3D Metafile Reference

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

ISBN 0-201-62202-5
1 2 3 4 5 6 7 8 9-MA-9998979695
First Printing, Month 1995

The paper used in this book meets the EPA standards for recycled fiber.

Library of Congress Cataloging-in-Publication Data

3D Metafile Reference / [Apple Computer, Inc.].
p. cm.
Includes index.
ISBN 0-201-62202-5
1. Macintosh (Computer)—Programming. 2.
I. Apple Computer, Inc.
nnnn.n.nnnnnnn 1995
nnn.nnn—nnnn 95-nnnnn
CIP

Figures, Tables, and Listings

CHAPTER 1

3D Metafile Reference

Contents

This document describes the 3D Metafile, a file format designed to permit the storage and interchange of 3D data.

▲ **WARNING**
This information in this document is preliminary and is subject to change. ▲

Introduction

The 3D Metafile is a file format for 3D graphics applications that make use of the QuickDraw 3D graphics library or other object-based 3D graphics libraries. This document describes the 3D Metafile file format.

The purposes of the metafile are

- to establish a standard file format for 3D graphics files
- to establish canonical forms for descriptions of familiar 3D graphics objects

This standard is put forward to promote compatibility among 3D graphics applications and is meant to facilitate the transfer and exchange of data between distinct applications. The file format also permits a project to be saved to a file in such a way that it may be resumed or altered at a later time.

The metafile file format permits objects to be labeled and referenced. A metafile may also include one or more tables of contents in which such labels and references are listed. A table of contents may provide a complete catalog of the items contained in a metafile and of all cross-references among those items. However, a metafile is not itself a database and does not have the capabilities of a database. Applications that wish to apply the capabilities of a database to the contents of a metafile must connect that file to a preexisting database program.

The canonical forms for descriptions of 3D graphics objects outlined in this document embody an object- and class-based approach to 3D graphics and reflect the structure of the QuickDraw 3D class hierarchy. This approach can be described briefly as follows. First, a number of basic data types are introduced. Next, more complex data types, called *objects*, are defined in terms of these basic data types. Finally, similar objects are grouped together to form classes of objects, arranged in a hierarchical structure.

Each class of objects, and thus each object, is correlated with a particular node in that structure. We use the terms *parent* and *child* to describe the relationships among objects located at immediately adjacent and connected nodes in the structure. For example, a color attribute may be included in a set of attributes that is assigned to a geometric object. In that case, the geometric object is a parent of the attribute set, which in turn is a parent of the color attribute, while the color attribute is a child of the attribute set, which in turn is a child of the geometric object. See the book *3D Graphics Programming With QuickDraw 3D* for complete details on this approach to the classification of 3D graphics objects.

The metafile file format includes two mechanisms that allow two or more objects to be grouped together to form a more complex object having as much hierarchical structure as desired. These mechanisms are the **container** and the **group,** which are described in the sections "Containers" on page 1-44 and "Groups" on page 1-194. The format also includes two special objects, file pointers and reference objects, that can be used to instantiate previously specified objects by reference. These objects are described in the sections "File Pointers" on page 1-24 and "Reference Objects" on page 1-37.

This document defines a format for ASCII text files and also defines a format for binary files. The two formats incorporate the same functional features, and there is a one-to-one correspondence between their components. Most objects are represented very similarly in the two formats. However, some objects, such as file pointers, are represented differently in the two file formats, as described below. Each text file has at least one binary file counterpart, and each binary file has at least one text file counterpart, but in general that counterpart is not unique.

Metafile File Structure

A metafile is simply a sequence or list of one or more valid metafile objects. Each metafile must contain exactly one 3D metafile header, and this header must be the first object to occur in the file. Objects following the header may occur in any order permitted by the metafile class hierarchy. Currently, every object that begins in a metafile must be wholly contained in that file; thus, it is not legal to truncate the description of an object at the end of a file.

A metafile may include one or more tables of contents, but need not include any. Should a metafile include more than one table of contents, each table of contents should continue the record provided by the immediately previous

table of contents (if such exists) without duplication. A table of contents may contain information about objects occurring before or after it, or both, but should not contain information about any object that either precedes an object mentioned in a previous table of contents or follows an object mentioned in a subsequent table of contents. Conventions for listing objects in tables of contents are described in the section "Tables of Contents."

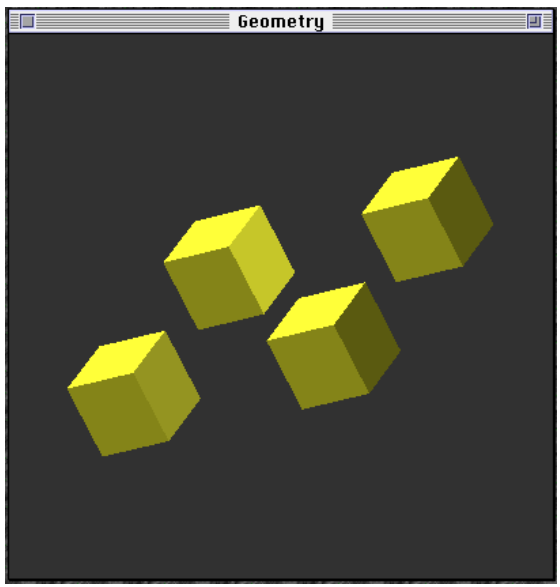
There are three principal types of metafile: stream, normal, and database. The type of a metafile is indicated by a flag in the metafile header. See the section "3D Metafile Header" for complete details on these flags.

In a **stream** file, objects can not be instantiated by reference; the complete specification of an object must occur at each place in the file at which that object is to be instantiated. Objects can be instantiated by reference in normal and database files. In a **normal** metafile, the complete specification of an object can occur once only; an object can be instantiated multiply only by reference. No such restrictions apply to database files. In a **database** file, every object that can be instantiated by reference and is not itself a reference object must be listed in a table of contents (whether or not that object has been instantiated by reference).

A stream file can but need not include a table of contents. A database file must include a table of contents. A normal file in which objects are instantiated by reference must include a table of contents; a normal file in which no objects are instantiated by reference can but need not include a table of contents.

To illustrate the differences among the three types of metafile, we show how a single model (Figure 1-1) is described in a text file of each type. The model consists of four occurrences (at different locations) of a colored box.

Figure 1-1 Four instantiations of a box



The following is a complete specification of the colored box shown in Figure 1-1.

```
Container (
    Box ( 0 1 0 0 0 1 1 0 0 0 0 0 )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0.9 0.9 0.2 )
    )
)
```

The expression `Container (...)` is used subsequently to abbreviate this specification. Transforms are used to place the box in various positions (each transform applies to the object specified or referenced immediately subsequent to it).

In a stream file, the specification of the box must occur four times, as shown in Listing 1-1.

Listing 1-1 A stream metafile

```
3DMF ( 1 0 Stream Label0> )   # header

Container ( ... )             # first instantiation of box

Translate ( 3 0 0 )           # transform
Container ( ... )             # second instantiation

Translate ( 0 3 0 )           # transform
Container ( ... )             # third instantiation

Translate ( -3 0 0 )          # transform
Container ( ... )             # fourth instantiation
```

Such repetition can make stream files lengthy. However, a stream file can be read by a parser having only sequential access to that file.

In a normal file, the box is completely specified once and is instantiated by reference three times. The file pointers and reference objects used to effect instantiations by reference are listed together in the table of contents. Other referenceable objects (such as the transforms) that are instantiated once only are not listed in the table of contents.

The normal metafile permits the most compact representation of the model.

Listing 1-2 A normal metafile

```
3DMetafile ( 1 0 Normal Label0> )

Label1: Container ( ... )        # first instantiation

Translate ( 3 0 0 )
Reference ( 1 )                  # second instantiation

Translate ( 0 3 0 )
Reference ( 1 )                  # third instantiation

Translate ( -3 0 0 )
```

```
Reference ( 1 )                 # fourth instantiation

Label0:   TableOfContents (
             2 -1 0 12
             1
             1 Label1>
          )
```

Note
`Label1>` is a file pointer correlated with the label `Label1` that precedes the specification of the box. `Reference ( 1 )` is a reference object correlated with `Label1>` (and thus with the specification of the box) in the table of contents. See the section "File Pointers" on page 1-24 for an explanation of how instantiation by reference is accomplished through the use of these objects. ◆

In a database file, the box is also instantiated by reference, and the file pointers and reference objects used to instantiate it are listed in the table of contents. And, (with the exception of reference objects themselves) all other referenceable objects (the attribute set which contains the box's color attributes, and the transforms) are referenced, and all of these references are listed in the table of contents.

The contents of a database file can be discovered quickly by inspection of its tables of contents.

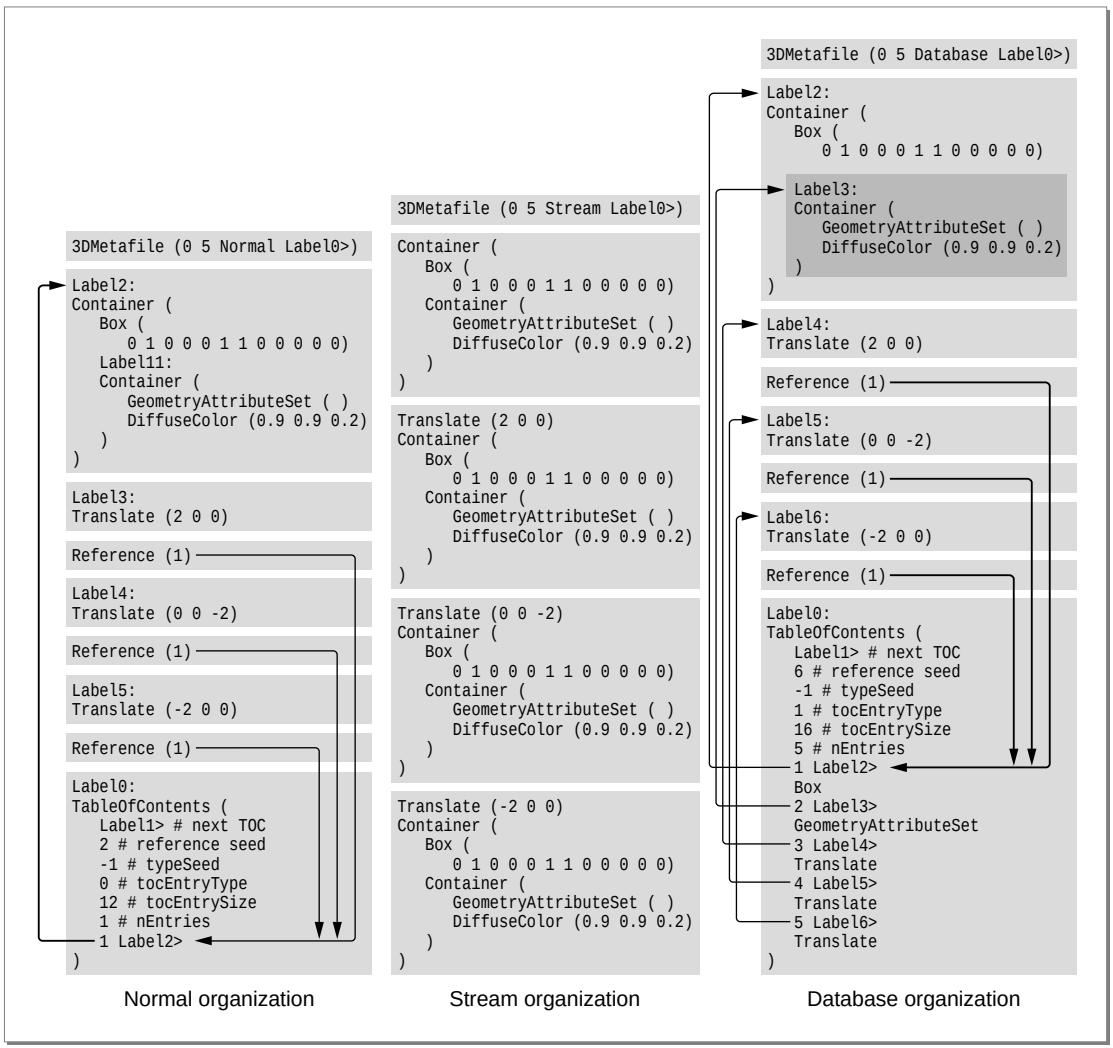
Listing 1-3 A database metafile

```
3DMetafile ( 1 0 Database Label0> )

Label1:
Container (
   Box ( 0 1 0 0 0 1 1 0 0 0 0 0 )
   Label2:
   Container (
      AttributeSet ( )
      DiffuseColor ( 0.9 0.9 0.2 )
```

```
    )
)

Label3:
Translate ( 3 0 0 )
Reference ( 1 )

Label4:
Translate ( 0 3 0 )
Reference ( 1 )

Label5:
Translate ( -3 0 0 )
Reference ( 1 )

Label0:
TableOfContents (
    2 -1 0 12
    5
    1 Label1>
    2 Label2>
    3 Label3>
    4 Label4>
    5 Label5>
)
```

Figure 1-2 shows, side by side, the three principal forms of a metafile.

Figure 1-2 Types of metafiles



Basic Data Types

All metafile object specifications, including specifications of custom objects you define yourself, must use only the following basic data types. Additional basic data types may be introduced in the future if the need for them arises.

Unsigned Integer Data Types

`Uns8`	An unsigned 8-bit integer.
`Uns16`	An unsigned 16-bit integer.
`Uns32`	An unsigned 32-bit integer.
`Uns64`	An unsigned 64-bit integer.

Signed Integer Data Types

`Int8`	A signed 8-bit integer.
`Int16`	A signed 16-bit integer.
`Int32`	A signed 32-bit integer.
`Int64`	A signed 64-bit integer.

Floating-Point Integer Data Types

`Float32`	A single-precision 32-bit floating-point number.
`Float64`	A double-precision 64-bit floating-point number.

Note
Floating point numbers must be represented in the manner specified by the IEEE floating-point standard (IEEE 754). See the book *Inside Macintosh: PowerPC Numerics* for information on the IEEE floating-point standard. ◆

Basic 3D Data Types

The following 3D data types are defined using the basic data types described in the previous section.

Two-Dimensional Points

```
typedef struct Point2D {
   Float32                  x;
   Float32                  y;
} Point2D;

typedef struct DPoint2D {
   Float64                  x;
   Float64                  y;
} DPoint2D;
```

Three-Dimensional Points

```
typedef struct Point3D {
   Float32                  x;
   Float32                  y;
   Float32                  z;
} Point3D;

typedef struct DPoint3D {
   Float64                  x;
   Float64                  y;
   Float64                  z;
} DPoint3D;
```

Three-Dimensional Rational Points

```
typedef struct RationalPoint3D {
   Float32                  x;
   Float32                  y;
   Float32                  w;
} RationalPoint3D;

typedef struct DRationalPoint3D {
   Float64                  x;
   Float64                  y;
   Float64                  w;
} DRationalPoint3D;
```

Four-Dimensional Rational Points

```
typedef struct RationalPoint4D {
   Float32                  x;
   Float32                  y;
   Float32                  z;
   Float32                  w;
} RationalPoint4D;

typedef struct DRationalPoint4D {
   Float64                  x;
   Float64                  y;
   Float64                  z;
   Float64                  w;
} DRationalPoint4D;
```

Note
Three- and four-dimensional points are used to represent two- and three-dimensional points respectively in homogeneous coordinate systems. ◆

Color Data Types

```
typedef struct RGBColor {
   Float32                  red;
   Float32                  green;
   Float32                  blue;
} RGBColor;
```

IMPORTANT
The values in the fields of an RGB color data type must lie in the closed interval [0, 1]. 0 is the minimum value; 1 is the maximum value. ▲

The 3D metafile currently supports only the RGB (red, green blue) color model.

