphere_obj = DoPrimSurf(DcSphere));
          DgAddObj(DoTranslate(2.0, 0.0, 0.0));
          DgAddObj(DoPickId(2));
          DgAddObj(box_obj = DoPrimSurf(DcBox));
          DgAddObj(DoTranslate(2.0, 0.0, 0.0));
          DgAddObj(DoPickId(3));
          DgAddObj(cylinder_obj = DoPrimSurf(DcCylinder));
```

```
DgClose();

/* set up display environment */
device = DoDevice("ardentx11", "-geometry =640x512+0+0");
DdInqExtent(device, &volume);
frame = DoFrame();
DdSetFrame(device, frame);
DfSetBoundary(frame, &volume);
view = DoView();
DvSetRendStyle(view, DcRealTime);
DgAddObjToGroup(DfInqViewGroup(frame), view);
DvSetBoundary(view, &volume);
DgAddObjToGroup(DvInqDisplayGroup(view), obj);
DgAddObjToGroup(DvInqDefinitionGroup(view), stu);
DdUpdate(device, DcFalse);

/* do the picking */
printf("device volume is (%f, %f, %f) to (%f, %f, %f)\n",
     volume.bll[0], volume.bll[1], volume.bll[2],
     volume.fur[0], volume.fur[1], volume.fur[2]);
DdSetPickPathOrder(device, DcBottomFirst);
DdSetPickCallback(device, DcPickAll);

while (1) {
   printf ("enter pick point (x = -1 to quit)0);
   scanf("%lf", &dval);  pick_point[0] = dval;
   scanf("%lf", &dval);  pick_point[1] = dval;
   scanf("%lf", &dval);  pick_point[2] = dval;

   if (pick_point[0] == -1) break;

   printf("0******* PICK REPORT ********0);

   printf("   pick point  (%g,%g,%g)0,
        pick_point[0],pick_point[1],pick_point[2]);

   DdPickObjs(device,pick_point,&hit_count,4,index,
            100,hit_list,z_values,wc_values,lc_values,DcNullPtr,&error);

   printf("   accepted    %d hit%c0,
        hit_count, hit_count != 1 ? 's' : ' ');

   for(i=0; i<hit_count; i++) {
      printf ("hit %d      z_value: %g0, i+1, z_values[i]);
      printf ("    world_point: (%g %g %g)0,
           wc_values[3*i],wc_values[3*i+1],wc_values[3*i+2]);
      printf ("    local_point: (%g %g %g)0,
           lc_values[3*i],lc_values[3*i+1],lc_values[3*i+2]);

      Size = (index[i+1]-index[i]) / 3;
      Path = hit_list+index[i];
      for (j=0; j<Size; j++) {
         if ((DtObject)Path[j*3] == sphere_obj)
             printf ("     -> sphere (Pick Id: %d)0, Path[j*3+1]);
         else if ((DtObject)Path[j*3] == box_obj)
             printf ("     -> box (Pick Id: %d)0, Path[j*3+1]);
```

```
            else if ((DtObject)Path[j*3] == cylinder_obj)
                printf ("     -> cylinder (Pick Id: %d)0, Path[j*3+1]);
          }
        }
    }
    DsReleaseObj(device);
    DsTerminateSystem();
}
```

Fortran code:

```
      PROGRAM MAIN
C
      IMPLICIT NONE
      INCLUDE '/usr/include/DORE'
C
      INTEGER*4 DEVICE, FRAME, VIEW
      INTEGER*4 CAMGRP, OBJGRP, OBJ, STU, SPHOBJ, BOXOBJ, CYLOBJ,SIZE
      REAL*8 VOLUME (3,2)
      REAL*8 ORIGIN(3), CAMFRM(3), LTFRM(3), UP(3), PICKPT(3)
      REAL*8 ZVALS(4), WCVALS(12), LCVALS(12)
      CHARACTER*8 CTYPE
      INTEGER*4 HITCT, INDX(4), HITLST(100), ERROR, I
C
      DATA ORIGIN / 0.0D0, 0.0D0, 0.0D0 /
      DATA CAMFRM / 0.0D0, 0.0D0, 8.0D0 /
      DATA LTFRM  / -8.0D0, 8.0D0, 8.0D0 /
      DATA UP     / 0.0D0, 1.0D0, 0.0D0 /
C
      CALL DSINIT(0)
C     MAKE THE STUDIO OBJECTS
      STU=DOG(DCTRUE)
         CALL DGAO(DOPAR(6.0D0, -0.01D0, -400.0D0))
         CALL DGAO(DOPUMX())
            CALL DGAO(DOLAF(ORIGIN, CAMFRM, UP))
            CALL DGAO(DOCM())
         CALL DGAO(DOPPMX())
         CALL DGAO(DOPUMX())
            CALL DGAO(DOLAF(ORIGIN, LTFRM, UP))
            CALL DGAO(DOLI(1.0D0))
            CALL DGAO(DOLT())
         CALL DGAO(DOPPMX())
      CALL DGCS
C     MAKE THE DISPLAY OBJECTS
      OBJ=DOG(DCTRUE)
         CALL DGAO(DOPS(DCON))
         SPHOBJ=DOPMS(DCSPHR)
         CALL DGAO(SPHOBJ)
         CALL DGAO(DOXLT(2.0D0, 0.0D0, 0.0D0))
         BOXOBJ=DOPMS(DCBOX)
         CALL DGAO(BOXOBJ)
         CALL DGAO(DOXLT(2.0D0, 0.0D0, 0.0D0))
         CYLOBJ=DOPMS(DCCYL)
         CALL DGAO(CYLOBJ)
      CALL DGCS
```