Two-Dimensional Vectors

```
typedef struct Vector2D {
   Float32                  x;
   Float32                  y;
} Vector2D;

typedef struct DVector2D {
   Float64                  x;
   Float64                  y;
} DVector2D;
```

Three-Dimensional Vectors

```
typedef struct Vector3D {
   Float32                  x;
   Float32                  y;
   Float32                  z;
} Vector3D;
```

```
typedef struct DVector3D {
   Float64                    x;
   Float64                    y;
   Float64                    z;
} DVector3D;
```

Parameterizations

```
typedef struct Param2D {
   Float32      u;
   Float32      v;
} Param2D;

typedef struct Param3D {
   Float32      u;
   Float32      v;
   Float32      w;
} Param3D;

typedef struct DParam2D {
   Float64      u;
   Float64      v;
} DParam2D;

typedef struct DParam3D {
   Float64      u;
   Float64      v;
   Float64      w;
} DParam3D;
```

Tangents

```
typedef struct Tangent2D {
   Vector3D                  uTangent;
   Vector3D                  vTangent;
} Tangent2D;

typedef struct Tangent3D {
   Vector3D                  uTangent;
   Vector3D                  vTangent;
   Vector3D                  wTangent;
} Tangent3D;

typedef struct DTangent2D {
   DVector3D                 uTangent;
   DVector3D                 vTangent;
} DTangent2D;

typedef struct DTangent3D {
   DVector3D                 uTangent;
   DVector3D                 vTangent;
   DVector3D                 wTangent;
} DTangent3D;
```

Matrices

```
typedef Float32 Matrix3x3        [3][3];

typedef Float32 Matrix4x4        [4][4];

typedef Float64 DMatrix3x3       [3][3];

typedef Float64 DMatrix4x4       [4][4];
```

Abstract Data Types

The 3D Metafile file format defines the following seven abstract data types: object type, size, bitfield, enumerated type, file pointer, string, and raw data. File pointers are discussed in another section. The other six abstract data types are discussed in this section.

Object Type

Every metafile object has a type. In a text file, an object type is expressed by a character string, such as `Polygon`. In a binary file, an object type is expressed by a 4-byte code, such as `plyg`. In both text and binary files, every object specification begins with an object type.

The metafile file format allows you to introduce new types of custom objects. Whenever you do so, you must first declare and register a new type tag for custom objects of that type. The manner in which new type tags may be declared and registered is explained in the section "Types" on page 1-42.

Size

All metafile objects are padded to 4-byte boundaries; thus, the size of an object is always a multiple of 4.

In a text file, object specifications are delimited by parentheses, as shown in the following example:

```
Polygon (                    # object type
   3                         # number of vertices
   0 0 0                     # first vertex
   1 0 0                     # second vertex
   0 1 0                     # third vertex
)
```

The type `Polygon` and the parentheses are ignored when the size of the polygon is computed. Only the sizes of the values in the fields of the structure

are taken into account. This polygon is a structure having two fields. The value in the first field is an unsigned 32-bit integer, and the value in the second field is an array of three three-dimensional points (the array tag and encapsulating brackets are omitted in the metafile specification). The size of an unsigned 32-bit integer is 4 bytes, and the size of a three-dimensional point is 12 bytes, so the size of the above polygon is 40 bytes.

In a binary file, an object specification must always include a line that indicates the size of that object. The above polygon would be specified in a binary file as follows:

```
00: 706C6967    plyg          # object type
04: 00000028    40            # object size
08: 00000003    3             # number of vertices
0A: 00000000    0.0           # x coordinate of first vertex
10: 00000000    0.0           # y coordinate of first vertex
14: 00000000    0.0           # z coordinate of first vertex
18: 3F800000    1.0           # x coordinate of second vertex
.
.
.
```

Indications of object type and object size do not contribute to the size of an object. Thus, the size of the above polygon is 40 bytes, not 44 or 48.

An object may be of size 0. In a text file, an object of size 0 is described by a tag followed by a pair of empty parentheses. For example, `AttributeSet ( )` specifies an object of size 0. Some objects have a defined default specification. If such an object is represented as being of size 0, it is understood that the default specification is intended.

IMPORTANT

With the exception of geometric objects, default object specifications are not written to a file. ▲

Bitfields and Enumerated Types

Bitfields and enumerated types establish associations between unsigned integers and ASCII text strings. The names of bitfields and enumerated types may include either text characters or digits, but do not include blank spaces or

punctuation marks. In binary files, bitfields and enumerated types are represented by unsigned 32-bit integers such as (in hexadecimal) `0x00000001`, `0x0000000E`. The strophe symbol (`|`) is used to catenate bits both in text and binary files.

Strings

In a text file, a string is a sequence of ASCII text symbols enclosed in double quotation marks.

Only the following escape sequences may occur in a text file string:

`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\'`	single quotation mark
`\"`	double quotation mark

In a binary file, a string is represented by a string of zero-terminated padded characters. The size of a string in a binary file is determined as follows:

```
len = strlen(string);
pad = (len + 1) % 4
size = len + ((pad > 0) ? 4 - pad : 0)
```

Raw Data

Raw data are used to store information that is platform dependent or is inherently nonalphanumeric.

In a text file, raw data are stored as hexadecimal strings prefixed by the characters `'0x'`. Strings of raw data are not padded in text files. However, your application may pad them if you wish.

In a binary file, raw data are stored as sequences of bytes, padded to a 4-byte boundary. The size of raw data is computed as follows:

```
pad = rawDataSize % 4
size = rawDataSize + (pad > 0 ? 4 - pad : 0)
```

File Pointers

DESCRIPTION

A metafile file pointer indicates the location of another object in that metafile, to which it points. A file pointer and the object to which it points (called the **target object**) must occur in the same file. A target object may occur before or after an associated file pointer in a metafile. A file pointer may fail to have a target object; such a file pointer is *null*. File pointers may occur both in ASCII text metafiles and in binary metafiles. A file pointer is neither declared nor initialized; it is identified as such by the positions in which it may appear and (in a text file) by the type of expression used to represent it.

In a binary metafile, a file pointer is represented by an unsigned 64-bit integer that reports the address or location of its target object in the metafile. A generator of a binary metafile must determine the number of bytes by which a target object is offset from the end of the header in order to write the correct value of a pointer to that object and must update that file pointer whenever any new objects are inserted between the positions occupied by the pointer and its target object in the metafile.

Note
A file pointer is offset relative to the end of the header of the file in which it occurs, not relative to the beginning of that file. ◆

In an ASCII text metafile, a file pointer is represented by a character string composed of at least two characters, the last of which is a right angle bracket (>). Thus `p>` and `Arrow>` are file pointers; `p`, `>`, and `Arrow` are not. In a text file, the target object of a file pointer must bear a label corresponding to that file pointer. The label corresponding to a file pointer is the result of omitting the final right angle bracket from the string representing that file pointer. For example, the label corresponding to `string>` is `string`. Such a label is always followed immediately by a colon, then by the target object: `string: targetobject`. Each file pointer may correspond to at most one label, and each label may correspond to at most one file pointer. A metafile should not contain

a label that does not correspond to a file pointer, but the presence of such a label does not invalidate a metafile.

Two types of file pointers may occur in a metafile, corresponding to two types of target object a file pointer may have. The target object of a file pointer of the first type is a table of contents; such a file pointer is meant to indicate the location of a table of contents and serves no other purpose. (A file pointer of this type must occur in the fourth field of each header. A file pointer of this type must also occur in the first field of each table of contents; this pointer points to the location of the next subsequent table of contents, if one exists. A file pointer of this type may occur in no other position.)

The target object of a file pointer of the second type must be either an object of type shared or a container the root object (that is, the first object) of which is of type shared. The root object of a container may not be the target object of a file pointer. The purpose of a file pointer of this type is to enable the metafile writer to make repeated reference to a target object without repeating that object's definition. (A file pointer of this type may occur only in the second field of a table of contents entry; thus, a metafile that contains file pointers of this type must include at least one table of contents.) The way in which repeated reference to an object is accomplished through the use of file pointers of this type is explained in the next paragraph.

An application may also permit a user to make reference in one context to an object specified or created in another context, in order to facilitate the construction of complex objects from component objects and to permit the user to place the same object in several scenes or at several positions in the same scene without having to specify or create that object several times. In a metafile, reference to objects defined elsewhere is accomplished in a manner involving several components: a file pointer, a target object, an integer, an entry in a table of contents, and a special metafile object called a *reference object*. (In a text file, the label corresponding to a file pointer must also be present.) The object to be referenced at some other position must be the target object of a file pointer. That file pointer must appear together with an appropriately chosen integer in an entry in a table of contents located in the file containing the target object. (If that file contains no table of contents, then a table of contents must be created.) The integer thus associated with that file pointer must be entered in the field of a reference object, one occurrence of which must be placed at each position at which the target object is to be referenced.

The target object, file pointer, and table of contents must all occur in the same file. The reference object associated with a target object may occur in the same file or in another file. A reference in a file to a target object located in the same

file is termed *internal*; a reference in a file to a target object located in a different file is termed *external*. In the latter case, the reference object must also have a child object that indicates the location of the home file of the target object. A lengthy specification may be modularized and spread across several files through the use of external references. See the sections "Tables of Contents" on page 1-32 and "Reference Objects" on page 1-37 for further information about these objects.

There may be at most one file pointer to any target object; thus, once an integer has been associated with a pointer to a target object in a table of contents entry, that integer is the only integer that may be used (in any file) to reference that target object. (A file pointer of this type that is not associated with a reference object is legal, but serves no purpose.)

Clearly, a metafile reader must be programmed to recognize and to respond appropriately to reference objects, tables of contents, and file pointers and not to confuse them with other types of objects. As noted, a metafile may contain file pointers and reference objects that are idle. A metafile reader cannot determine whether a file pointer or reference object is idle by inspection of that object; thus, an efficient reader should search out these special metafile objects prior to reading the body of the metafile.

EXAMPLES

Here are some examples of legal uses of file pointers in an ASCII text metafile:

```
3DMetafile (                    # header
   1 0
   Normal
   toc>                         # pointer to table of contents
)
.
.
.
Linus: Line ( 0 0 0  1 0 0)     # label and target object
.
.
.
Translate ( 0 1 0 )
Reference ( 1 )                 # reference object
.
```

```
.
.
toc: Table of Contents (      # label and target object
      nextTOC>                # pointer to next table of contents
                                    (may be idle)
...
      1 Linus>            # table of contents entry, including
                          file pointer and integer occurring in
                          related reference object
.
.
.
)
```

The file pointer `Linus>` is used to place its target object within the scope of a translation; thus, it adds to the model the image of the original line under a translation.

```
arrow:
BeginGroup ( DisplayGroup ( ) )
   arrowTip:
   BeginGroup ( DisplayGroup ( ) )
      Translate ( ... )
      Scale ( ... )
      Container (
         cone ( ... )
         arrowColor:
         Container (
            AttributeSet ( )
            DiffuseColor ( ... )
         )
      )
   EndGroup ( )
   arrowShaft:
   BeginGroup ( DisplayGroup ( ) )
      Scale ( ... )
      Container (
         Cylinder ( ... )
         Reference ( 1 )
```

```
        )
    EndGroup ( )
EndGroup ( )
. . .
toc:
TableOfContents (
    ...
    1  arrowColor>
    2  arrowTip>
    3  arrowShaft>
    4  arrow>
)
```

The complex object specified in this example is an arrow-shaped object having a cylindrical shaft and a conical tip, both of which are the same color. The components of the arrow are labeled and associated with file pointers so that they may be referenced elsewhere if desired. The object labeled `arrowColor` occurs within the definition of the object labeled `arrowTip`. The reference object `Reference ( 1 )` is used to include that object in the specification of the arrow's shaft, in order to assign to the arrow's shaft the color already assigned to its tip. See the sections "Display Groups" on page 1-194 and "Containers" on page 1-44 for explanations of how these objects may be used to form complex structured objects.

Additional Type Definitions

Boolean Enumerated Types

Text	Binary
`False`	0x00000000
`True`	0x00000001

Variable-Sized Integer Types

We define two variable-sized integer types `Uns` and `Int`. These integer types are used primarily to pack arrays of indices. Objects must be padded to the next long word whenever these integer types are used.

Use of `Uns` in packing should accord with the following conventions. If the maximum index value is less than 256, use `Uns8`. If the maximum index value is greater than or equal to 256 and less than or equal to 65,536, use `Uns16`. If the maximum index value is greater than or equal to 65,536, use `Uns32`.

Use of `Int` in packing should accord with the following conventions. If the maximum index value is greater than or equal to –127 and less than or equal to 128, use `Int8`. If the maximum index value is greater than or equal –32,767 and less than or equal to –127, use `Int16`. If the maximum index value is less than –32,767 or greater than or equal to 32,768, use `Int32`.

Metafile Object Specifications

The following sections contain descriptions of all currently valid metafile objects. Each section concerns a particular type of metafile object, and indicates the required form of specification for objects of that type in text files and in binary files. Each section also includes an example of a valid text file object specification and other pertinent information.

Special Metafile Objects

This section describes seven special metafile objects.

3D Metafile Header

LABELS

ASCII	`3DMetafile`
Binary	`3DMF (= 0x33444D46)`

Each metafile header includes a flag that indicates the uses to which file pointers and reference objects are put in that metafile. The flags that may appear in a metafile header are defined by the metafile flags data enumeration.

METAFILE FLAGS

```
Normal          0x00000000
Stream          0x00000001
Database        0x00000002
```

Constant descriptions

`Normal` This flag indicates that, for every shared object specified in the metafile, if that object is instanced more than once in the metafile, then all instantiations of that object other than its original specification are accomplished through the use of file pointers and reference objects. If this flag is set, then the full specification of an object never appears more than once in the metafile. In order to read a normal metafile, a parser should have random access to that file.

`Stream` This flag indicates that there are no internal references in the metafile. (A reference in a file to an object located in the same file is termed *internal*; a reference in a file to an object located in a different file is termed *external*.) In order to read a stream metafile, a parser need have sequential access only.

`Database` This flag indicates that every shared object in the metafile that is not itself a reference object is the target object of a file pointer appearing in a table of contents in the metafile. All of the contents of a database metafile may be discovered by a parser through examination of its tables of contents.

DATA FORMAT

```
Uns16                   majorVersion
Uns16                   minorVersion
MetafileFlags           flags
FilePointer             tocLocation
```

Field descriptions

`majorVersion`	The version number of the metafile. Currently, the version number is 1.
`minorVersion`	The revision number of the metafile. Currently, the revision number is 0.
`flags`	The flag of the header.
`tocLocation`	A file pointer to the location (in the metafile) of a table of contents object. If the value in this field is `NULL`, then the entire metafile must be parsed in order to find any extant tables of contents.

DATA SIZE

20

DESCRIPTION

A metafile header is a structure having four fields. The first two fields specify the version and revision numbers of the metafile. The third field contains a flag indicating the type of the metafile (normal, stream, or database). The fourth field contains a pointer to the location of a table of contents for the metafile. A metafile header in a file indicates that the file is a metafile and provides some information about its contents.