```
C     SET UP DISPLAY ENVIRONMENT
      DEVICE=DOD('ardentx11', 9, '-geometry =640x512+0+0', 22)
      CALL DDQE(DEVICE, VOLUME)
      FRAME=DOFR()
      CALL DDSF(DEVICE, FRAME)
      CALL DFSB(FRAME, VOLUME)
      VIEW=DOVW()
      CALL DVSRS(VIEW, DCRLTM)
      CALL DGAOG(DFQVG(FRAME), VIEW)
      CALL DVSB(VIEW, VOLUME)
      CALL DGAOG(DVQIG(VIEW), OBJ)
      CALL DGAOG(DVQDG(VIEW), STU)
      CALL DDU(DEVICE, DCFALS)
C     DO THE PICKING
      WRITE(6,*)'DEVICE VOLUME IS (',VOLUME(1,1),',',VOLUME(2,1),',',
     X VOLUME(3,1),') TO (',VOLUME(1,2),',',VOLUME(2,2),',',VOLUME(3,2),
     X ')'
      CALL DDSPPO(DEVICE, DCBOTF)
      CALL DDSPCB(DEVICE, DCPKAL)
C
10    CONTINUE !DO FOREVER LOOP STARTS HERE!
      WRITE(6,*) 'ENTER PICKPOINT (x=-1.0 TO QUIT)'
      READ(5,*) PICKPT
      IF (PICKPT(1).EQ.-1.0D0) GO TO 20
      WRITE(6,*) '******PICK REPORT******'
      WRITE(6,9901)PICKPT
9901  FORMAT('PICK POINT X,Y,Z=',3D20.13)
C
      CALL DDPO (DEVICE,PICKPT, HITCT, 4, INDX, 100,
     X HITLST, ZVALS, WCVALS, LCVALS, DCNULL, ERROR)
C
      WRITE(6,*)'NUMBER OF HITS ACCEPTED=',HITCT
      DO 15 I=1, HITCT
      WRITE(6,*) I, ZVALS(I),'=ZVALUE'
      WRITE(6,*)'WORLD POINT:', WCVALS(3*I-2), WCVALS (3*I-1),
     X  WCVALS(3*I)
      WRITE(6,*)'LOCAL POINT:', LCVALS(3*I-2), LCVALS(3*I-1),
     X  LCVALS(3*I)
      DO 14 J=INDX(I), INDX(I+1), 3
      IOBJ=HITLST(J)
      CTYPE='           '
      IF(IOBJ.EQ.SPHOBJ)CTYPE='SPHERE'
      IF(IOBJ.EQ.BOXOBJ)CTYPE='BOX'
      IF(IOBJ.EQ.CYLOBJ)CTYPE='CYLINDER"
      WRITE(6,*)'->'CTYPE,'PICKID=',HITLST(J+1)
14    CONTINUE
15    CONTINUE
C
      CALL DSRO(DEVICE)
      CALL DSTERM
      END
```

Pick Callbacks

Each time a hit is detected, but before it is written into the hit list, the current pick callback function on the device is passed all information regarding the hit (pick path, view, z value, local coordinate, world coordinate, hit count). The callback object takes a peek at the information on each hit and decides what to do with the information. Doré provides standard pick callbacks, but you can also provide your own. The standard Doré pick callback objects are:

DcPickFirst <DCPKFR>
the first hit (the default)

DcPickClosest <DCPKCL>
which keeps only the hit closest to the viewer

DcPickAll <DCPKAL>
which adds all hits to the hit list

The *return* value of the pick callback object is important because it tells Doré what to do. The three valid return values are:

DcHitAccept <DCHACC>
DcHitReject <DCHREJ>
DcHitOverwrite <DCHOVW>

DcPickFirst <DCPKFR>, for example, always returns *DcHitAccept <DCHACC>*, then makes a call to *DsExecutionAbort <DSEA>*. *DcPickClosest <DCPKCL>* returns *DcHitOverwrite <DCHOVW>* if the object is closer, otherwise it returns *DcHitReject <DCHREJ>*. *DcPickAll <DCPKAL>* always returns *DcHitAccept <DCHACC>*. In addition, the pick callback object can call *DsExecutionAbort <DCEA>* or *DsExecutionReturn <DSER>*.

User-Written Pick Callbacks

You can also write your own pick callback function—to select only certain kinds or a certain number of objects, for example. The following example shows a user-written callback, my_pick_callback, which accepts only text objects and polymarker objects and rejects all other types of objects.

```
C code:

DtPickControlStatus
my_pick_callback(data, elmts, path, depth, view, hits)
DtObject view;
DtPtr data;
```

```
DtInt elmts, path[], hits;
DtReal depth;
{
     DtInt obj_type;
     obj_type = DsInqObjType(path[3*(elmts-1]); /* obj handle of hit obj */
     if((obj_type==DcTypeText)||(obj_type==DcTypePolymarker))
          return(DcHitAccept);
     return(DcHitReject);
}

/* This call appears in the "main" program, or wherever
   the pick callback is set */

DdSetPickCallback(device, DoCallback(my_pick_callback, DcNullPtr));


Fortran code:


        INTEGER*4 FUNCTION CBXMPL(DATA,ELMTS,PATH,DEPTH,VIEW,HITS)
        INTEGER*4 DATA,VIEW,ELMTS,PATH(1),HITS,IOBJ
        REAL*8 DEPTH
C
        IOBJ=DSQOT(PATH(3*ELMTS-1))     !Object handle of hit object
        CBXMPL=DCHREJ
        IF( (IOBJ.EQ.DCTTXT).OR.(IOBJ.EQ.DCTPMK) ) CBXMPL=DCHACC
        RETURN
        END