Each metafile must contain exactly one metafile header, and this header must precede every other object in that file. Though each metafile header contains a pointer to the location of a table of contents, there need be no corresponding table of contents in the metafile.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
3DMetafile (
   1  0               # majorVersion, minorVersion
   Normal             # flag
   toc>               # file pointer
)

.
.
.                     # list of objects

toc: TableOfContents ( ... )
```

Tables of Contents

LABELS

ASCII	`TableOfContents`
Binary	`toc` (= `0x746F6320`)

DATA TYPE DEFINITION: TOC ENTRY TYPE 0

```
TOCEntry (
   Uns32                refID
   FilePointer          objLocation
)
```

SIZE

12

DATA TYPE DEFINITION: TOC ENTRY TYPE 1

```
TOCEntry (
   Uns32                refID
   FilePointer          objLocation
   ObjectType           objType
)
```

SIZE

16

Field descriptions

`refID`	The value of the `refID` field of a reference object.
`objLocation`	A pointer to the location of a referenceable metafile object.
`objType`	The type tag of the target object of the file pointer listed in the `objLocation` field.

Note
Type 1 table of contents entries allow a parser to determine the type of a referenced object by inspection of tables of contents; type 0 table of contents entries do not. The table of contents entries in a stream metafile normally are of type 0; the table of contents entries in a database metafile normally are of type 1. ◆

DATA FORMAT

```
FilePointer          nextTOC
Uns32                refSeed
Int32                typeSeed
Uns32                tocEntryType
Uns32                tocEntrySize
Uns32                nEntries
TOCEntry             tocEntries[nEntries]
```

Field descriptions

`nextTOC`	A pointer to the location of the next table of contents in the metafile. (If there is no subsequent table of contents, then this pointer is idle.)
`refSeed`	The least integer that may occur in the `refID` field of a reference object added to the metafile after this table of contents is written. The value in this field must be greater than 0 and is incremented whenever a new reference object is added to the preceding section of the metafile or is listed in a TOC entry added to this table of contents.
`typeSeed`	The greatest integer that may occur in the `typeID` field of a type object added to the metafile after this table of contents is written. The value in this field must be less than 0 and is decremented whenever a new type object is added to the preceding section of the metafile.
`tocEntryType`	A numerical constant that indicates the type of the entries contained in the table of contents. The permitted values of this field are 0 and 1. A value of 0 indicates that all entries in the array `tocEntries[]` are of type 0; a value of 1 indicates that all entries in that array are of type 1. The occurrence of this constant should cause no confusion, as all entries in any particular table of contents must be of the same type.
`tocEntrySize`	A numerical constant that indicates the binary sizes of the entries contained in the table of contents. The permitted values of this field are 12 and 16. If the value in the previous field is 0, then the value in this field must be 12; if the value in the previous field is 1, then the value in this field must be 16. Again, this constant should cause no confusion, as all entries in any particular table of contents must be of the same size.
`nEntries`	The number of entries contained in the table of contents; that is, the size of the array `tocEntries[]`. If the value in this field is 0, then that array is empty.
`tocEntries[]`	An array of `TOCEntry` objects, all of which are of the same entry type.

DATA SIZE

```
20 + (tocEntrySize * nEntries)
```

DESCRIPTION

A table of contents is a structure that provides a record of associations made between reference objects and file pointers. These associations are reported by the TOC entries of the table of contents. A metafile reader must use its tables of contents to discover linkages between reference objects and file pointers, as there is no other record of those associations. See the sections "File Pointers" on page 1-24 and "Reference Objects" on page 1-37 for complete details regarding these objects.

A reference in a file to a target object located in the same file is termed *internal*; a reference in a file to a target object located in a different file is termed *external*. A metafile in which there are internal references or file pointers whose target objects are not tables of contents must include at least one table of contents.

If a metafile contains more than one table of contents, then each table of contents should continue the record provided by the immediately previous table of contents (if such exists) without duplication. A table of contents may contain information about objects occurring before or after it or both, but should not contain information about any object that either precedes an object mentioned in a previous table of contents or follows an object mentioned in a subsequent table of contents.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
3DMF (
   1  0
   Normal
   toc>
)
.
.
.
cube: box ( ... )
rotateTransform( ...)
Reference ( 1 )                          # internal reference
.
.
.
scaleTransform ( . . . )
Container (
   Reference (4)                         # external reference
   UnixPath ( ".../geometryobjecs/ellipsoids/elli.1.a" )
)
.
.
.
Type ( -1 "Lino" )
.
.
.
tory: Torus ( ... )
.
.
.
Reference (2)
.
.
.
Reference ( 1 )
.
```

```
.
.
Reference ( 1 )
.
.
.
toc: TableOfContents (
          nextTOC>
          5                        # refSeed
          -2                       # typeSeed
          0                        # tocEntryType
          12                       # tocEntrySize
          2                        # nEntries
          1  cube>                 # TOCEntries
          2  tory>
       )
```

Reference Objects

LABELS

ASCII	`Reference`
Binary	`rfrn (= 0x7266726E )`

DATA FORMAT

`Uns32`	`refID`

Field descriptions

`refID` A nonnegative integer. If the value in this field is not 0, then this reference object is linked to a file pointer and is used to reference the target object of that file pointer. If the value in this field is 0, then this reference object is not linked to a file pointer. A value of 0indicates that the object to be referenced is an entire file rather than an object located within a file. The relevant file is specified in a

storage object that occurs as a child object to the reference object.

DATA SIZE

4

DESCRIPTION

A reference object is used to permit an object defined elsewhere to be referenced at one or more locations in a metafile. A reference in a file to a target object located in the same file is termed *internal*; a reference in a file to a target object located in a different file is termed *external*.

A reference object and the object it is intended to reference may be contained in separate files. In such a case, the reference object must also have, as a child object, a storage object that indicates the location of the home file of the object to be referenced. See the sections "UNIX Path" on page 1-39 and "Macintosh Path" on page 1-41 for descriptions of these objects. See the section "File Pointers" on page 1-24 for an explanation of the relationships that must obtain among a reference object, a table of contents, and a file pointer in order for that reference object to serve its purpose.

PARENT HIERARCHY

Shared.

PARENT OBJECTS

A reference object sometimes but not always has a parent object.

CHILD OBJECTS

One UNIX path or Macintosh path object (optional). A reference object must contain a child object whenever the object it is used to reference is (or is located in) a separate file.

EXAMPLE

```
.
.
.
Reference ( 23 )                #internal reference
.
.
.
toc: TableOfContents
   nextTOC> 35 -1 0 12
   .
   .
   .
   20 CarFrame>
   21 Axle>
   23 WheelOfCar>
   .
   .
   .
   )
.
.
.
Container (                     #external reference
Reference ( 23 )
UnixPath ( ".:car:parts.eb" )   #Unix pathname Object
)
```

UNIX Path

LABELS

ASCII	`UnixPath`
Binary	`unix (= 0x756E6978)`

DATA FORMAT

```
String          unixPath
```

Field descriptions

`unixPath` A character string consisting of a pathname enclosed in double quotation marks.

DATA SIZE

```
sizeof(pathName)
```

DESCRIPTION

A UNIX path object may occur only as the child object of a reference object. Its purpose is to allow a reference object located in one file to reference an object located in another file. The file pointer, label, and table of contents entry associated with the parent reference object are located in the home file of the target object.

PARENT HIERARCHY

Shared, Storage.

PARENT OBJECTS

Always.

CHILD OBJECTS

None.

EXAMPLE

```
Container (                     #external reference
Reference ( 23 )
UnixPath ( ".:Graphics:Models:Conics:Cone.3" )
)
```

Note
In the above example, the object to be referenced is in the file Cone.3. The file reader must open that file and search its tables of contents to find the file pointer associated with the integer 23 in order to locate that object. ◆

Macintosh Path

LABELS

ASCII	`MacintoshPath`
Binary	`alis ( = 0x616C6973 )`

DATA FORMAT

`String`	`pathName`

Field descriptions

`pathName`	A character string consisting of a Macintosh pathname enclosed in double quotation marks. The pathname should be specified in accordance with Macintosh pathname conventions.

DATA SIZE

```
sizeof(String)
```

DESCRIPTION

A Macintosh path object may occur only as the child object of a reference object. Its purpose is to allow a reference object located in one file to access a target object located in another file. The file pointer, label, and table of contents entry associated with the parent reference object are located in the home file of the target object. This object may be used only on platforms running on the Macintosh Operating System.

PARENT HIERARCHY

Shared, storage.

PARENT OBJECTS

Always. This object may occur only as the child object of a reference object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   Reference ( 17 )
   MacintoshPath ( ... :3DGraphics:Models:Stemware.2 )
```

Note
In the above example, the object to be referenced is in the file Stemware.2. The file reader must open that file and search its tables of contents to find the file pointer associated with the integer 17 in order to locate that object. ◆

Types

LABELS

ASCII	`Type`
Binary	`type (= 0x74797065)`

DATA FORMAT

`Int32`	`typeID`
`String`	`owner`

Field descriptions

`typeID`	A negative integer. No two type objects in the same file may have the same value in this field.
`owner`	An ISO 9070 registered owner string. The value of this field may not occur in any other type object.

DATA SIZE

4 + `sizeof(String)`

DESCRIPTION

The metafile file format permits the inclusion of custom objects, provided that they have been assigned a type. The type object is used to declare a custom data type, and must be used whenever a custom data type appears in a metafile. A type object must appear in a file prior to any custom object of that type. At most 2^{31} (= 2,147,483,648) custom types may appear in a single file.

Note
To include a custom object in a binary metafile, you must specify the size of that object, padded to the nearest byte. To include a custom object in a text metafile, enclose the object in parentheses and prefix the object's `typeID` number. ◆

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Type (
    -1
    "Program: Tree"
)

Type (
    -1
    "Program: Forest"
)
```

Containers

LABELS

ASCII	`Container`
Binary	`cntr ( = 0x636E7472 )`

DATA FORMAT

No data.

DATA SIZE

$8k + \Sigma$, where k is the number of elements of the container and Σ is the sum of the sizes of those elements.

DESCRIPTION

A container is an ordered collection of objects. Containers are used to form complex objects from simpler objects in ways permitted by the structure of the metafile object hierarchy. In particular, child objects are attached to parent objects through the use of containers. Every container must contain at least one object. Containers may be nested. The relation of containment is not transitive (that is, the elements of a container occurring within another container are not themselves elements of the latter container). However, an object may be

instantiated more than once in a hierarchy of nested containers, as shown in the example at the end of this section.

The notation for containers in text files is as follows:

```
Container (
    object0
    .
    .
    .
    objectnobjects-1
)
```

Notations for contained objects are separated by blank spaces rather than by punctuation marks, as is the case in the notation for other objects having nonzero size.

The first element of a container is called the *root object* of that container. The root object of a container must be a shared object, may not be a container itself, and may not be the target object of a file pointer. The position in the metafile object hierarchy of the root object of a container constrains the number, type, and in some cases the order of occurrence of other elements of that container. Each element of a container other than the root object must be either a legitimate child object of the root object or another container. In the latter case, the root object of the inner container must be a legitimate child object of the root object of the outer one.

A container may be the target object of a file pointer.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    Cylinder ( . . . )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0 1 0 )
    )
    Caps ( Bottom )
    Container (
        BottomCapAttributeSet ( )
        Container (
        AttributeSet ( )
        DiffuseColor ( 0 1 0 )
        )
    )
)
```

String Objects

C Strings

LABELS

ASCII	`CString`
Binary	`strc(= 0x73747263)`

DATA FORMAT

`String`	`cString`

Field descriptions

`cString`	A string constant (that is, a sequence of ASCII characters enclosed in double-quotation marks). See the section

"Strings" on page 1-23 for a list of the escape sequences that may occur in a `cString` object.

DATA SIZE

```
sizeof(String)
```

DESCRIPTION

A C string may be used to include text in a metafile.

PARENT HIERARCHY

Shared, string.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
cString (
"Copyright Apple Computer, Inc., 1995"
)
```

Unicode Objects

LABELS

ASCII	`Unicode`
Binary	`uncd (= 0x756E6364)`

DATA FORMAT

```
uns32           length
RawData         unicode[length * 2]
```

Field descriptions

`length` The length of the encoded text.

`unicode[]` An array of raw data that encodes text.

DATA SIZE

```
4 + length * 2
```

DESCRIPTION

A unicode object may be used to include text in a binary metafile.

PARENT HIERARCHY

Shared, String.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Unicode (
   6
   0x457363686572
)
```

Geometric Objects

This section describes the geometric objects currently supported by the metafile specification.

Points

LABELS

ASCII	`Point`
Binary	`pnt ( = 0x706E7420 )`

DATA FORMAT

`Point3D`	`point`

Field descriptions

`point`	A three-dimensional point.

DATA SIZE

12

DESCRIPTION

A point object is used to specify a point in world space. A point object may appear only in a group or as part of the definition of a custom data type. Unlike the corresponding point data type, a geometric point object may be assigned attributes such as color. Thus, an application may use point objects to specify visible dots.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Point ( 0 0 0 )
```

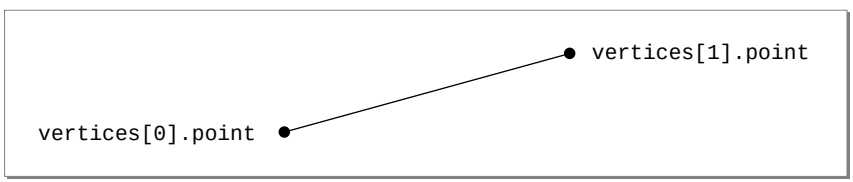
DEFAULT SIZE

None.

Lines

Figure 1-3 shows a line.

Figure 1-3 A line



LABELS

ASCII	`Line`
Binary	`line ( = 0x6C696E65 )`

DATA FORMAT

```
Point3D          start
Point3D          end
```

Field descriptions

`start` One endpoint of the line.

`end` The other endpoint of the line.

DATA SIZE

24

DESCRIPTION

A line is a straight segment in three-dimensional space defined by its two endpoints. Attributes may be assigned to the vertices of a line and to the entire line.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a line is (0, 0) at `start` and (1, 0) at `end`.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional), vertex attribute set list (optional). An attribute set may be used to assign attributes to the entire line. The vertex attribute set list may include attribute sets for one or both vertices of the line. For the purpose of attribute assignment, the `start` and `end` vertices of a line are indexed by the

integers 0 and 1 respectively. See the section "Vertex Attribute Set Lists" on page 1-141 for a description of these lists.

EXAMPLE

```
Container (
   Line (
      0 0 0
      1 0 0
   )
   Container (
      VertexAttributeSetList ( 2 Exclude 0 )
      Container (
         AttributeSet ( )
         DiffuseColor ( 1 0 0 )
      )
      Container (
         AttributeSet ( )
         DiffuseColor ( 0 0 1 )
      )
   )
)
```

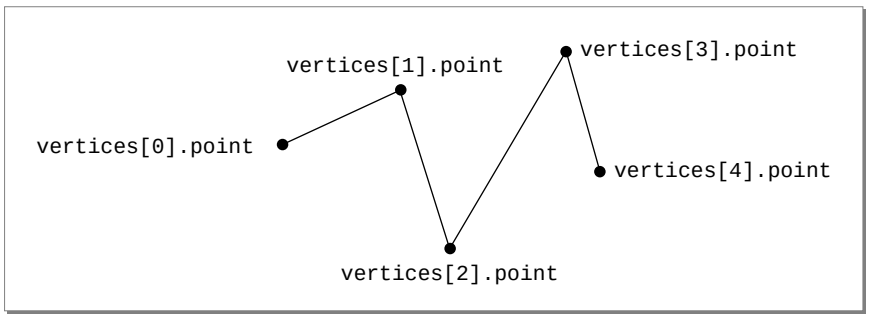
DEFAULT SIZE

None.

Polylines

Figure 1-4 shows a polyline.

Figure 1-4 A polyline



LABELS

ASCII	`Polyline`
Binary	`plin ( = 0x706C696E )`

DATA FORMAT

```
Uns32           numVertices
Point3D         vertices[numVertices]
```

Field descriptions

`numVertices` The number of vertices of the polyline.

`vertices[]` An array of vertices that define the polyline.

DATA SIZE

```
4 + (numVertices * 12)
```

DESCRIPTION

A polyline is a collection of n lines defined by the $n+1$ points that define the vertices of its segments. For $1 \leq i \leq n-1$, the second vertex of the ith line is the first vertex of the $i+1$st line; the $n+1$st vertex of a polyline is not connected to the first. Attributes may be assigned separately to each vertex and to each segment of a polyline as well as to the entire polyline.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set, geometry attribute set list, vertex attribute set list. Use a vertex attribute set list to assign attribute sets to as many vertices as desired; use a geometry attribute set list to assign attribute sets to as many segments as desired. Use an attribute set to assign attributes to the entire polyline.

EXAMPLE

```
PolyLine(
   5                #numVertices
   0  0  0          #first vertex
   1  1  0          #second vertex
   .5 .5 0
   0  1  0
   1  1  0
)
```

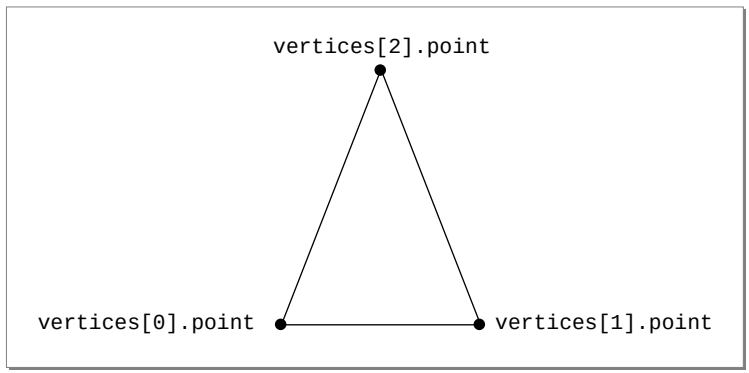
DEFAULT SIZE

None.