C Declarations required and establishment of callback object:
        INTEGER*4 CBXMPL
        EXTERNAL CBXMPL
C
        CALL DDSPCB(DEVICE, DOCB(CBXMPL,DCNULL) )
```

Since one of the arguments passed to the pick callback is the number of hits accepted so far during this pick, you could also write a function that would accepts the first five hits, for example, and then stop. Or you might want to use the *view* argument to select objects depending on which view they are associated with.

In some cases, the pick callback function could do all the work for the application, and the information returned by *DdPickObjs* <*DDPO*> itself would not be used. For example, if the user callback were to find the three closest hits, the callback could maintain an array of the three closest hits and their *z* values. The user callback would handle the management of this array as it evaluates the hits and would write values into the array. At the end, the application would read the values stored in the array by the callback.

Pick paths passed to pick callback functions are always organized *top first*, regardless of the order specified by *DdSetPickPathOrder* <*DDSPPO*>.

Chapter Summary

Chapter 12 describes the main execution methods that involve traversal of the Doré database: rendering, picking, and computing bounding volumes. When a method is initiated on a group hierarchy, it propagates through the hierarchy. There are two modes for executing a database: executing a stored database and immediate execution, which uses *DsExecuteObject* <*DSEO*>. These modes can be intermixed as needed.

Rendering actually consists of two methods: rendering initialization and rendering itself. During *rendering initialization*, the studio objects and their attributes added to definition groups are executed to set up the scene viewing parameters. During *rendering*, primitive objects added to display groups are executed using the current value for each primitive attribute. Rendering is triggered by one of the update functions (*DdUpdate* <*DDU*>, *DfUpdate* <*DFU*>, or *DvUpdate* <*DVU*>).

Standard Doré includes two renderers: a real-time renderer and a highly realistic production time renderer, which uses ray tracing. Efficiency measures that can be used to speed up rendering include setting the view update type, backface culling, the hidden surface switch, and bounding volumes, which provide Doré with a sense of spatial coherence.

Picking, triggered by the *DdPickObjs* <*DDPO*> function, identifies the drawable primitive objects contained in a specified volume of a Doré device. Each time a hit is detected, but before it is written into the hit list, the current pick callback function is passed the information and decides what to do with it. You can use Doré's standard pick callback functions, or you can write your own.

Computing a bounding volume is another method that involves executing the Doré database. *DsCompBoundingVol* <*DSCBV*> computes a tight volume that encloses a primitive object or group. When used in a scene, bounding volumes can help speed up rendering traversal.

SYSTEM FUNCTIONS

CHAPTER THIRTEEN

This chapter shows how you can use *valuators* to make changes in the Doré database in response to asynchronous external events such as mouse or dial movements and keyboard input. Key system functions are also described here. Terms and concepts introduced in this chapter include valuators and input slots.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DoCallback <DOCB>
DoDataPtr <DODP>
DoDataVal <DODV>
DoInputSlot <DOIS>
DsCompBoundingVol <DSCBV>
DsHoldObj <DSHO>
DsInitializeSystem <DSINIT>
DsInputValue <DSIV>
DsInqErrorMessage <DSQEM>
DsInqExeDepthLimit <DSQEDL>
DsInqHoldObj <DSQHO>
DsInqObj <DSQOI, DSQOS>
DsInqObjName <DSQONT; DSQONI, DSQONS>
DsInqObjStatus <DSQVOS>
DsInqObjType <DSQOT>
DsInqSafeFlag <DSQSF>
DsInqValuatorGroup <DSQVG>
DsPrintObj <DSPO>
DsReleaseObj <DSRO>
DsSetExeDepthLimit <DSSEDL>
DsSetObjName <DSSONI, DSSOND, DSSONS>
DsSetSafeFlag <DSSSF>
DsTerminateSystem <DSTERM>
DsUpdateAllViews <DSUAV>
DsValuatorSwitch <DSVS>

Doré Input: Valuators

Doré provides a *valuator* mechanism to handle asynchronous input. This mechanism is one of many ways you can achieve dynamic displays with Doré. Input values can be used to affect the Doré database directly—for example, to change object attributes or geometry. The input values can be generated by application programs, or can be generated asynchronously by external events such as dial or mouse movements.

Modifying the Application Database

Figure 13-1 illustrates the basic computer graphics model. In this model, the application waits for user input. When an input event occurs, the application updates its database and tells Doré to redraw the image. Doré updates the graphics database and then renders the modified image. This standard loop is driven by input events and is under control of the application.

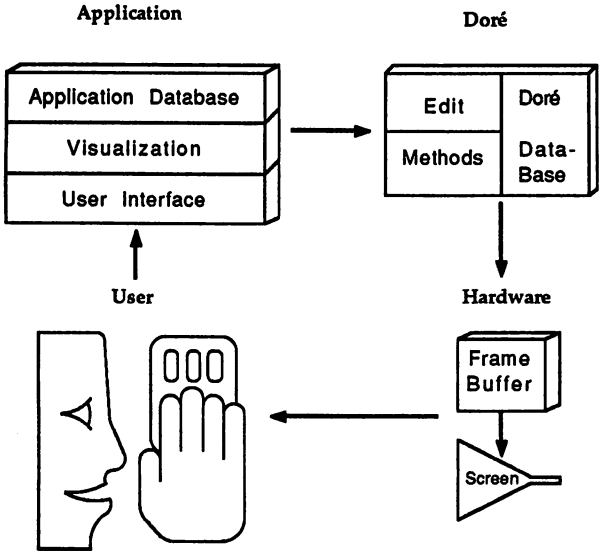


Figure 13-1. Basic Interactive Computer Graphics Model

Modifying the Doré Database Directly

Valuator functions can also be used to modify the Doré database directly, without application control (see Figure 13-2). This capability is provided mainly for compatibility with the "thin-wire" graphics model, in which a low-speed terminal such as the Tektronix 4129 is used. In this model, the RS-232 line is too slow for interactive responses, so the terminal is programmed to do the responses itself. The drawback to this technique is that, since the application is not part of the update loop, it has more trouble finding out how the Doré database has been modified. In this model, an event such as a mouse or dial movement causes a software interrupt, which generates an input value. The input value affects the Doré database, and then Doré renders this modified image.

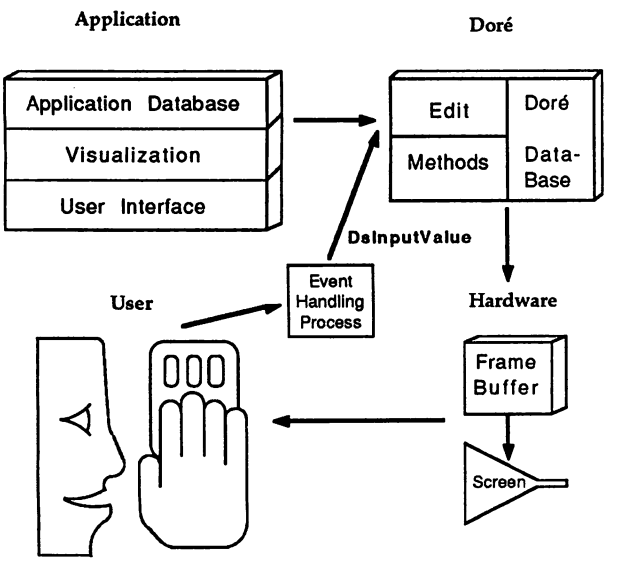


Figure 13-2. Using Valuators to Modify the Doré Database

Input Slots

An *input slot* is an organizational object that contains a *valuator group*. For convenience, Doré provides a set of standard input slots for translate, rotate, and scale values. You can create other input slots using *DoInputSlot <DOIS>*.

Valuator Group

A valuator group contains a set of callback functions associated with a particular input slot. When a value is input to a slot with *DsInputValue <DSIV>*, the callback objects contained in the valuator group for that slot are activated. This means that the valuator functions embedded in the callback objects are passed (1) the data with which the callback was created, (2) the slot, and (3) a value to be input to that slot.

Example of a Simple Valuator

The following example uses one of the predefined Doré input slots, *DcScaleXSlot <DCSSX>*. This input slot has one valuator function associated with it, change_xscale. When *DsInputValue <DSIV>* inputs a value to *DcScaleXSlot <DCSSX>*, the valuator function change_xscale is called and is passed the slot, *DcScaleXSlot <DCSSX>* and the input value, 12.9. (In this example, no data is passed.) The input value is used as the new x scaling parameter for the sphere group.

```
C code:

change_xscale(data, slot, value)
DtPtr data;
DtObject slot;
DtReal value;
{
     DgReplaceObjInGroup(sphere_group, DoScale(value, 1.0, 1.0));
}

callback_obj = DoCallback(change_xscale, DcNullObject);
DgAddObjToGroup(DsInqValuatorGroup(DcScaleXSlot), callback_obj);
DsInputValue(DcScaleXSlot, 12.9);
```

Fortran code:

```
      SUBROUTINE CHGXSC(DATA, SLOT, VALUE)
      INTEGER*4 SLOT, SPHGRP
      REAL*8 VALUE
C
      CALL DGROG(SPHGRP, DOSC(VALUE, 1.0D0, 1.0D0))
      RETURN
      END
C
      CBOBJ= DOCB(CHGXSC, DCTNUL)
      CALL DGAOG(DSQVG(DCISSX), CBOBJ)
      CALL DSIV(DCISSX, 12.9D0)
      END
```

DsSetErrorVars

DsSetErrorVars <DSSEV> enables you to install an error file and a user-written error handler. With this function, you can pass a file descriptor to Doré so that errors will be written to that file. By default, errors are written to the screen. The Doré Library also provides a default error handler. *DsInqErrorMessage <DSQEM>*, a related function, returns a text string associated with an error number and is intended to be used by user-written error handling functions.

Other System Functions

DsInitializeSystem <DSINIT> is the first Doré function called and specifies how many processors to use in real-time rendering on a multiprocessor system. *DsTerminateSystem <DSTERM>* is the last Doré call.

Chapter Summary

Chapter 13 describes how *valuators* can be used to modify the Doré database directly, without application control. Doré provides a set of standard input slots for translate, rotate, and scale values, and you can create additional input slots using *DoInputSlot <DOIS>*. Each input slot contains a valuator group composed of a set of *callback functions*. When a value is input to a slot with *DsInputValue <DSIV>*, the callback objects contained in the valuator group for that slot are activated.

Another important system function is *DsSetErrorVars <DSSEV>*, which enables you to install an error file and a user-written error handler.

USER-DEFINED PRIMITIVES

CHAPTER FOURTEEN

This chapter explains why you might want to create a new Doré primitive. It describes the primitives currently available in Doré and outlines in detail the components of a Doré primitive. Steps for creating a user-defined primitive are explained, along with simple examples. Conventions for naming and implementing user-defined primitives are recommended so that users can share a common approach to developing Doré extensions.

Concepts and terms introduced in this chapter include base primitives, alternate representations, class identifiers, naming conventions for user-defined primitives, and implementation conventions for user-defined primitives.

Related Functions

Boldface type indicates that this function is used in the chapter examples.

DeAddClass <DEAC>
DeCreateObject <DECO>
DeDeleteObject <DEDO>
DeExecuteAlternate <DEEA>
DeInitializeObjPick <DEIOP>
DeInqPickable <DEQP>
DeInqRenderable <DEQR>
<DEDOD> (Fortran only)
<DEROD> (Fortran only)
<DEWOD> (Fortran only)
DsInqClassId <DSQCI>

Why Define Your Own Primitives?

Doré comes with a standard set of primitives, described in Chapter 5. If an application requires only standard primitive types, or groupings of them, there is no need to create new primitives. Once you have defined a group of objects, you can refer to it

as necessary. For example, you could create one bolt and then use that bolt a number of times in a wheel group. Then when modeling a car you would reference the basic wheel group four times. If the same grouping is used repeatedly by many applications, you could generate the group in a subroutine and have the different applications use the same subroutine.

On the other hand, you may want to consider adding your own primitive if an application requires objects whose shape or function cannot be defined by a simple combination of existing primitives—a special kind of curve, for example.

Another case where you probably want to add a new primitive is if you often create objects that have things in common, but differ on parameters that are known only at creation time. For example, you could define a primitive that defines a general L-bracket shape whose dimensions are specified when objects are created.

A third case where you might want to add a new primitive is if you want a variable parameter that can only be determined during traversal of the database to affect the shape or behavior of an object. For example if you want an object to be cube shaped or sphere shaped depending on the state of a button in the user interface, you could add a primitive whose render method would check the the state of the button and change the the representation accordingly.

Or you might want a curve that is a curved line when rendered with the dynamic renderer and a curved tube when rendered with the production renderer. In this case, a simple subroutine is not sufficient, since you want *either* a line list *or* a group of cylinders, depending on the render method.

In summary, when the shape or behavior of an object is not the same as that of any existing Doré primitive or group of primitives, user-defined primitives are the solution.

Base Primitives

Each Doré renderer can directly render only a subset of the standard Doré primitives. This subset of directly drawable primitives for a given renderer is referred to as the *base primitives* for that renderer. All other primitives compile *alternate representations*, which are ultimately compiled into and rendered as base primitives. The set of base primitives varies with the type of renderer and the graphics hardware. For example, the dynamic renderer on the Stardent 1500/3000 cannot directly draw spheres. The

Stardent 1500/3000 does not have hardware support for spheres, and it would take too long to do in software. Instead, a triangle mesh approximating the sphere is compiled and rendered. All primitives that are not directly drawable by a renderer will compile one or more alternate geometry objects. The same primitive may compile different objects for different renderers. The sphere, for example, does not need an alternate representation for the production renderer, as it can directly draw spheres.

The base primitive types for the dynamic renderer, as implemented on the Stardent 1500/3000, are:

point list
line list
connected line list
triangle list
triangle mesh

For dynamic rendering, all other Doré primitives compile into one or more of these.

The base primitive types for the Doré standard production renderer are:

sphere primitive surface
cylinder primitive surface
box primitive surface
cone primitive surface
triangle list
triangle mesh

Other Doré primitives compile triangle lists or meshes, if possible. Lines and points are currently not drawn by the standard production renderer.

This chapter describes how to add primitives that compile alternate objects using the set of base primitives directly drawable by each renderer. Adding base primitives to the renderers is not discussed.

Alternate Representations

At this time, all user-defined primitives need to compile an *alternate representation* of themselves. They are not drawn directly by the renderer. In most cases, the alternate representation compiled by a primitive will be an object of one of the base primitive types directly drawable by a renderer.

The alternate representation does not have to be a base primitive, however. It can be a primitive object that in turn will compile an alternate representation. For example, an L-bracket primitive might generate a simple polygon mesh, which in turn generates a triangle list.

The alternate representation may also be a group of objects. For example, a primitive that has a spherical part and a cylindrical part might create an alternate representation that is a group containing a primitive surface sphere object and a primitive surface cylinder object.

An object can also have more than one alternate representation, as in the example mentioned above of a primitive that could be represented as a curved line when rendered by the dynamic renderer and as a curved tube when rendered by the production renderer.

If the representation does not change during the lifetime of the object, the alternate representation may be compiled when the object is created. If different representations are required depending on some parameter determined at traversal time, they must be recompiled at traversal time, at least if the value of the parameter has changed. (See also the section on "Efficiency Improvements," below.)

Components of a Doré Primitive

Each Doré object is a collection of private data and a set of methods that operate on the data. Specifically, each Doré primitive consists of the following elements:

- Private data
- A class identifier
- An initialization routine
- A creation routine, including creation of the alternate representation
- A set of methods

Private Data

For primitive objects, the private data is the three-dimensional shape information, or information sufficient to compile the three-dimensional structure.

Class Identifier

Each standard Doré primitive has a class identifier that can be queried with the *DsInqClassId* <*DSQCI*> function. The implementor of a user-defined primitive provides a global variable that can be accessed by all the routines for that primitive. The primitive class identifier is a number that is assigned for that class by Doré to the global variable. When application programs need to access the class identifier, for use with functions such as *DoExecSet* <*DOEX*>, *DoNameSet* <*DONS*>, and *DoFilter* <*DOFL*>, the function *DsInqClassId* <*DSQCI*> returns the class identifier.

Initialization Routine

Doré uses symbolic method names. Each of these symbolic method names needs to be associated with a routine for the new primitive. (Recall from Chapter 12 that a method is a function defined across all classes. Each method has one symbolic name used across all classes, and each class has its own implementation of the method.)

The initialization routine tells Doré the actual routine names that the user-defined primitive uses for each method invoked during traversals of the graphical database (see also the subsection below on "Methods"). An array sets up the correspondence between the Doré symbolic method names and the actual routine names for the new primitive.

Creation Routine

The creation routine is called by the application program. It returns a handle to the created object, just like the routines that create standard Doré objects. The function *DeCreateObject* <*DECO*> is used to create an internal Doré object. The alternate object is also defined at this time, but it may not actually be compiled until later (see "Efficiency Improvements" later in this chapter).

Methods

Each Doré method has a symbolic name that is the same for all primitives. The implementation of a method depends on the primitive type. The Doré symbolic names for the primitive methods are as follows:

DcMethodCmpBndVolume <DCMCBV>
Compute bounding volume box. Used by *DsCompBoundingVolume <DSCBV>*.

DcMethodDestroy <DCMDST>
Deallocate space used by an object and its private data.

DcMethodDynRender <DCMDR>
Dynamic rendering.

DcMethodGlbrndIniObjs <DCMGIO>
Standard production rendering.

DcMethodPick <DCMPCK>
Determine if an object has been picked.

DcMethodPrint <DCMPRT>
Print information about an object. Used by *DsPrintObj <DSPO>*.

DcMethodUpdStdAltObj <DCMSAO>
Updates the alternate object.

DcMethodStdRenderDisplay <DCMSRD>
Executes the alternate object (updates it if necessary).

Note that *DcMethodUpdStdAltObj <DCMSAO>* and *DcMethodStdRenderDisplay <DCMSRD>* are required methods, since other parts of the Doré system may rely on them.

When adding a new primitive type, you implement routines for each of these methods. The symbolic method names are associated with the routine names when the primitive type is initialized. In some cases the same routine can be used for more than one method. In other cases a null routine can be used.

All the routines take one argument of type (*DtObjectStructure* *); in Fortran, an INTEGER*4. *DtObjectStructure* is the structure of objects as used internally in the Doré system. It should not be directly manipulated by application programs. Some of the routines will merely pass the work on to the alternate object using *DeExecuteAlternate <DEEA>*. In some cases, they may do some processing first. In others, they may check for certain conditions, and perhaps execute different alternate objects depending on the result. For example, they may check the state of an interactive input device.

A Simple Example: The L-bracket Primitive

The steps necessary to create a user-defined primitive are illustrated below with both C and Fortran code examples. This new primitive lets you create L-bracket objects, with the dimensions specified at object creation time. The implementation of a user-defined primitive involves six basic steps. Each step is described in detail.

(1) Define the private data.

(2) Implement an initialization routine for the primitive type.

(3) Implement a routine to create an object of the new primitive type.

(4) Implement the compilation of the alternate representation.

(5) Implement the methods that operate on the data.

(6) Install the new primitive type.

Step 1: Define the Private Data

The actual structure of an object's private data depends, of course, on the nature of the primitive. A Doré line list, for example, stores the line count, vertex information (type, locations, normals, colors), and bounding box information.

In addition to the 3D shape information, all user-defined primitive objects also store at least one pointer to an alternate representation object.

For example, the private data of an L-bracket primitive could look like this:

C code:

```
struct Lbracket {
        DtReal side1;
        DtReal side2;
        DtReal width;
        DtReal thickness;
        DtObject alternate_object;

 };
```

Fortran code:

```
C.....  THIS IS THE INCLUDE FILE LBRACKET_DATA.
        REAL*8 LBRACK(5)
        REAL*8 SIDE1, SIDE2, WIDTH, THICK
        INTEGER*4 ALTOBJ
        EQUIVALENCE (SIDE1, LBRACK(1)), (SIDE2, LBRACK(2))
        EQUIVALENCE (WIDTH, LBRACK(3)), (THICK, LBRACK(4))
        EQUIVALENCE (ALTOBJ, LBRACK(5))
```

Figure 14-1 shows the L-bracket parameters.

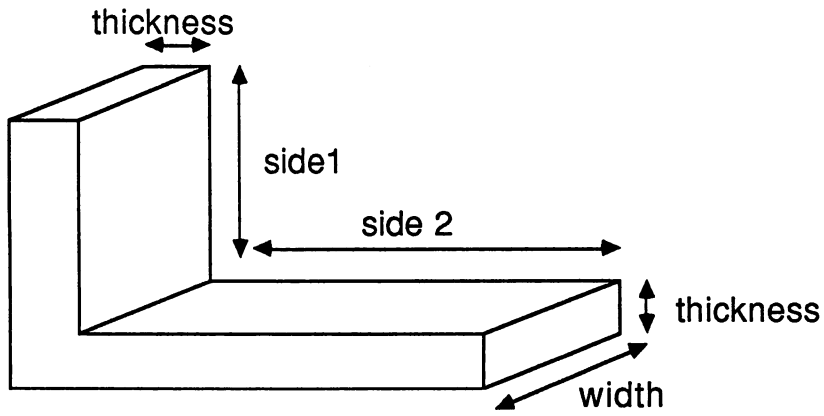


Figure 14-1. L-bracket Parameters

Step 2: Initialize the Primitive Class

The first time an object creation routine is called, it checks to see if the user-defined primitive has been initialized. If the user-defined primitive has not been initialized, it does so.

When a primitive class is initialized, the names of actual routines are associated with the Doré symbolic method names. The class is also given a name, and a primitive class identification number is set by Doré. All this is done with one call to *DeAddClass* <*DEAC*>. The class identification number is used by the method routines, the object creation routine, and also by the application program in some cases, for example with *DoExecSet* <*DOES*>.

The initialization routine will be very similar for all user-defined primitives. The initialization routine for the L-bracket example looks like this:

C code:

```
DtInt TypeLbracket;  /* primitive class identification number */

extern LbracketDestroy();
extern LbracketPrint();
extern LbracketComputeBoundVol();
extern LbracketUpdateAlternate();
extern LbracketRender();
extern LbracketPick();
extern LbracketGlbrndIniObjs();

static DtInt methods[] = {
     DcMethodDestroy, (DtInt)LbracketDestroy,
     DcMethodPrint, (DtInt)LbracketPrint,
     DcMethodCmpBndVolume, (DtInt)LbracketComputeBoundVol,
     DcMethodUpdStdAltObj, (DtInt)LbracketUpdateAlternate,
     DcMethodStdRenderDisplay, (DtInt)LbracketRender,
     DcMethodDynRender, (DtInt)LbracketRender,
     DcMethodGlbrndIniObjs, (DtInt)LbracketRender,
     DcMethodPick, (DtInt)LbracketPick,
};


LbracketInitialize()
{
     TypeLbracket = DeAddClass("DUoLbracket", 8, methods, DcNullPtr);
}
```

Fortran code:

```
     SUBROUTINE LBINI

     INCLUDE '/USR/INCLUDE/FORTRAN/DOREMETHODS'
     INCLUDE '/USR/INCLUDE/FORTRAN/DORE'
     INTEGER*4 MTHARR(12)
     EXTERNAL LBDST
     EXTERNAL LBPRT
     EXTERNAL LBCBV
     EXTERNAL LBRND
     EXTERNAL LBPCK
     EXTERNAL LBUSAO

     INTEGER*4 CLSID
     COMMON CLSID

     CALL DEAMTH(MTHARR, DCMDST, LBDST, 0)
     CALL DEAMTH(MTHARR, DCMPRT, LBPRT, 1)
     CALL DEAMTH(MTHARR, DCMCBV, LBCBV, 2)
     CALL DEAMTH(MTHARR, DCMSAO, LBUSAO, 3)
     CALL DEAMTH(MTHARR, DCMSRD, LBRND, 4)
     CALL DEAMTH(MTHARR, DCMDR,  LBRND, 5)
     CALL DEAMTH(MTHARR, DCMGIO, LBRND, 6)
     CALL DEAMTH(MTHARR, DCMPCK, LBPCK, 7)