Triangles

Figure 1-5 shows a triangle.

Figure 1-5 A triangle



LABELS

ASCII	`Triangle`
Binary	`trng` (= `0x74726E67`)

DATA FORMAT

`Point3D`	`vertices[3]`

Field descriptions

`vertices[]`	An array of triangle vertices.

DATA SIZE

36

DESCRIPTION

A triangle is a closed plane figure defined by three vertices. Attributes may be assigned to each vertex of a triangle and also to its entire face.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Vertex attribute set list (optional), attribute set (optional). A vertex attribute set list may be used to attach attributes to one or more vertices of the triangle. An attribute set may be used to attach attributes to the entire face of the triangle.

EXAMPLE

```
Container (
   Triangle (
      -1 -0.5 -0.25
      0 0 0
      -0.5 1.5 0.45
   )
   Container (
      VertexAttributeSetList ( 3 Exclude 0 )
      Container (
         AttributeSet ( )
         DiffuseColor ( 1 0 0 )
      )
      Container (
         AttributeSet ( )
         DiffuseColor ( 0 1 0 )
      )
      Container (
         AttributeSet ( )
         DiffuseColor ( 0 0 1 )
      )
   )
   Container (
      AttributeSet ( )
```

```
        DiffuseColor ( 0.8 0.5 0.2 )
    )
)
```

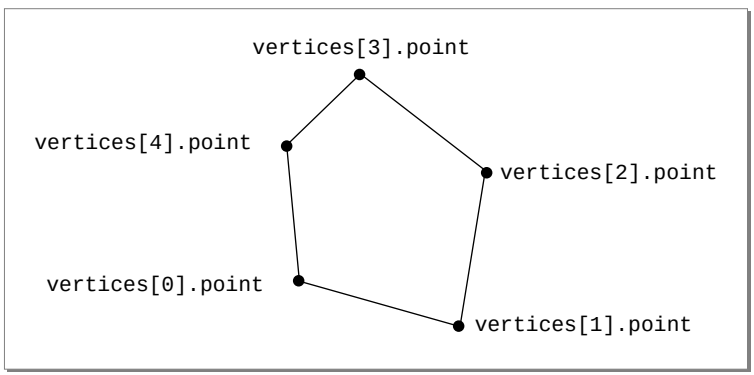
DEFAULT SIZE

None.

Simple Polygons

Figure 1-6 shows a simple polygon.

Figure 1-6 A simple polygon



LABELS

ASCII	`Polygon`
Binary	`plyg` (= `0x706C7967`)

DATA FORMAT

`uns32`	`nVertices`
`Point3D`	`vertices[nVertices]`

Field descriptions

`nVertices`	The number of vertices of the polygon.
`vertices[]`	An array of vertices that define the polygon.

DATA SIZE

```
4 + (numVertices * 12)
```

DESCRIPTION

A simple polygon is a convex plane figure defined by a list of vertices. In other words, a simple polygon is a polygon defined by a single contour. (Vertices are assumed to be coplanar to within floating-point tolerances.) The lines connecting the vertices of a simple polygon do not cross. Attributes may be assigned to each vertex of a simple polygon and also to its entire face.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Vertex attribute set list (optional), attribute set (optional). A vertex attribute set list may be used to attach attribute sets to one or more vertices of the simple polygon. An attribute set may be used to attach attributes to the entire face of the simple polygon. For the purpose of attribute assignment, the vertices of a polygon are indexed by position in the array `vertices[]`; that is, the index of `vertices[i]` is *i*. See the section "Vertex Attribute Set Lists" on page 1-141 for an explanation of the structure and syntax of these objects.

EXAMPLE

```
Polygon(
   5                 #nVertices
   0   0   0
   1   0   0
   2   1   0
   1   2   0
   0   1   0
)
```

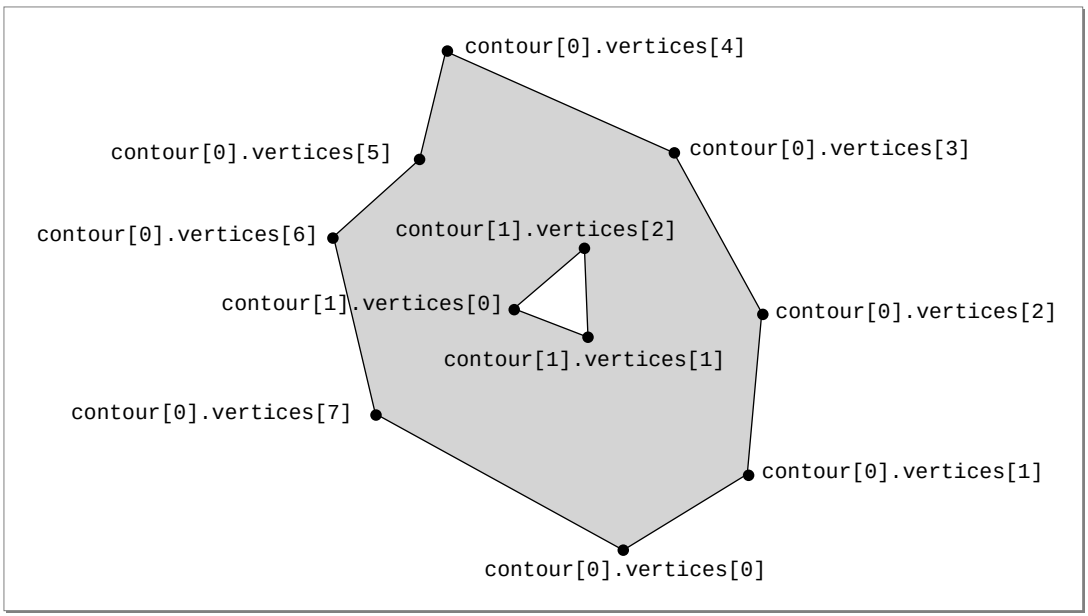
DEFAULT SIZE

None.

General Polygons

Figure 1-7 shows a general polygon.

Figure 1-7 A general polygon



LABELS

ASCII	`GeneralPolygon`
Binary	`gpgn` (= `0x6770676E`)

POLYGON DATA DATA TYPE

`uns32`	`nVertices`
`Point3D`	`vertices[nVertices]`

Field descriptions

`nVertices`	The number of vertices of this contour of the general polygon.
`vertices[]`	An array of vertices that define this contour of the general polygon.

DATA FORMAT

```
Uns32                   nContours
PolygonData             polygons[nContours]
```

Field descriptions

`nContours` The number of contours of the general polygon.

`polygons[]` An array of contours that define the general polygon.

DATA SIZE

```
sizeof(PolygonData) = 4 + nVertices * 12
sizeof(GeneralPolygon) = 4 + sizeof(polygons[0...nContours-1])
```

DESCRIPTION

A general polygon is a closed plane figure defined by one or more lists of vertices. In other words, a general polygon is a polygon defined by one or more contours. Each contour may be concave or convex, and contours may be nested. All contours, however, must be coplanar. A general polygon can have holes in it. If it does, the even-odd rule is used to determine which regions are included in the polygon. Attributes may be assigned to each vertex of each contour of a general polygon and to the entire general polygon.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set, general polygon hint, vertex attribute set list (all optional). Use an attribute set to attach attributes to an entire general polygon. Use a general polygon hint to specify whether a general polygon is concave, convex, or complex; see the section "General Polygon Hints" on page 1-64 for complete details on this object. Use a vertex attribute set list to assign attributes to the vertices of the contours of a general polygon. For purposes of attribute assignment, the vertices of a general polygon are indexed in the order of their occurrence in the specification of that polygon; the index does not distinguish between contours. For purposes of attribute assignment, the *n*th contour of a general polygon is the contour defined by `(polygons[n-1])[1]`, and the index of the *n*th contour is *n*–1. The *n*th vertex of a general polygon is the *p*th vertex of the *m*th contour, where

$m = \max\{k \le$ `nContours` $: \Sigma_{0 \le i < k-1}$ `(polygons[i])[0] < n`$\}$,

and $n = \Sigma_{0 \le i < m}$ `(polygons[i])[0]` $+ p$; the index of the *n*th vertex of a general polygon is *n*-1. The *p*th vertex of the *m*th contour of a general polygon is the $(\Sigma_{0 \le i < \mu-1}$ `(polygons[`i`])[0]` $+ p)$th vertex of the general polygon; its index is $\Sigma_{0 \le i < \mu-1}$ `(polygons[`i`])[0]` $+ (p - 1)$. See the sections "Face Attribute Set Lists" on page 1-137 and "Vertex Attribute Set Lists" on page 1-141 for explanations of the structure and syntax of these objects.

EXAMPLE

```
Container (
   GeneralPolygon (
      2                    # nContours
   #contour 0
      3                    # nVertices, contour 0
      -1 0 0               # vertex 0
      1 0 0                # vertex 1
      0 1.7 0              # vertex 2
   #contour 1
      3                    # nVertices, contour 1
      -1 0.4 0             # vertex 3
      1 0.4 0              # vertex 4
      0 2.1 0              # vertex 5
   )
   Container (
```

```
        VertexAttributeSetList ( 6 Exclude 2 0 4 ) #see note
        Container (
            AttributeSet ( )              # vertex 1
            DiffuseColor ( 0 0 1 )
        )
        Container (
            AttributeSet ( )              # vertex 2 (contour 0)
            DiffuseColor ( 0 1 1 )
        )
        Container (
            AttributeSet ( )              # vertex 3 (contour 1)
            DiffuseColor ( 1 0 1 )
        )
        Container (
            AttributeSet ( )              # vertex 5 (contour 1)
            DiffuseColor ( 1 1 0 )
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor ( 1 1 1 )
    )
)
```

Note
In the above example, the general polygon has two contours. Each contour is a triangle. The triangles overlap. The intersection of the triangles is included in an even number of contours; thus, it constitutes a hole in the general polygon. The relative complements of the triangles are included in an odd number of contours; thus, they are included in the general polygon. ◆

DEFAULT SIZE

None.

General Polygon Hints

LABELS

ASCII	`GeneralPolygonHint`
Binary	`gplh` (= `0x67706C68`)

GENERAL POLYGON HINTS

`Complex`	`0x00000000`
`Concave`	`0x00000001`
`Convex`	`0x00000002`

Constant descriptions

`Complex`	The parent general polygon may include concave, convex, and self-intersecting polygons.
`Concave`	All contours of the parent general polygon are concave and none is self-intersecting.
`Convex`	All contours of the parent general polygon are convex and none is self-intersecting.

DATA FORMAT

```
GeneralPolygonHintEnum        shapeHint
```

Field descriptions

`shapeHint`	The value in this field must be one of the constants defined above.

DATA SIZE

4

DESCRIPTION

A general polygon hint object is used to provide a reading application with an indication of the shape of a general polygon.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Data.

PARENT OBJECTS

General polygon. A general polygon hint object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   GeneralPolygon ( ... )
   GeneralPolygonHint ( Complex )
)
```

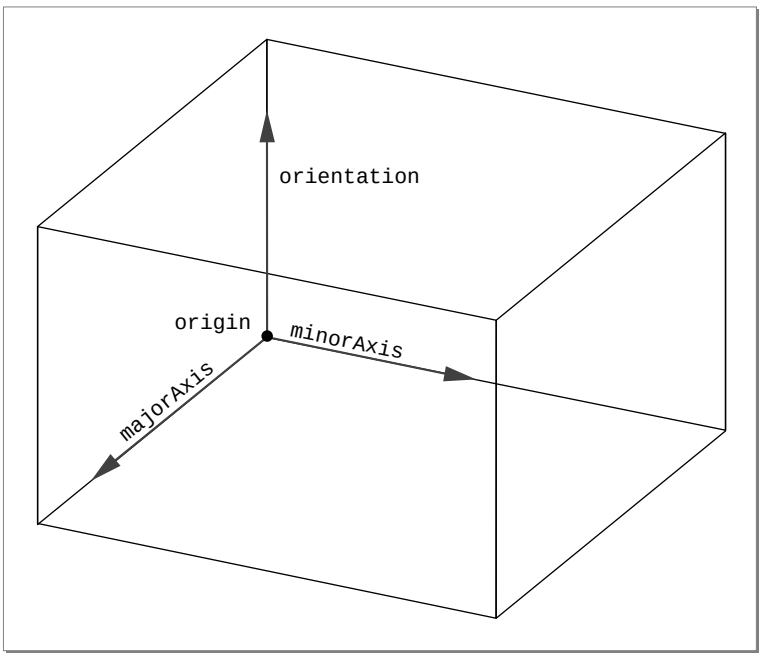
DEFAULT VALUE

`Complex`.

Boxes

Figure 1-8 shows a box.

Figure 1-8 A box



LABELS

ASCII	`Box`
Binary	`box (= 0x626F7820)`

DATA FORMAT

```
Vector3D        orientation
Vector3D        majorAxis
Vector3D        minorAxis
Point3D         origin
```

Field descriptions

`orientation` The orientation of the box.

`majorAxis` The major axis of the box.

`minorAxis`	The minor axis of the box.
`origin`	The origin of the box.

DATA SIZE

0 or 48

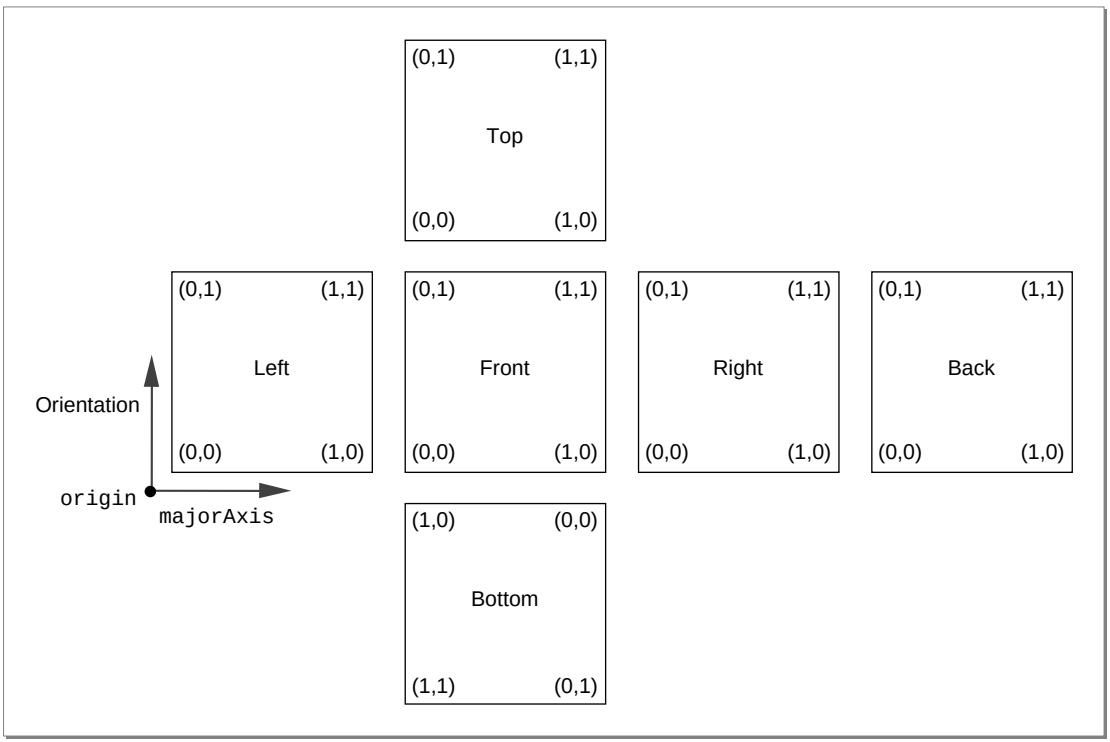
DESCRIPTION

A box is a three-dimensional object defined by an origin (that is, a corner of the box) and three vectors that define the edges of the box that meet in that corner. A box may be used to model a cube, rectangular prism, or other parallelipiped. Attributes may be applied to each of the six faces of a box and to the entire geometry of the box.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a box is as shown in Figure 1-9.

Figure 1-9 The default surface parameterization of a box



PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Face attribute set list (optional), attribute set (optional). For the purpose of attribute assignment, the faces of a box are indexed as follows:

0	The face perpendicular to the orientation vector having the endpoint of the orientation vector as one of its vertices. In Figure 1-9, this is the top face of the box.
1	The face perpendicular to the orientation vector having the origin as one of its vertices. In Figure 1-9, this is the bottom face of the box.
2	The face perpendicular to the major axis having the endpoint of the major axis as one of its vertices. In Figure 1-9, this is the front face of the box.
3	The face perpendicular to the major axis having the origin as one of its vertices. In Figure 1-9, this is the back face of the box.
4	The face perpendicular to the minor axis having the endpoint of the minor axis as one of its vertices. In Figure 1-9, this is the right face of the box.
5	The face perpendicular to the minor axis having the origin as one of its vertices. In Figure 1-9, this is the front face of the box.

EXAMPLE

```
Container (
   Box ( ... )
   Container (
      FaceAttributeSetList (6 Exclude 2 1 4)
      Container (
         AttributeSet ( )              #left face
         DiffuseColor ( 1 0 0 )
      )
      Container (
         AttributeSet ( )              #bottom face
         DiffuseColor ( 0 1 1 )
      )
      Container (
         AttributeSet ( )              #top face
         DiffuseColor ( 0 1 0 )
      )
      Container (
```

```
            AttributeSet ( )                #front face
            DiffuseColor ( 1 0 1 )
         )
      )
      Container (
         AttributeSet ( )
         DiffuseColor(0 0 0)
      )
)
```

DEFAULT SIZE

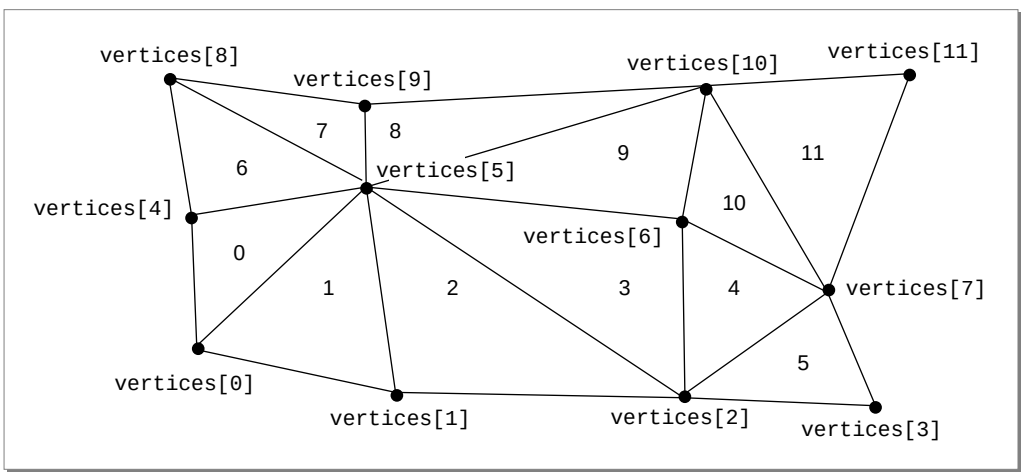
For objects of size 0, the default is

```
1 0 0
0 1 0
0 0 1
0 0 0
```

Trigrids

Figure 1-10 shows a trigrid.

Figure 1-10 A trigrid



LABELS

ASCII	`TriGrid`
Binary	`trig ( = 0x74726967 )`

DATA FORMAT

```
Uns32           numUVertices
Uns32           numVVertices
Point3D         vertices[numUVertices * numVVertices]
```

Field descriptions

`numUVertices` The number of vertices in the *u* parametric direction.

`numVVertices` The number of vertices in the *v* parametric direction.

`vertices[]` An array of vertices. The size of this array must equal the number of vertices of the trigrid. Vertices are to be listed in a rectangular order, first in the direction of increasing *v*, then in the direction of increasing *u*. That is, the vertex having parametric coordinates (u, v) precedes the vertex having parametric coordinates (u', v') if and only if either $u < u'$, or $u = u'$ and $v < v'$.

DATA SIZE

```
8 + (numUVertices * numVVertices * 12)
```

DESCRIPTION

A trigrid is a grid composed of triangular facets. The triangulation should be serpentine (that is, quadrilaterals are divided into triangles in an alternating fashion) to reduce shading artifacts when using Gouraud or Phong shading. Attributes may be assigned to each vertex and to each facet of a trigrid, and also to the entire trigrid.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Vertex attribute set list (optional), face attribute set list (optional), attribute set (optional). A face attribute set list may be used to assign attributes to the facets of a trigrid. The number of facets of a trigrid is the same as the number of its vertices. The vertices and facets of a trigrid are indexed in the manner shown by Figure 1-10. The vertex index prefers *u* to *v* and prefers 0 to 1; thus, it follows the canonical lexicographical ordering of the points in *uv* parametric space. The facet index is less easily defined but is readily apprehended. Consider first the serpentine path through the trigrid along the diagonals belonging to facets of the trigid. Now consider the alternative serpentine path composed of segments connecting all and only those vertices not on the first path. The second path passes through each facet and intersects all of the diagonals on the first path. The facets of the trigrid are numbered in the order that they would be encountered by a traveler along the second serpentine path.

EXAMPLE

```
Container (
   TriGrid (
      3          #numUVertices
      4          #numVVertices
      2  0  0      2  1  0      2  2  0      2  3  0
      1  0  0      1  1  0      1  2  0      1  3  0
      0  0  0      0  1  0      0  2  0      0  3  0
   )
   Container (
      FaceAttributeSetList (12 include 61 3 5 7 9 11)
      Container (
         AttributeSet( )
         DiffuseColor (1 1 1)
      )
      .
      .
      .
```

```
        Container (
            AttributeSet( )
            DiffuseColor (1 1 1)
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0 0 0 )
    )
)
```

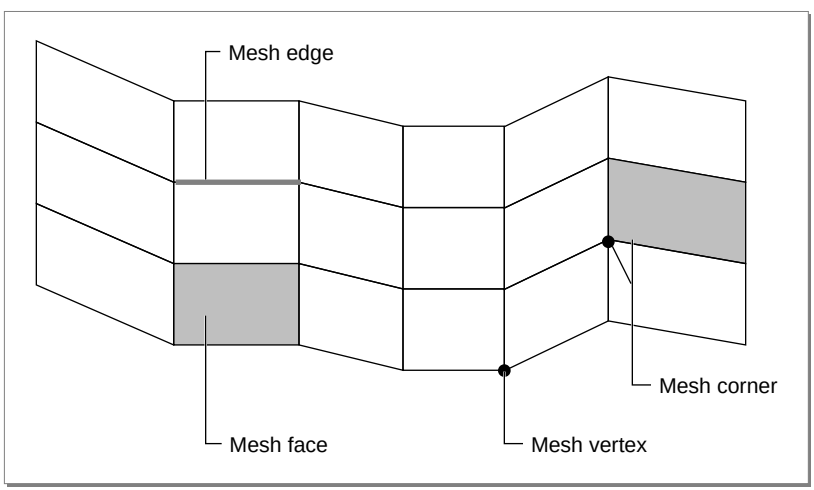
DEFAULT SIZE

None.

Meshes

Figure 1-11 shows a mesh.

Figure 1-11 A mesh



LABELS

ASCII	`Mesh`
Binary	`mesh ( = 0x6D657368 )`

MESH FACE DATA TYPE

`Int`	`nFaceVertexIndices`
`Uns`	`faceVertexIndices[|nFaceVertexIndices|]`

Field descriptions

`nFaceVertexIndices`

An integer the absolute value of which is equal to the number of indices to the vertices of a mesh face or mesh contour: that is, equal to the number of vertices of that face or contour. The value of this field may be positive or negative. A positive value indicates that this mesh face object specifies a face (to which attributes may be assigned). A negative value indicates that this mesh face object specifies a hole (here called a *contour*). The absolute value of the value in this field must be at least 3.

`faceVertexIndices[ ]`

An array of indices to elements of the array `vertices[]`, where *i* is the index of `vertices[i]`. This array specifies a verticed object by giving the indices of its vertices. The specified object is either a face or a contour of the mesh, as determined by the value of `nVertices`. The number of fields of this array must equal the absolute value of the value of the `nVertices` field.

DESCRIPTION

The mesh face data type is used to specify a verticed object and to specify whether that object is a face or a contour of a mesh. This data type occurs only as the value of a field in the `faces[]` array of a mesh specification.

DATA FORMAT

```
Uns32                   nVertices
Vertex3D                vertices[nVertices]
Uns32                   nFaces
Uns32                   nContours
MeshFace                faces[nFaces + nContours]
```

Field descriptions

`nVertices` The number of vertices of the mesh. The value of this field must be at least 3.

`vertices[]` An array of vertices.

`nFaces` The number of faces of the mesh.

`nContours` The number of contours of the mesh (that is, the number of holes in the mesh).

`faces[]` An array of mesh face objects, each of which specifies either a face or a contour (hole) of the mesh. The size of this array is equal to the sum of the values of the `nVertices` and `nContours` fields. Each array element that specifies a face should precede any and all array elements that specify holes in that face; any such latter elements may occur in any order but should be grouped together and should precede any subsequent array element that specifies a face: if the value of field *i* specifies a face intended to have *n* holes, then the objects that specify those holes must occupy the next *n* fields: that is, fields *i*+1, ..., *i*+*n*.

DATA SIZE

```
sizeof(MeshFace) = fabs(Int) * 4
sizeof(Mesh) = 4 + nVertices * 12 + 8 +
sizeof(faces[0...nFaces+nContours-1])
```

DESCRIPTION

A mesh is an object defined by a collection of vertices, faces, and contours. Meshes may be used to model polyhedra, grids, and other faceted objects. A mesh may have a boundary. The term *contour* is used here to refer to a

polygonal hole contained in a single face of a mesh. A mesh face (or contour) is a list of vertices that defines a polygonal facet. A face (or contour) need not be planar, and a contour and its surrounding face need not be coplanar; however, rendering of a mesh having a nonplanar face or contour, or having a contour not coplanar with its surrounding face, may lead to unexpected results.

The specification of a mesh includes an array of vertices and an array of faces and contours. The vertices of a mesh are indexed by array position; these indices are used to specify the faces and contours of that mesh. Faces and contours are also indexed by array position; this index does not distinguish between faces and contours. Both of these indices are used in the specification of child objects.

Attributes may be attached separately and selectively to the vertices, faces, face edges, and corners of a mesh.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Face attribute set list (optional), vertex attribute set list (optional), mesh corners (optional), mesh edges (optional). See the sections “Mesh Corners” on page 1-77 and “Mesh Edges” on page 1-80 for descriptions of these objects.

EXAMPLE

```
Mesh (
   10                                   # nVertices
      -1 1 1                            # enumeration of vertices
      -1 1 -1
```

```
        1 1 -1
        1 -1 -1
        1 -1 1
        0 -1 1
        -1 -1 0
        -1 -1 -1
        1 1 1
        -1 0 1
    7                                   # nFaces
    0                                   # nContours
        3 6 5 9                         # enumeration of contours
        5 7 6 9 0 1
        4 2 3 7 1
        4 2 8 4 3
        4 1 0 8 2
        5 4 8 0 9 5
        5 3 4 5 6 7
)
```

DEFAULT SIZE

None.

Mesh Corners

LABELS

ASCII	`MeshCorners`
Binary	`crnr ( = 0x63726E72 )`

MESHCORNER DATA TYPE

```
Uns32           vertexIndex
Uns32           nFaces
Uns32           faces[nFaces]
```

Field descriptions

`vertexIndex` The index of a vertex of the parent mesh.

`nFaces` The number of faces of the parent mesh sharing the vertex that is the value of the `vertexIndex` field which are to be correlated with child objects of the mesh corners object. The value of this field must not exceed the number of faces of the parent mesh meeting at the vertex whose index is the value of the `vertexIndex` field.

`faces[]` An array of face indices representing faces of the parent mesh. The vertex whose index is the value of the `vertexIndex` field must be among the vertices of each face of the parent mesh whose face index appears in this array. The number of fields of this array must equal the value of `nFaces`.

DATA FORMAT

```
Uns32          nCorners
MeshCorner     corners[nCorners]
```

Field descriptions

`nCorners` The number of corners of the parent mesh treated by this mesh corners object.

`corners[]` An array of mesh corners data types. The elements of this array are correlated with attribute sets which occur as child objects of the mesh corners object. The number of fields of this array must equal the value of `nCorners`.

DATA SIZE

```
sizeof(MeshCorner) = 8 + nFaces * 4
sizeof(MeshCorners) = 4 + sizeof(corners[0...nCorners-1])
```

DESCRIPTION

The mesh corners object is used to attach more than one attribute set to a vertex of a mesh and to override other attributes inherited by a vertex or assigned to it elsewhere. You can use mesh corners in various ways: for example, to apply

different normals and shadings in order to create the appearance of a sharp edge or peak. This object occurs only as a child object to a mesh and always has attribute sets as child objects of its own.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Mesh (always).

CHILD OBJECTS

Attribute sets (always). The number of child objects is equal to the value of the `numCorners` field. Child objects are correlated with elements of the array `corners[]` in the order of their occurrence in the specification of the mesh corners object and its child objects; that is, the *i*th child object is correlated with the *i*th element of the array `corners[]`.

EXAMPLE

```
Container (
    Mesh (...)                # parent mesh
    Container(
        MeshCorners (
            2                 # numCorners
            # Corner 0
            5 # vertexIndex
            2 # faces
            25 26 # face indices
            # Corner 1
            5 # vertexIndex
            2 # faces
            23 24 # face indices
        )
        Container (
            AttributeSet ( )
```

```
            Normal ( -0.2 0.8 0.3 )
        )
        Container (
            AttributeSet ( )
            Normal ( -0.7 -0.1 0.4 )
        )
    )
)
```

Mesh Edges

LABELS

ASCII	`MeshEdges`
Binary	`edge ( = 0x65646765 )`

MESH EDGE DATA TYPE

`Uns32`	`vertexIndex1`
`Uns32`	`vertexIndex2`

Field descriptions

`vertexIndex1`	The smaller of the indices of the two vertices of the mesh edge. The indices are taken from the vertex index of the parent mesh.
`vertexIndex2`	The larger of the indices of the two vertices of the mesh edge.

IMPORTANT

The edge defined by a mesh edge data type must be an edge of a face (not merely a contour) of the parent mesh. ▲

DATA FORMAT

`Uns32`	`nEdges`
`MeshEdge`	`edges[nEdges]`

Field descriptions

`nEdges`	The number of edges of the parent mesh treated by this Mesh Edge object. The value in this field must be greater than 0 and less than or equal to the number of edges of faces of the parent mesh.
`edges[]`	An array of mesh edge data types. The elements of this array are correlated with attribute sets that occur as child objects of the mesh edges object. The number of fields of this array must equal the value of `nEdges`.

DATA SIZE

```
4 + sizeof(edges[0...nEdges-1])
```

DESCRIPTION

The mesh edges object is used to attach attribute sets separately and selectively to one or more edges of faces of a mesh.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Mesh (always).

CHILD OBJECTS

Attribute sets (always). The number of child objects is equal to the value of the `nEdges` field. Child objects are correlated with elements of the array `edges[]` in the order of their occurrence in the specification of the Mesh Edges object and its child objects; that is, the *i*th child object is correlated with the *i*th element of the array `edges[]`.

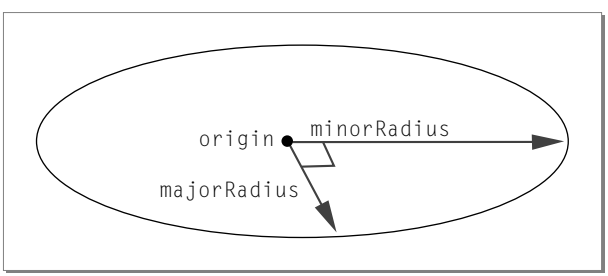
EXAMPLE