     CLSID = DEAC('DUOLBR', 6, 8, MTHARR, DCNULL)
```

```
RETURN
END
```

The same routine is used for both *DcMethodDynRender* <*DCMDR*> and *DcMethodGlbrndIniObjs* <*DCMGIO*>. The last argument passed to *DeAddClass* <*DEAC*> is the default routine to be used for methods that are not explicitly specified in the list of methods. Here we have chosen a null routine as the default method. Note that the default method must be specified even though all the methods of interest are included in the list of methods, as in this example.

NOTE
Note that, in Fortran, you need an include statement within every routine. In C, the include statement needs to appear only once at the beginning of the file.

In Fortran, the method list is built using *DEAMTH*. The calling routine allocates space for the method list (twice as many elements as methods to be inserted). *DEAMTH* is called once for every method/routine pair to be added to the method list. *DEAMTH* takes as parameters the name of the array, the symbolic method name, the routine name associated with that method, and the number of elements that have already been added to the list.

Naming Conventions

Each Doré class (type) must have a *unique* string name. Each object creation routine name must also be unique. Adhering to one naming convention for new primitives helps to simplify matters. It is recommended that you use the prefix *DUo-* for user-defined object creation routines (standard Doré object creation routines begin with *Do*; just insert the "U" for User). It is also recommended that you use the same name for both the string name associated with the object and for the object creation routine name. In our example, we used *DUoLbracket* (in Fortran, *DUOLBR*) for both the string name and the object creation routine name.

Step 3: Creating an Object of the New Primitive Class

The following code creates an L-bracket object. The private data is set, and the alternate representation object, a simple polygon mesh, is created before *DeCreateObject* <*DECO*> is called.

In C, the routine *DeCreateObject* <*DECO*> is used to create an internal Doré object. You pass it the identification number for the primitive class and a pointer to the private data value for the object you are creating, and it passes back a handle to the object, which you pass back to the application program.

The calling sequence for creating an object is different in Fortran and C. In Fortran, the method routines of a user-defined primitive do not have direct access to an object's private data. Instead, the private data of an object is accessed from Fortran through calls to *DEROD*, *DEWOD*, and *DEDOD*. To create an object, use *DECO*, which takes as parameters the class identification number, a pointer to the private data values for the object (here, the array *lbrack*), and the number of bytes to allocate for copying the private data (here, 36 bytes). *DECO* allocates space for the private data of the object and copies 36 bytes from *lbrack* to the newly allocated space.

C code:

```
DtObject DUoLbracket(side1, side2, width, thickness)
DtReal side1;
DtReal side2;
DtReal width;
DtReal thickness;

{
     struct Lbracket *lbracket;
     DtObject lbracket_object;
     DtObject LbracketCreateAlternateGeom();
     static DtFlag initialize = DcTrue;

     if (initialize) {
          LbracketInitialize();
          initialize = DcFalse;
     }

     lbracket = (struct Lbracket *)malloc(sizeof *lbracket);

     lbracket->side1 = side1;
     lbracket->side2 = side2;
     lbracket->width = width;
     lbracket->thickness = thickness;

     lbracket->alternate_object =
       LbracketCreateAlternateGeom(lbracket);

     lbracket_object = DeCreateObject(TypeLbracket, lbracket);

     return(lbracket_object);
}



        DtObject LbracketCreateAlternateGeom(lbracket)
        struct Lbracket *lbracket;
        {
                DtReal vtx[14][3];
                DtInt contours[10];
```

```
        DtInt vlist[40];
        DtObject pmesh_object;

        /*
           Determine vertex locations from
     L-bracket dimension data
                .
                .
           Set up data structures needed to
     create a polygon mesh
                .
                .
        */

        pmesh_object = DoSimplePolygonMesh(DcRGB, DcLoc,
                       14, vtx, 10, contours, vlist,
                       DcConvex, DcFalse);

        return(pmesh_object);
}
```

Fortran code:

```
      INTEGER*4 FUNCTION DUOLBR(S1, S2, W, T)
      REAL*8 S1
      REAL*8 S2
      REAL*8 W
      REAL*8 T

      INCLUDE '/USR/INCLUDE/FORTRAN/DORE'
      INCLUDE 'LBRACKET_DATA'

      INTEGER*4 OBJECT
      EXTERNAL CRTALT
      INTEGER*4 CRTALT

      INTEGER*4 CLSID
      COMMON CLSID

      INTEGER INITIALIZE
      DATA INITIALIZE/DCTRUE/

      IF (INITIALIZE .EQ. DCTRUE) THEN
           CALL LBINI
           INITIALIZE = DCFALS
      ENDIF

      SIDE1 = S1
      SIDE2 = S2
      WIDTH = W
      THICK = T
      ALTOBJ = CRTALT(SIDE1, SIDE2, WIDTH, THICK)

      DUOLBR = DECO(CLSID, LBRACK, 36)
```

```
        RETURN
        END


        INTEGER*4 FUNCTION CRTALT(SIDE1, SIDE2, WIDTH, THICK)
        REAL*8 SIDE1
        REAL*8 SIDE2
        REAL*8 WIDTH
        REAL*8 THICK

        INCLUDE '/USR/INCLUDE/FORTRAN/DORE'

        REAL*8 VTX(42)
        INTEGER*4 CNTRS(10)
        INTEGER*4 VLIST (40)

C...... DETERMINE VERTEX LOCATIONS FROM L-BRACKET DIMENSION
C...... DATA.

C...... SET UP DATA STRUCTURES NEEDED TO CREATE A POLYGON
C...... MESH.

        CRTALT = DOSPM(DCRGB, DCL, VTX, 10, CNTRS, VLIST,
      1 DCCNVX, DCFALS)

        RETURN
        END
```

Step 4: Compiling an Alternate Representation

In this example, the alternate representation is compiled above in the creation routine (step 3). Compilation can, however, be delayed until the object is actually needed—during rendering, for example. A user-defined primitive may have two alternate representations, one for the dynamic renderer and one for the production renderer. In this case, compilation should occur *after* the renderer has been selected. (See "Efficiency Improvements," below, for recommendations on when compilation should occur.)

Step 5: Implement the Set of Methods for the New Primitive

The implementation details of each method routine depend on the nature of the primitive type. In most cases, the routines are short and simple.

All the methods take one argument (in C, a pointer to a *DtObjectStructure*; in Fortran, an INTEGER*4). The global variable *TypeLbracket*, which is used by some of the following L-bracket method routines, is the primitive class identification number that was set in the primitive class initialization routine. (See Step 2, above.)

DcMethodCmpBndVolume <DCMCBV>

Computes the bounding volume box for an object. If the object bounding volume is the same as the bounding volume of the alternate object, the work should be passed on to the alternate object by invoking *DeExecuteAlternate <DEEA>*. If the desired result is a volume that bounds the union of many alternate objects, each of those alternate objects should be executed in turn.

C code:

```
LbracketComputeBoundVol(object)
DtObjectStructure *object;
{
        struct Lbracket *lbracket;

        lbracket = (struct Lbracket *)(object->data);

        DeExecuteAlternate(lbracket->alternate_object);
}
```

Fortran code:

```
SUBROUTINE LBCBV(OBJECT)
INTEGER*4 OBJECT

INCLUDE 'LBRACKET_DATA'

CALL DEROD(OBJECT, LBRACK, 36)
CALL DEEA(ALTOBJ)