```
Container (
    Mesh ( ... )
    Container (
        MeshEdges (
        2                               # numEdges
        0 1                             # first edge
        1 3                             # second edge
    )
        Container (                     # first edge attribute set
            AttributeSet ( )
            DiffuseColor ( 0.2 0.8 0.3 )
        )
        Container (                     # second edge attribute set
            AttributeSet ( )
            DiffuseColor ( 0.8 0.2 0.3 )
        )
    )
)
```

Ellipses

Figure 1-12shows an ellipse.

Figure 1-12 An ellipse



LABELS

ASCII	`Ellipse`
Binary	`elps ( = 0x656C7073 )`

DATA FORMAT

```
Vector3D        majorAxis
Vector3D        minorAxis
Point3D         origin
```

Field descriptions

`majorAxis`	The (semi-) major axis of the ellipse.
`minorAxis`	The (semi-) minor axis of the ellipse.
`origin`	The center of the ellipse.

DATA SIZE

0 or 36

DESCRIPTION

An ellipse is a two-dimensional object defined by an origin (that is, the center of the ellipse) and two orthogonal vectors that define the major and minor radii of the ellipse. The origin and the two endpoints of the major and minor radii define the plane in which the ellipse lies. Attributes may be assigned only to the entire ellipse.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Ellipse (
   2   0   0          #majorRadius
   0   1   0          #minorRadius
   0   0   0          #origin
)
```

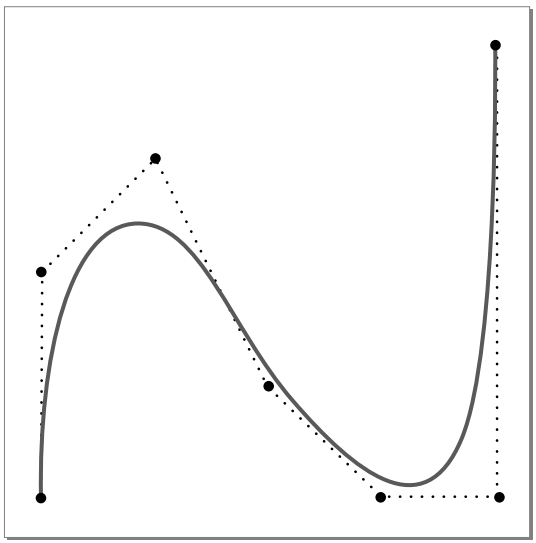
DEFAULT SIZE

For objects of size 0, the default is shown in the example above.

NURB Curves

Figure 1-13 shows a NURB curve.

Figure 1-13 A NURB curve



LABELS

ASCII	`NURBCurve`
Binary	`nrbc (= 0x6E726263)`

DATA FORMAT

```
Uns32                   order
Uns32                   nPoints
RationalPoint4D         points[nPoints]
Float32                 knots[order + nPoints]
```

Field descriptions

`order` The order of the NURB curve. For NURB curves defined by ratios of cubic B-spline polynomials, the order is 4. In general, the order of a NURB curve defined by polynomial equations of degree n is n+1. The value of this field must be greater than 1.

`nPoints`	The number of control points that define the NURB curve. The value of this field must be greater than 1.
`points[]`	An array of rational four-dimensional control points that define the NURB curve. The *w* coordinate of each control point must be greater than 0.
`knots[]`	An array of knots that define the NURB curve. The number of knots in a NURB curve is the sum of the values in the `order` and `nPoints` fields. The values in this array must be nondecreasing. Successive values may be equal, up to a multiplicity equivalent to the order of the curve; that is, if the order of a NURB curve is *n*, then at most *n* successive values may be equal.

DATA SIZE

```
8 + (nPoints * 16) + ((nPoints + order) * 4)
```

DESCRIPTION

A nonuniform rational B-spline (NURB) curve is a three-dimensional projection of a four-dimensional curve. A NURB curve is specified by its order, the number of control points used to define it, the control points themselves, and the knots used to define it. Attributes may be applied only to the entire NURB curve.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
NURBCurve (
   4                                        # order
   7                                        # nPoints
   0 0 0 1                                  # points
   1 1 0 1
   2 0 0 1
   3 1 0 1
   4 0 0 1
   5 1 0 1
   6 0 0 1
   0 0 0 0 0.25 0.5 0.75 1 1 1 1            # knots
)
```

DEFAULT SIZE

None.

2D NURB Curves

LABELS

ASCII	`NURBCurve2D`
Binary	`nb2c ( = 0x6E623263 )`

DATA FORMAT

```
Uns32                   order
Uns32                   nPoints
RationalPoint3D         points[nPoints]
Float32                 knots[order + nPoints]
```

Field descriptions

`order`	The order of the NURB curve. In general, the order of a NURB curve defined by polynomial equations of degree *n* is *n*+1. The value of this field must be greater than 1.
`nPoints`	The number of control points that define the 2D NURB curve. The value of this field must be greater than 1.
`points[]`	An array of three-dimensional control points that define the 2D NURB curve. The *z* coordinate of each point in this array must be greater than 0.
`knots[]`	An array of knots that define the 2D NURB curve. The number of knots in a NURB curve is the sum of the values in the `order` and `nPoints` fields. The values in this array must be nondecreasing, but successive values may be equal.

DATA SIZE

```
8 + 12 * nPoints + 4 * (order + nPoints)
```

DESCRIPTION

See the section "NURB Curves" on page 1-84 for a general description of NURB curves. 2D NURB curves occur only as child objects to trim loop objects, and trim loop objects occur only as child objects to NURB patches. This object is the only two-dimensional curve permitted by 3D metafile version 1.0.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Trim loop object (always).

CHILD OBJECTS

None.

DEFAULT SIZE

None.

Trim Loops

LABELS

ASCII	`TrimLoop`
Binary	`trml (= 0x74726D6C)`

DATA FORMAT

None.

DATA SIZE

0

DESCRIPTION

A trim loop object is used to bind two-dimensional curves to a NURB patch for the purpose of trimming that patch. As of this release, only 2D NURB curves may be used for trimming.

Trimming curves are attached to a NURB patch by placing them in a container the root object of which is a trim loop object and placing that container in a further container together with the relevant NURB patch.

The two-dimensional curves governed by a trim loop object must form a sequence such that the last control point of the *i*th curve is also the first control point of the *i*+1st curve, and the last control point of the last curve is also the first control point of the first curve.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Data.

PARENT OBJECTS

NURB patch (always).

CHILD OBJECTS

2D NURB curves (required). A trim loop object may have several child objects.

EXAMPLE

```
Container (
   NURBPatch (...)
   Container (
   TrimLoop ( )
      NURBCurve2D (...)
      .
      .
      .
      NURBCurve2D (...)
   )
```

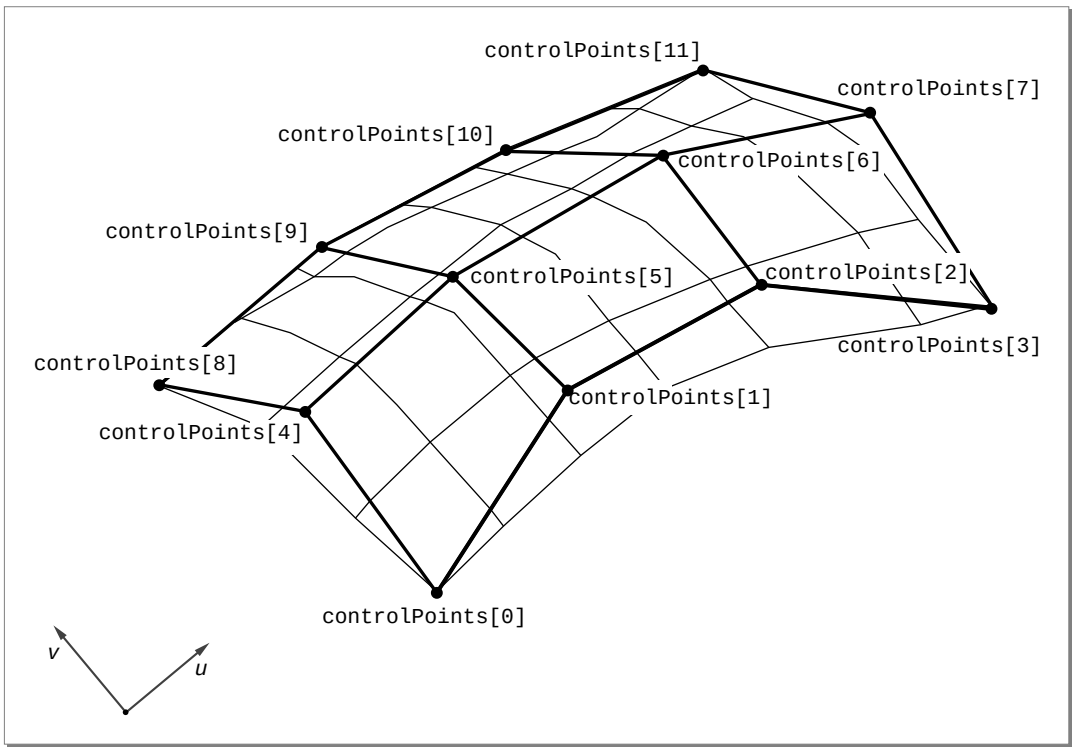
DEFAULT SIZE

None.

NURB Patches

Figure 1-14 shows a NURB patch.

Figure 1-14 A NURB patch



LABELS

ASCII	`NURBPatch`
Binary	`nrbp` (= `0x6E726270`)

DATA FORMAT

`Uns32`	`uOrder`
`Uns32`	`vOrder`
`Uns32`	`numMPoints`
`Uns32`	`numNPoints`

```
RationalPoint4D             points[numMPoints * numNPoints]
Float32                     uKnots[uOrder + numMPoints]
Float32                     vKnots[vOrder + numNPoints]
```

Field descriptions

`uOrder` The order of a NURB patch in the *u* parametric direction. For NURB patches defined by ratios of B-spline polynomials that are cubic in *u*, the order is 4. In general, the order of a NURB patch defined by polynomial equations in which *u* is of degree *n* is *n*+1.

`vOrder` The order of a NURB patch in the *v* parametric direction. For NURB patches defined by ratios of B-spline polynomials that are cubic in *v*, the order is 4. In general, the order of a NURB patch defined by polynomial equations in which *v* is of degree *n* is *n*+1.

`numMPoints` The number of control points in the *u* parametric direction. The value of this field must be greater than 1.

`numNPoints` The number of control points in the *v* parametric direction. The value of this field must be greater than 1.

`points[]` An array of rational four-dimensional control points that define the NURB patch. The size of this array is as indicated in the data format.

`uKnots[]` An array of knots in the *u* parametric direction that define the NURB patch. The values in this array must be nondecreasing, but successive values may be equal. The size of this array is as indicated in the data format.

`vKnots[]` An array of knots in the *v* parametric direction that define the NURB patch. The values in this array must be nondecreasing, but successive values may be equal. The size of this array is as indicated in the data format.

DATA SIZE

```
16 + (16 * numMPoints * numNPoints) + (uOrder + numNPoints +
vOrder + numMPoints) * 4
```

DESCRIPTION

A NURB patch is a three-dimensional surface defined by ratios of B-spline surfaces, which are three-dimensional analogs of B-spline curves.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization of a NURB patch is as shown in Figure 1-14.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Trim curves (optional). A trim curves object is a collection of two-dimensional NURB curves that are used to trim a NURB surface. See the sections "Trim Loops" on page 1-89 and "2D NURB Curves" on page 1-87 for descriptions of these objects.

EXAMPLE

```
NURBPatch (
   4                                                    #uOrder
   4                                                    #vOrder
   4                                                    #numMPoints
   4                                                    #numNPoints

   -2 2 0 1   -1 2 0 1   1 2 0 1   2 2 0 1             #points
   -2 2 0 1   -1 2 0 1   1 0 5 1   2 2 0 1
   -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
   -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
```

```
   0 0 0 0 1 1 1 1                                    #uKnots
   0 0 0 0 1 1 1 1                                    #vKnots
)
```

Note
The control points of a NURB patch are listed in a rectangular order, first in order of increasing *v*, then in order of increasing *u*. ◆

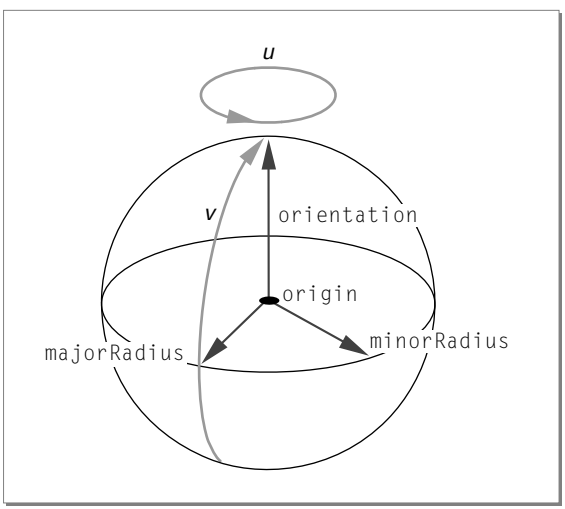
DEFAULT SIZE

None.

Ellipsoids

Figure 1-15 shows an ellipsoid.

Figure 1-15 An ellipsoid



LABELS

ASCII `Ellipsoid`

Binary `elpd ( = 0x656C7064 )`

DATA FORMAT

```
Vector3D        orientation
Vector3D        majorRadius
Vector3D        minorRadius
Point3D         origin
```

Field descriptions

`orientation` The orientation of the ellipsoid.

`majorRadius` The major radius of the ellipsoid.

`minorRadius` The minor radius of the ellipsoid.

`origin` The origin (that is, the center) of the ellipsoid.

DATA SIZE

0 or 48

DESCRIPTION

An ellipsoid is a three-dimensional object defined by an origin (that is, the center of the ellipsoid) and three pairwise orthogonal vectors that define the orientation and the major and minor radii of the ellipsoid.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for an ellipsoid is as shown in Figure 1-15. To the left of the major radius, $v = 0$; to the right of the major radius, $v = 1$. At the (top of the) orientation vector, and at the bottom of the ellipsoid, $u = 0$.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Ellipsoid ( )
Ellipsoid (
2 0 0
0 1 0
0 0 1
0 0 0
)
Container (
   Ellipsoid ( )
   Container (
      AttributeSet ( )
      DiffuseColor ( 1 1 0 )
   )
)
```

DEFAULT SIZE

For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
```

Caps

LABELS

ASCII	`Caps`
Binary	`caps (= 0x63617073)`

CAPS FLAGS

`None`	`0x00000000`
`Top`	`0x00000001`
`Bottom`	`0x00000002`

Constant descriptions

`None`	The parent cone or cylinder shall not have any caps.
`Top`	The parent cylinder shall have a cap at the end opposite to its base.
`Bottom`	The parent cone or cylinder shall have a cap at its base.

DATA FORMAT

`CapsFlags`	`caps`

Field descriptions

`caps`	A bitfield expression specifying one or more flags.

DATA SIZE

4

DESCRIPTION

A cap is a plane figure having the shape of an oval that closes the base of a cone or one end of a cylinder. A cone and a cylinder may each be supplied with a bottom cap. Only a cylinder may be supplied with a top cap. The length of the semimajor axis of a cap is equal to the length of the major radius of its parent object; the length of the semiminor axis of a cap is equal to the length of the minor radius of its parent object. A cap lies in a plane perpendicular to the orientation vector of its parent object. The center of a top cap is at the end of the orientation vector of its parent object; the center of a bottom cap is at the origin of its parent object. A separate attribute set may be assigned to each cap of an object having one or more caps.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cone, cylinder (always).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   Cone ( ... )
   Caps ( Top | Bottom )
   Container (
      BottomCapAttributeSet ( )
      DiffuseColor ( 0 1 0 )
   )
)
```

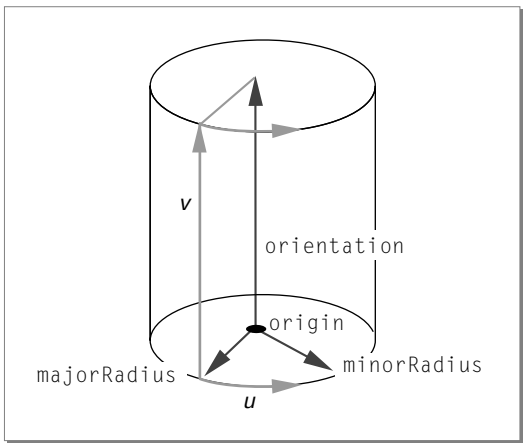
DEFAULT VALUE

None.

Cylinders

Figure 1-16 shows a cylinder.

Figure 1-16 A cylinder



LABELS

ASCII	`Cylinder`
Binary	`cyln ( = 0x63796C6E )`

DATA FORMAT

```
Vector3D        orientation
Vector3D        majorRadius
Vector3D        minorRadius
Point3D         origin
```

Field descriptions

`orientation`	The orientation of the cylinder.
`majorRadius`	The major radius of the cylinder.
`minorRadius`	The minor radius of the cylinder.
`origin`	The origin (that is, the center of the base) of the cylinder.

DATA SIZE

0 or 48

DESCRIPTION

A cylinder is a three-dimensional object defined by an origin (that is, the center of the cylinder) and three mutually perpendicular vectors that define the orientation and the major and minor radii of the cylinder. A cylinder may include a top cap, a bottom cap, or both. Attributes may be assigned to each included cap, to the face of the cylinder, and to the entire cylinder.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a cylinder is as shown in Figure 1-16.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Caps (top), top cap attribute set, caps (bottom), bottom cap attribute set, face cap attribute set, attribute set. All child objects are optional.

EXAMPLE

```
Cylinder ( )

Cylinder (
   0 2 0
   0 1 0
   0 0 1
   0 0 0
)
Container (
   Cylinder ( )
   Caps ( Bottom | Top )
   Container (
```

```
        BottomCapAttributeSet ( )
        Container (
        AttributeSet ( )
        DiffuseColor ( 0 1 0 )
        )
    )
    Container (
        FaceCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 1 )
        )
    )
    Container (
        TopCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 1 0 )
        )
    )
)
```

Note
In the above example, color attributes are attached to the surface of the cylinder very indirectly. As you see, color objects are elements of ordinary attribute sets rather than of cap attribute sets. Those attribute sets are elements of containers, which, in turn, are elements of cap attribute sets. The cap attribute sets serve to bind the ordinary attribute sets to the caps of the cylinder. ◆

DEFAULT SIZE

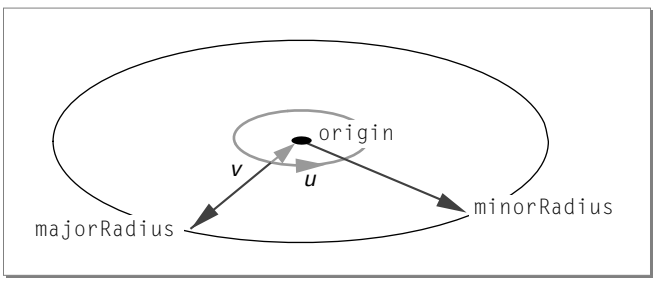
For objects of size 0, the default is:

1 0 0
0 1 0
0 0 1
0 0 0

Disks

Figure 1-17 shows a disk.

Figure 1-17 A disk



LABELS

ASCII	`Disk`
Binary	`disk ( = 0x6469736B )`

DATA FORMAT

`Vector3D`	`majorRadius`
`Vector3D`	`minorRadius`
`Point3D`	`origin`

Field descriptions

`majorRadius`	The major radius of the disk.
`minorRadius`	The minor radius of the disk.
`origin`	The center of the disk.

DATA SIZE

0 or 36

DESCRIPTION

A disk is a two-dimensional object defined by an origin (that is, the center of the disk) and two vectors that define the major and minor radii of the disk. A disk may have the shape of a circle, ellipse, or other oval. Attributes may be assigned to the entire disk only.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a disk is as shown in Figure 1-17.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Disk (
   1  0  0         # majorRadius
   0  1  0         # minorRadius
   0  0  0         # origin
)
```

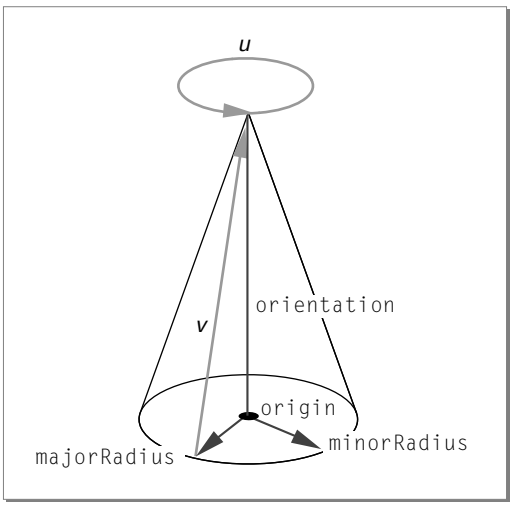
DEFAULT SIZE

For objects of size 0, the default is as in the previous example.

Cones

Figure 1-18 shows a cone.

Figure 1-18 A cone



LABELS

ASCII	`Cone`
Binary	`cone (= 0x636F6E65)`

DATA FORMAT

```
Vector3D        orientation
Vector3D        majorRadius
Vector3D        minorRadius
Point3D         origin
```

Field descriptions

`orientation`	The orientation of the cone. This vector also specifies the height of the cone.
`majorRadius`	The major radius of the cone.
`minorRadius`	The minor radius of the cone.
`origin`	The origin (that is, the center of the base) of the cone.

DATA SIZE

0 or 48

DESCRIPTION

A cone is a three-dimensional object defined by an origin (that is, the center of the base) and three vectors that define the orientation and major and minor radii of the cone. A cap may be attached to the base of a cone. Attributes may be assigned to the cap and face of a cone, and also to the entire cone.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a cone is as shown in Figure 1-18.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Caps (optional), bottom cap attribute set (optional), face cap attribute set (optional), attribute set (optional). A cone must have a bottom cap in order to have a bottom cap attribute set. Use `Caps ( Bottom )` to set a cap on the base of a cone.

EXAMPLE

```
Container (
   Cone (
      0 1 0 # orientation
      0 0 1 # major axis
      1 0 0 # minor axis
      0 0 0 # origin
   )
```

```
    Caps ( Bottom )
    Container (
        BottomCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
        )
    )
    Container (
        FaceCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )
        )
    )
)
```

Note
See the note in the section "Cylinders" for an explanation of cap attribute sets. ◆

DEFAULT SIZE

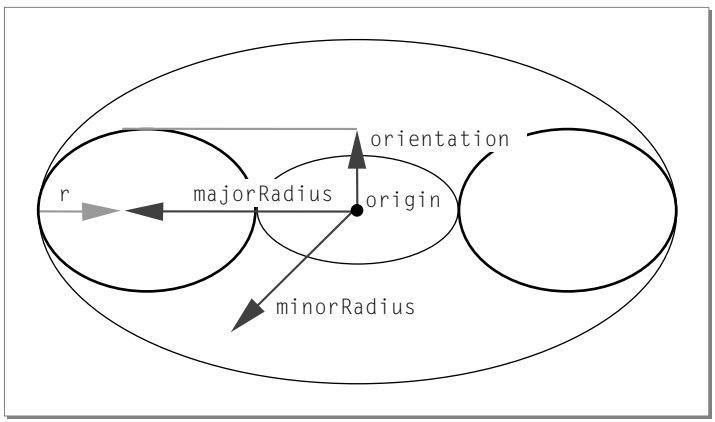
For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
```

Tori

Figure 1-19 shows a torus.

Figure 1-19 A torus



LABELS

ASCII	`Torus`
Binary	`tors (= 0x746F7273)`

DATA FORMAT

```
Vector3D        orientation
Vector3D        majorRadius
Vector3D        minorRadius
Point3D         origin
Float32         ratio
```

Field descriptions

`orientation`	The orientation of the torus. This field specifies the axis of rotation and half-thickness of the torus. The orientation must be orthogonal to both the major and minor radii.
`majorRadius`	The major radius of the torus.
`minorRadius`	The minor radius of the torus.
`origin`	The center of the torus.
`ratio`	The ratio of the length of the major radius of the rotated ellipse to the length of the orientation vector of the torus.

(In Figure 1-19, this is ρ ÷ length(`orientation`.) This field indicates the eccentricity of a vertical cross-section through the torus (wide if $\rho > 1$, narrow if $\rho < 1$).

DATA SIZE

0 or 52

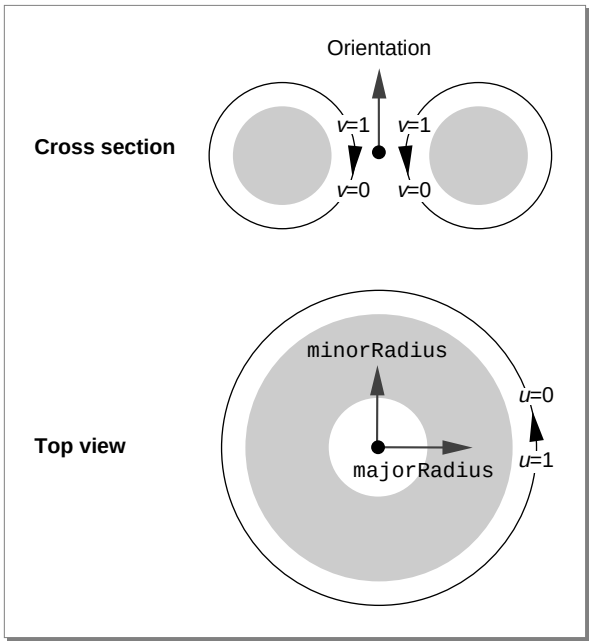
DESCRIPTION

A torus is a three-dinensional object formed by the rotation of an ellipse about an axis in the plane of the ellipse that does not cut the ellipse. The major and minor radii of the torus are the distance of the center of the ellipse from that axis.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a torus is as shown in Figure 1-20.

Figure 1-20 The defalt surface parameterization of a torus



PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLES

```
Container (
    Torus (
        0   .2  0       #orientation
        1   0   0       #majorRadius
        0   0   1       #minorRadius
        0   0   0       #origin
        .5              #ratio
    )
    Container (
        AttributeSet ( )
        DiffuseColor (1 1 0)
    )
)
```

DEFAULT SIZE

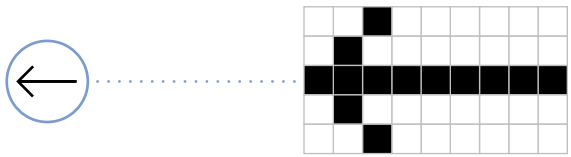
For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
1
```

Markers

Figure 1-21 shows a marker.

Figure 1-21 A marker



LABELS

ASCII	`Marker`
Binary	`mrkr ( = 0x6D726B72 )`

DATA FORMAT

```
Point3D     location
Uns32       width
Uns32       height
Uns32       rowBytes
Int32       xOffset
Int32       yOffset
RawData     data[imageSize]
```

Field descriptions

`location`	The origin of the marker.
`width`	The width of the marker, in pixels. The value of this field must be greater than 0.
`height`	The height of the marker, in pixels. The value of this field must be greater than 0.
`rowBytes`	The number of bytes in a row of the marker.
`xOffset`	The number of pixels, in the horizontal direction, to offset the upper-left corner of the marker from the origin specified in the `location` field.
`yOffset`	The number of pixels, in the vertical direction, to offset the upper-left corner of the marker from the origin specified in the `location` field.

`data[]` This field defines a bitmap that specifies the image to be drawn.

DATA SIZE

32 + (`rowBytes` * `height`) + padding

DESCRIPTION

A marker is a two-dimensional object typically used to indicate the position of an object (or part of an object) in a window. The marker is drawn perpendicular to the viewing vector, aligned with the window, with its origin at the specified location. A marker is always drawn with the same size, shape, and orientation, no matter what transformations are active. However, a transformation may move the origin and thereby affect the position of the marker in the window. Attributes may be assigned only to the entire marker; these attributes apply to those bits in the bitmap that are set to 1.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Container (
       Marker (
          0.5 0.5 0.5                                  # location
```

```
            56                                          # width
            6                                           # height
            7                                           # rowBytes
            -28                                         # xOffset
            -3                                          # yOffset
            0x7E3C3C667E7C18606066666066187C3C
            0x607E7C661860066066607C1860066666
            0x6066007E3C3C667E6618
        )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.8 0.2 0.6 )
        )
    )

Marker (
    0 0 0                       # location
    32                          # width
    32                          # height
    4                           # rowBytes
    -16                         # xOffset
    -16                         # yOffset

    0x001000402167E0201098181011300C08
    0x1E60C6860D403A461880274CB0C041FC
    0x60A0811C608301193080119E30908B38
    0x18604E300CC1CA3037B23C7043181870
    0x0387E82001A01DC000502B4000502A80
    0x00506A80005DD3000076220000484C00
    0x00501800006060000041800000420000
    0x0042000000FF000000FF000000FF0000
)
```

DEFAULT SIZE

None.

Attributes

Diffuse Color

LABELS

ASCII	`DiffuseColor`
Binary	`kdif (= 0x6B646966)`

DATA FORMAT

`ColorRGB`	`diffuseColor`

Field descriptions

`diffuseColor`	A structure having three fields: red, green, blue. The permitted values of these fields are 32-bit floating-point numbers in the closed interval [0, 1], where 0 is the minimum value and 1 is the maximum value.

DATA SIZE

12

DESCRIPTION

Diffuse color is the color of the light of a diffuse reflection (the type of reflection that is characteristic of light reflected from a dull, non-shiny surface). A diffuse color attribute specifies the color of the light diffusely reflected by the objects to which it is assigned.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A diffuse color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   DiffuseColor ( 0 1 0 )
)
```

Specular Color

LABELS

ASCII	`SpecularColor`
Binary	`kspc` (`= 0x6b737063`)

DATA FORMAT

`ColorRGB`	`specularColor`

Field descriptions

`specularColor`	A structure having three fields: red, green, blue. The permitted values of these fields are 32-bit floating-point numbers in the closed interval [0, 1], where 0 is the minimum value and 1 is the maximum value.

DATA SIZE

12

DESCRIPTION

Specular color is the color of the light of a specular reflection (specular reflection is the type of reflection that is characteristic of light reflected from a shiny surface). A specular color attribute specifies the color of the light specularly reflected by the objects to which it is assigned. Note that the diffuse color and specular color assigned to the same object can differ.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A specular color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   DiffuseColor ( .1 .1 .1)      # near-black
   SpecularColor ( 1 1 1 )       # white
)
```

Specular Control

LABELS

ASCII	`SpecularControl`
Binary	`cspc ( = 0x63737063 )`

DATA FORMAT

```
Float32          specularControl
```

Field descriptions

`specularControl` The exponent to be used in computing the intensity of the specular color of one or more objects. The value of this field must be greater than or equal to 0, and is normally an integer greater than or equal to 1.

DATA SIZE

4

DESCRIPTION

A specular control object specifies the specular reflection exponent used in the Phong and related illumination models.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A specular control object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   DiffuseColor ( 1 0 0 )              # red
   SpecularColor ( 1 1 1 )             # white highlights
   SpecularControl ( 60 )              # sharp fall-off
```

```
   )
Ellipsoid( )
)
```

Transparency Color

LABELS

ASCII	`TransparencyColor`
Binary	`kxpr ( = 0x6B787072 )`

DATA FORMAT

`ColorRGB`	`transparency`

Field descriptions

`transparency`	A structure having three fields: red, green, blue. The permitted values of these fields are 32-bit floating-point numbers in the closed interval [0, 1], where 0 is the minimum value and 1 is the maximum value.

DATA SIZE

12

DESCRIPTION

A transparency color attribute affects the amount of color allowed to pass through an object that is not opaque. The transparency color values are multiplied by the color values of obscured objects during pixel color computations. Thus, the transparency color values (1 1 1) indicate complete transparency and the values (0 0 0) indicate complete opacity. The values (0 1 0) indicate that all light in the green color channel is allowed to pass through the foreground object, and no light in the red and blue channels is allowed to pass through the foreground object.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A transparency color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   TransparencyColor ( .5 .5 .5 )
)
```

Surface UV

LABELS

ASCII	`SurfaceUV`
Binary	`sruv ( = 0x73727576 )`

DATA FORMAT

`Param2D`	`surfaceUV`

Field descriptions

`surfaceUV`	The values in the two fields of this structure specify a surface *uv* parameterization for one or more objects. Both of these values must be floating-point numbers greater than or equal to 0 and less than or equal to 1.