RETURN
END
```

In the L-bracket case, the bounding volume of the L-bracket is the same as the bounding volume of the simple polygon mesh used as the alternate object, so invoking *DeExecuteAlternate <DEEA>* is all that is done in C.

In Fortran, the routine *DEROD* is used to obtain access to the private data of the object (which includes the alternate object). *DEROD* takes as parameters the handle to the object, the name of the space to which the object data is to be copied (here, *lbrack*), and the number of bytes of private data to copy to that space (here, 36 bytes). Once the data has been copied to *lbrack*, *DEEA* can be called to execute the alternate object.

DcMethodDestroy <DCMDST>

Frees up the space used by the object and its private data. The Doré routine *DeDeleteObject <DEDO>* should be used to deallocate the space used by an alternate object. In C, use *free* to deallocate space that was dynamically allocated with *malloc* for any of the private data.

C code:

```
LbracketDestroy(object)
DtObjectStructure *object;
{
        struct Lbracket *lbracket;

        lbracket = (struct Lbracket *)(object->data);

        DeDeleteObject(lbracket->alternate_object);
        free(lbracket);
}
```

Fortran code:

```
      SUBROUTINE LBDST(OBJECT)
      INTEGER*4 OBJECT

      INCLUDE 'LBRACKET_DATA'

      CALL DEROD(OBJECT, LBRACK, 36)
      CALL DEDO(ALTOBJ)
      CALL DEDOD(OBJECT)

      RETURN
      END
```

In C, the L-bracket private data does not have any dynamically allocated fields, so all that needs to be done is delete the alternate object using *DeDeleteObject <DEDO>* and then free up the space used by the Lbracket structure.

In Fortran, *DEROD* is again used to obtain access to the private data of the object (including the alternate object). *DEDO* deletes the alternate object. Then *DEDOD* deallocates the space used for the object's private data.

DcMethodDynRender <DCMDR> and DcMethodGlbrndInlObjs <DCMGIO>

Renders the object for the dynamic renderer and the production renderer, respectively. In both cases, nothing needs to be done if the primitive type is not executable, or if invisibility is enabled. This is determined by calling *DeInqRenderable <DEQR>*. Otherwise the rendering work is passed on to the alternate object using *DeExecuteAlternate <DEEA>*. As in the examples above, Fortran requires the additional step of calling *DEROD* to obtain access to the object's private data.

C code:

```
LbracketRender(object)
DtObjectStructure *object;
{
        struct Lbracket *lbracket;

        if (!DeInqRenderable(TypeLbracket)){
                return;
        }

        lbracket = (struct Lbracket *)(object->data);

        DeExecuteAlternate (lbracket->alternate_object);

}
```

Fortran code:

```
      SUBROUTINE LBRND(OBJECT)
      INTEGER*4 OBJECT

      INCLUDE '/USR/INCLUDE/FORTRAN/DORE'
      INCLUDE 'LBRACKET_DATA'

      INTEGER*4 CLSID
      COMMON CLSID

      IF (DEQR(CLSID)) THEN
           CALL DEROD(OBJECT, LBRACK, 36)
           CALL DEEA(ALTOBJ)
      ENDIF

      RETURN
      END
```

The L-bracket example uses the same alternate representation for both the dynamic and production renderers, so the same routine (*LbracketRender*; in Fortran, *LBRND*) is used for both methods.

DcMethodPick <DCMPCK>

Determines if the object has been picked. If picking is not enabled for this primitive type, or if the primitive type is not executable, nothing needs to be done. Otherwise picking must be initialized for this particular object by invoking *DeInitializeObjPick <DEIOP>*. If there are multiple alternate objects, *DeInitializeObjPick <DEIOP>* need only be invoked once. Then the work is passed on to the alternate object by invoking *DeExecuteAlternate <DEEA>*. Fortran requires the additional step of calling *DEROD* to obtain access to an object's private data.

C code:

```
LbracketPick(object)
DtObjectStructure *object;
{
        struct Lbracket *lbracket;

        if (!DeInqPickable(TypeLbracket)){
                return;
        }

        lbracket = (struct Lbracket *)(object->data);

        DeInitializeObjPick(object);
        DeExecuteAlternate (lbracket->alternate_object);

}
```

Fortran code:

```
      SUBROUTINE LBPCK(OBJECT)
      INTEGER*4 OBJECT

      INCLUDE '/USR/INCLUDE/FORTRAN/DORE'
      INCLUDE 'LBRACKET_DATA'

      INTEGER*4 CLSID
      COMMON CLSID

      IF (DEQP(CLSID)) THEN
         CALL DEIOP(OBJECT)
         CALL DEROD(OBJECT, LBRACK, 36)
         CALL DEEA(ALTOBJ)
      ENDIF

      RETURN
      END
```

DcMethodPrint <DCMPRT>

Prints object information.

C code:

```
LbracketPrint(object)
DtObjectStructure *object;
{
        struct Lbracket *lbracket;

        lbracket = (struct Lbracket *)(object->data);

        if (lbracket == (struct Lbracket *)0 ) {
               printf ("L-bracket is NULL\n");
               return;
        }
        printf ("L-bracket data:");
        printf (" side1 %f side2 %f width %f  thickness %f\n",
           lbracket->side1, lbracket->side2,
           lbracket->width, lbracket->thickness);
}
```

Fortran code:

```
  SUBROUTINE LBPRT(OBJECT)
  INTEGER*4 OBJECT

  INCLUDE 'LBRACKET_DATA'

  CALL DEROD(OBJECT, LBRACK, 36)

  PRINT *, 'LBRACKET DATA:'
  PRINT *, 'SIDE1 ', SIDE1
  PRINT *, 'SIDE2 ', SIDE2
  PRINT *, 'WIDTH ', WIDTH
  PRINT *, 'THICKNESS ', THICK

  RETURN
  END
```

DcMethodUpdStdAltObj <DCMSAO>

C code:

```
DtObject
LbracketUpdateAlternate(object)
DtObjectStructure *object;
{
     struct Lbracket *lbracket;
     DtObject LbracketCreateAlternateGeom();

     lbracket = (struct Lbracket *)(object->data);
```

```
    if (lbracket->alternate_object == DcNullObject)
        return (LbracketCreateAlternateGeom(lbracket));
    else
        return (lbracket->alternate_object);
}
```

Fortran code:

```
    INTEGER*4 FUNCTION LBUSAO(OBJECT)
    INTEGER*4 OBJECT

    INCLUDE '/USR/INCLUDE/FORTRAN/DORE'
    INCLUDE 'LBRACKET_DATA'

    CALL DEROD(OBJECT, LBRACK, 36)
    IF (ALTOBJ .EQ. DCNULL) THEN
        ALTOBJ = CRTALT(SIDE1, SIDE2, WIDTH, THICK)
        CALL DEWOD(OBJECT, LBRACK, 36)
    ENDIF
    LBUSAO = ALTOBJ

    RETURN
    END
```

DcMethodStdRenderDisplay <DCMSRD>

For the L-bracket example, *DcMethodStdRenderDisplay <DCMSRD>* uses the same method as the other two render methods. (See "DcMethodDynRender and DcMethodGlbrndIniObjs," above.)

Step 6: Install the New Primitive Type

When all the code described above has been written, it must be compiled and made available to application users. Note that Doré source code is not needed. If the code is in more than one file, or if you want to combine several user-defined primitives, use the loader or the library utility to combine the objects into one object module or library. Application programs will link to the object module for the new primitives in addition to the Doré object module or library.

```
% cc -c myapplication.c
% cc myapplication.o userprimitives.o /usr/lib/dore.o
    -o myapplication
```

Efficiency Improvements

For maximum efficiency, follow these two recommendations:

(1) Use base primitives as alternate objects when possible.

(2) Compile the alternate representation only when it is actually needed, for example, in the method routines rather than in the object creation routines.

Recommendation 1

It is more efficient to use base primitives as alternate objects than it is to use primitives that will in turn compile alternate objects. In the L-bracket example for maximum efficiency we could have created a triangle list instead of using a simple polygon mesh. All the code used to implement the L-bracket would be the same except for the *LbracketCreateAlternateGeom* routine (in Fortran, *CRTALT*.) Using the primitive would be the same.

Recommendation 2

In the implementation of the standard Doré primitives, the alternate geometry object is not compiled unless it is actually needed. For example, when a torus object is created with *DoTorus(bigradius, smallradius)* the private data of the torus object is initialized, but the decomposition into triangles only happens the first time the object is rendered.

We could have done the same thing in the L-bracket example. In that case, we would not compile the alternate object in the object creation routine. Instead, we would have the method routines check to see if the alternate object exists before executing it, and call the alternate object compiler first if it does not exist. For example, the *LbracketComputeBoundVol* (in Fortran, *LBCBV*) routine would then look like this:

```
C code:

LbracketComputeBoundVol(object)
DtObjectStructure *object;
{
        struct Lbracket *lbracket;

        lbracket = (struct Lbracket *)(object->data);

        LbracketUpdateAlternate(object); /* see DcMethodUpdStdAltObj */
        if (lbracket->alternate_object != DcNullObject) {
```

```
                DeExecuteAlternate(lbracket->alternate_object);
        }
}
```

Fortran code:

```
        SUBROUTINE LBCBV(OBJECT)
        INTEGER*4 OBJECT

        INCLUDE '/USR/INCLUDE/FORTRAN/DORE'

        EXTERNAL LBUSAO
        INTEGER*4 LBUSAO
        INTEGER*4 ALT

        ALT = LBUSAO(OBJECT)   ! SEE DCMETHODUPDSTDALTOBJ

        IF (ALT .NE. DCNULL) THEN
                CALL DEEA(ALT)
        ENDIF

        RETURN
        END
```

Using a New Primitive

The application program will use *DUoLbracket <DUOLBR>* to create L-bracket objects, just as it would use the object creation routine for a standard Doré primitive.

C code:

```
        #include "dore.h"


        main()
        {

                .
                .
              DsInitializeSystem(0);
                .
                .
              DgAddObj(DoPrimSurf(DcSphere));
                .
                .
              DgAddObj(DUoLbracket(5., 10., 5., 1.5));
                .
                .
        }
```

Fortran code:

```
PROGRAM MAIN
      .
      .
CALL DSINIT(0)
      .
      .
CALL DGAO(DOPMS(DCSPH))
      .
      .
CALL DGAO(DUOLBR(5.0D0, 10.0D0, 5.0D0, 1.50D0))
      .
      .
STOP
END
```

Conventions for Implementing User-Defined Primitives

The following directory structure is recommended as a standard for implementing user-defined primitives. Figure 14-2 illustrates this directory structure. Online examples and templates for these files are available; see the current Release Notes for the exact location of these files.

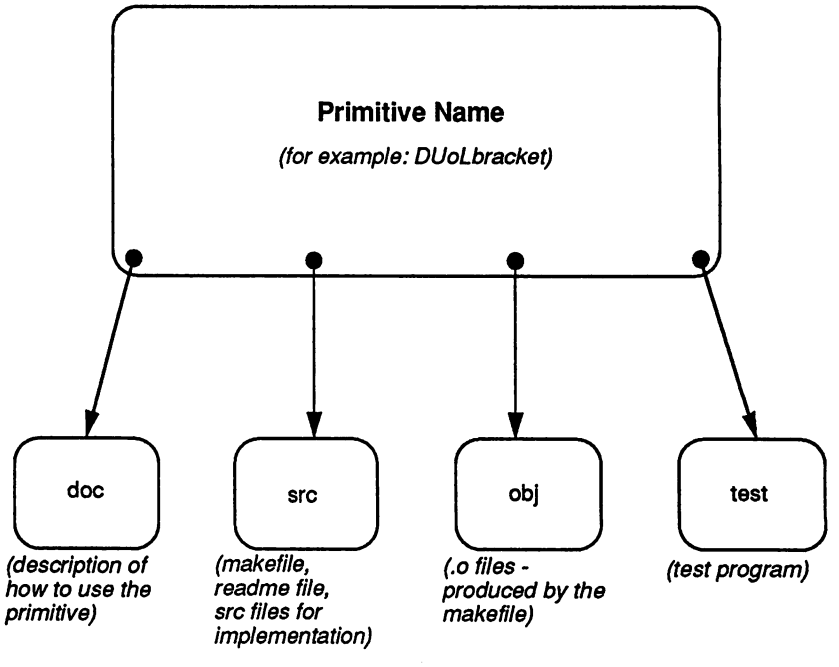


Figure 14-2. Recommended Directory Structure

For simplicity, use the same name for both the user-defined primitive object creation routine and for the top-level directory (for example, DUoLbracket or DUOLBR). This top-level directory includes the *doc*, *src*, *obj*, and *test* subdirectories, described below.

doc Subdirectory

The *doc* subdirectory includes a description of how to use the new primitive, in "man" page style. The online template file contains a general description of "man" page style, as well as a checklist of things to remember to include in the function description.

src Subdirectory

The *src* subdirectory includes

- a Readme file explaining how to build the primitive
- a makefile
- source files for the implementation of the user-defined primitive

obj Subdirectory

The *obj* subdirectory, which will include

- the .o files produced by the makefile
- either a .a library built with *ar* or a combined .o file that users of the primitive can link to

test Subdirectory

This subdirectory contains all the necessary files for testing the new primitive. It includes

- a Readme file describing how to build and run the test program
- a makefile
- source files

This subdirectory will also include the object files and executable files produced by the makefile.

Chapter Summary

User-defined primitives can be added to the set of standard Doré primitives when the shape or behavior of an object is not the same as that of any existing Doré primitive or group of primitives.

Each Doré renderer can directly render only a subset of the standard Doré primitives. This subset of directly drawable primitives for a given renderer is referred to as the *base primitives* for that renderer. All other primitives, including user-defined primitives, compile *alternate representations*, which are ultimately compiled into and rendered as base primitives.

The alternate representation for a user-defined primitive can be a base primitive, another alternate representation, or a group of objects. In addition, an object can have more than one alternate representation. In this case, the render method for the object would check specified conditions to determine which alternate representation should be displayed.

All Doré primitives consist of private data, a class identifier, an initialization routine, an object creation routine, a routine to create the alternate representation, and a set of methods. This chapter includes C and Fortran examples for each element of a primitive.

Each Doré class must have a unique string name. The object creation routine name must also be unique. The recommended naming convention is to use the prefix *DUo-* for the user-defined object creation routine. Use this same name for the class string name as well.

For maximum efficiency, base primitives should be used as alternate representations when possible. Another way to increase efficiency is to compile the alternate representation only when it is actually needed— for example, in the method routines rather than in the object creation routines.