DATA SIZE

8

DESCRIPTION

A surface UV object is used to specify a surface *uv* parameterization for one or more objects. A surface UV object is normally used in conjunction with a trim shader.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute set. A Surface UV object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   Mesh ( ... )
   Container (
      VertexAttributeSetList (
         200 Include 4 10 21 22 11
      )
      Container (
         AttributeSet ( )
         SurfaceUV ( 0 0 )
      )
      Container (
         AttributeSet ( )
         SurfaceUV ( 0 1 )
      )
      Container (
```

```
            AttributeSet ( )
            SurfaceUV ( 1 1 )
        )
        Container (
            AttributeSet ( )
            SurfaceUV ( 1 0 )
        )
    )
)
```

Shading UV

LABELS

ASCII	`ShadingUV`
Binary	`shuv` (= `0x73687576`)

DATA FORMAT

`Param2D`	`shadingUV`

Field descriptions

`shadingUV`	The values in the two fields of this structure specify parameters in *u* and *v* for the purpose of shading. Both of these values must be floating-point numbers greater than or equal to 0 and less than or equal to 1.

DATA SIZE

8

DESCRIPTION

A Shading UV object is used to specify *uv* parameters for the purpose of shading. A shading UV object is normally used in conjunction with a texture shader.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute set. A shading UV object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   ShadingUV ( 0 0 )
)
```

Surface Tangents

LABELS

ASCII	`SurfaceTangent`
Binary	`srtn(= 0x7372746E)`

DATA FORMAT

```
Vector3D        paramU
Vector3D        paramV
```

Field descriptions

`paramU` The tangent in the *u* parametric direction.

`paramV` The tangent in the *v* parametric direction.

DATA SIZE

24

DESCRIPTION

A surface tangent object is used to specify three-dimensional tangents to the surface of a geometric object. These tangents serve to indicate the direction of increasing *u* and *v* in the surface parameterization of that object.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute set. A surface tangent always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   SurfaceUV ( 0 0 )
   SurfaceTangent (
      1 0 0
      0 1 0
   )
)
```

Normals

LABELS

ASCII	`Normal`
Binary	`nrml (= 0x6E726D6C )`

DATA FORMAT

`Vector3D` `normal`

Field descriptions

`normal` The surface normal at a vertex. This vector should be normalized.

DATA SIZE

12

DESCRIPTION

The surface normal at a vertex of a verticed object is the average of the normals to the faces of that object sharing that vertex. This normal is obtained by normalizing the relevant face normal vectors, adding those vectors together, and normalizing the result. The surface normal vector is used in Gouraud shading calculations.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A normal always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   Normal ( -1 0 0 )
)
```

Ambient Coefficients

LABELS

ASCII	`AmbientCoefficient`
Binary	`camb (= 0x63616D62 )`

DATA FORMAT

```
Float32      ambientCoefficient
```

Field descriptions

`ambientCoefficient`
The value of this field must lie in the closed interval [0, 1]. 0 is the minimum value, 1 is the maximum value.

DATA SIZE

4

DESCRIPTION

The ambient coefficient is a measure of the level of an object's reflection of ambient light. Ambient coefficients may be assigned separately and selectively to the facets and vertices of faceted and verticed objects, and the same ambient coefficient may be assigned to several objects by placing the coefficient in a suitably located attribute set.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. An ambient coefficient always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AttributeSet ( )
   AmbientCoefficient ( 0.5 )
   DiffuseColor ( 1 1 1 )
)
```

Highlight State

LABELS

ASCII	`HighlightState`
Binary	`hlst` (`= 0x686C7374`)

DATA FORMAT

`Boolean`	`highlighted`

Field descriptions

`highlighted` A value of `True` indicates that affected geometric objects are to receive the highlighting effects specified by an associated highlight style object during rendering. A value of `False` indicates that the affected objects are not to receive those effects.

DATA SIZE

4

DESCRIPTION

A highlight state object is used to specify whether affected geometric objects are to receive highlighting effects during rendering. The relevant highlighting effects are specified by an associated highlight style object. If a geometric object's highlight state is set to `True` (and an associated highlight style object has been defined), then any renderer that supports highlighting will apply the attributes specified by the highlight style object to that geometric object when rendering; these attributes will override incompatible attributes assigned to that geometric object by other means. A highlight state object is idle if no associated highlight style object exists. See the section "Highlight Styles" on page 1-148 for complete details on highlight style objects.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A highlight state object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   Container (
      HighlightStyle ( )              # highlight style object
      Container (
         AttributeSet ( )
         DiffuseColor ( 1 0 0 )       # highlighting: red color
      )
   )
```

```
Container (
    Polygon ( ... )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0 0 1 ) # polygon's normal color: blue
        HighlightState ( True )# polygon is to be highlighted
    )                                   and will appear red when
)                                       rendered
```

Attribute Sets

Attribute Sets

LABELS

ASCII	`AttributeSet`
Binary	`attr (= 0x61747472 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An attribute set is a collection of attributes to be applied to an object, a facet of an object, or a vertex of an object. An attribute set may include attribute objects of as many types as desired, but may include only one attribute object of any particular type. Thus, an attribute set may contain both a diffuse color attribute and a specular color attribute, but may not contain two diffuse color attributes.

Though any attribute object may be included in any attribute set, some attributes cannot sensibly be applied to objects of certain types. For example, a

normal cannot sensibly be applied to an entire view, as encapsulated in a view hints object. An application should disregard such attribute specifications.

Attributes may be assigned to other objects only indirectly, through the use of attribute sets. Attributes are included in an attribute set by placing the attribute objects and the attribute set object together in a container. The attributes in that set may be assigned to a geometric object by placing the relevant container and the geometric object together in a further container.

An attribute set may also be placed in a cap attribute set of any type; in this way, attributes may be assigned separately and selectively to the caps and face of a cone or cylinder. Attribute sets may also be placed in face, geometry, and vertex attribute set lists; in this way, attributes may be assigned separately and selectively to the facets, segments, and vertices of geometric objects having those features. An attribute set may also be placed in a group. Unless overridden, the attributes in an attribute set placed in a hierarchically structured group are inherited by objects at lower levels in the hierarchy of that group. (An application should not permit an attribute to be inherited by an object to which that attribute cannot sensibly be applied.) See the sections on cap attribute sets, attribute set lists, and groups for complete details on the composition of these objects.

PARENT HIERARCHY

Shared, set.

PARENT OBJECTS

Any geometric object, cap attribute set, attribute set list, or group. An attribute set always has a parent object.

CHILD OBJECTS

Attributes: ambient coefficient, diffuse color, specular color, specular control, transparency color, highlight state, shading UV, surface UV (all optional).

EXAMPLE

```
Container (
   Polygon (...)       # all attributes in set applied to polygon
   Container (         # container puts attributes in set
      AttributeSet ( )
      AmbientCoefficient (...)
      DiffuseColor (...)
      SpecularColor (...)
      SpecularControl (...)
      Normal (...)
   )
)
```

Top Cap Attribute Sets

LABELS

ASCII	`TopCapAttributeSet`
Binary	`tcas (= 0x74636173 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A top cap attribute set is used to attach attributes to the top cap of a cylinder that has an optional top cap. The attributes to be assigned to the cap are placed in a regular attribute set in the usual manner. Then the container holding the regular attribute set and the attributes is placed in the cap attribute set by including that container and the cap attribute set in a further container.

The attributes associated with a top cap attribute set are not drawn if the parent object lacks a top cap.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cylinder (always).

CHILD OBJECTS

Attribute set (optional). An empty top cap attribute set has no effect.

EXAMPLE

```
Container (
   Cylindner ( ... )
   Caps ( Top )
   Container (
      TopCapAttributeSet ( )
      Container (
         AttributeSet ( )
         DiffuseColor ( 0.2 0.9 0.4 )
      )
   )
)
```

Bottom Cap Attribute Sets

LABELS

ASCII	`BottomCapAttributeSet`
Binary	`bcas ( = 0x62636173 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A bottom cap attribute set is used to attach attributes to the bottom cap of a cone or cylinder that has an optional bottom cap. The attributes to be assigned to the cap are placed in a regular attribute set in the usual manner. Then the container holding the regular attribute set and the attributes is placed in the cap attribute set by including that container and the cap attribute set in a further container.

The attributes associated with a bottom cap attribute set are not drawn if the parent object lacks a bottom cap.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cone, cylinder (always).

CHILD OBJECTS

Attribute set (optional). An empty bottom cap attribute set has no effect.

EXAMPLE

```
Container (
   Cylinder ( )
   Caps ( Bottom )
   Container (
      BottomCapAttributeSet ( )
      Container (
```

```
            AttributeSet ( )
            DiffuseColor ( 0.2 0.9 0.4 )
        )
    )
)
```

Face Cap Attribute Sets

LABELS

ASCII	`FaceCapAttributeSet`
Binary	`fcas (= 0x66636173 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A face cap attribute set is used to attach an attribute set to the surface of a cone or cylinder but not to its caps. This object is used to apply attributes to a cone or cylinder in a way that does not cause them to be inherited by its caps.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cone, cylinder (always).

CHILD OBJECTS

Attribute set (optional). An empty face cap attribute set has no effect.

EXAMPLE

```
Container (
    Cylinder ( )
    Caps ( Top )
    Container (
        AttributeSet ( )
        SurfaceShader (...)
    )
    Container (
        FaceCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
        )
    )
)
```

Attribute Set Lists

Geometry Attribute Set Lists

LABELS

ASCII	`GeometryAttributeSetList`
Binary	`gasl ( = 0x6761736C )`

DATA FORMAT

```
Uns32           nObjects
PackingEnum     packing
Uns32           nIndices
Uns             indices[nIndices]
```

Field descriptions

`nObjects` The total number of instances of the relevant feature of the parent geometric object possessed by that object. If the parent object is a polyline, the relevant feature is polyline segment, so the value of this field is the total number of segments of the polyline.

`packing` See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of this field.

`nIndices` The size of the following array. See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of this field.

`indices[]` An array of indices. A standard method of indexing instances of the relevant feature of the parent object is assumed to have been established, as with the segments of a polyline. The values of this field are the indices of such instances and are to be specified in increasing order. See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of this field.

DATA SIZE

16 + `nIndices` * `sizeof(Uns)` + padding

DESCRIPTION

A geometry attribute set list is used to assign sets of attributes separately and selectively to distinct instances of a tractable feature of geometric objects. A standard method of indexing the instances of such a feature is presupposed by a geometry attribute set list.

At present, the polyline is the only primitive geometric object to which a geometry attribute set list may be attached. The attribute sets appearing in a geometry attribute set list are assigned to the line segments of which the

polyline is composed, not to the vertices of the polyline. (To attach attributes to the vertices, use a vertex attribute set list.)

The standard index of the segments of a polyline is described in the section "Polylines." To recapitulate, the segment having index *i* is the segment having as its endpoints `vertices[i]` and `vertices[i+1]`.

PARENT HIERARCHY

Data, attribute set list.

PARENT OBJECTS

Polyline (always).

CHILD OBJECTS

Attribute sets (required). See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of how child objects are correlated with instances of the relevant features of the parent geometric object.

EXAMPLE

```
Container (
   PolyLine (...)                      #parent geometric object

   Container (
      GeometryAttributeSetList ( )
      6 exclude 4      # there are 6 segments; exclude 4 of them
      0  2  3  5      # indices of the segments to be excluded
      #child objects
      Container (
         AttributeSet        '         #applied to segment 1
         DiffuseColor (...)
      )
      Container (
         AttributeSet                  #applied to segment 4
         DiffuseColor (...)
```

```
        )
    )
)
```

Face Attribute Set Lists

LABELS

ASCII	`FaceAttributeSetList`
Binary	`fasl (= 0x6661736C )`

PACKING ENUM DATA TYPE

`PackingEnum`

The permitted values are `include` (`= 0x00000000`) and `exclude` (`= 0x00000001`).

DATA FORMAT

```
Uns32           nObjects
PackingEnum     packing
Uns32           nIndices
Uns32           indices[nIndices]
```

Field descriptions

`nObjects` The total number of faces or facets of the parent object. If the parent object is a box, the value of this field is 6. If the parent object is a trigrid, the value of this field is the number of vertices used to define that trigrid, which is also the number of facets of the trigrid. If the parent object is a mesh, the value of this field is the number of faces of that mesh.

`packing` The value of this field determines whether the facets of the parent object of the set list to receive attributes are those whose facet indices appear in the array `indices[]` or are

those whose indices do not appear in that array. A value of `include` indicates the former; `exclude` indicates the latter. You may wish to select `include` if most facets of the parent object are not to receive any attributes. Should any other value appear in this field, the entire set list and all of its child objects should be ignored.

`nIndices` The number of facets of the parent object to which the action specified in the packing field is to be applied; that is, the number of facets to be included in (or excluded from) the group of facets to receive attributes. The value of this field may not exceed that of the `nObjects` field.

`indices[]` An array of facet indices. The values in the fields of this array are the indices of those facets of the parent object to be subject to the action of the value of the packing field, in the event that the number of facets to receive attributes is less than the value in the `nObjects` field. The size of this array must equal the value in the `nIndices` field. Indices are to be entered in fields of this array in increasing order; no index may appear more than once. If the value in the `packing` field is `include`, then the field values represent those facets which are to receive attributes in consequence of the set list. If the value in the `packing` field is `exclude`, then the field values represent those facets that are not to receive attributes in consequence of the set list. If the value in the `packing` field is `exclude` and the value in the `nIndices` field is 0, then this field may be left unspecified; similarly, if the value in the `packing` field is `include` and the value in the `nIndices` field is equal to the value in the `nObjects` field, then this field may be left unspecified.

DATA SIZE

```
16 + nIndices * sizeof(Uns) + padding
```

DESCRIPTION

A face attribute set list is used to assign sets of attributes separately and selectively to one or more facets of a multi-faceted geometric object (that is, to the faces of a box or mesh, or to the triangular facets of a trigrid). A face attribute set list may not be assigned to a general polygon. The listed attribute

sets themselves occur as child objects of the set list object and are correlated with facets of the parent object of the set list as described later in this section. You may think of the child objects as the items in the set list; officially, the set list is the object defined in this section.

For convenience, the `packing` field allows you to choose whether to specify (by inclusion) the facets to receive attributes or to specify (by exclusion) the facets not to receive attributes. The number of child objects you must specify is equal to the number of facets actually to receive attributes, whichever option you select for the `packing` field. You may wish to specify by inclusion rather than by exclusion if most facets are not to receive any attributes. This option can reduce the size of the `indices[]` array, save work, and save disk space.

If the value of the `packing` field of a set list is `include`, then the number of child objects of that set list must equal the value of the `nIndices` field of that set list. If the value of the `packing` field is `exclude`, then the number of child objects must equal the (absolute value of) the difference between the values of the `nObjects` and `nIndices` fields.

Child objects are correlated with facets of the parent object of the set list as follows. Let the child objects of the set list be enumerated in the order of their occurrence in the metafile. If the value of the `packing` field is `include`, then the *i*th child object is correlated with the facet whose index is the value of the *i*th field of the array `indices[]`, or `indices[i-1]`. If the value of the `packing` field is `exclude`, then the *i*th child object is correlated with the facet whose facet index is the *i*th element of the sequence (in increasing order) of facets whose indices do *not* appear in the array `indices[]`. For example, suppose that the parent object is a mesh having 17 faces, `packing` is set to `exclude`, `nIndices` is 11, and the elements of `indices[]` are 1, 2, 4, 6, 7, 8, 11, 12, 13, 14, 16. Then six facets are to receive attributes: facets 0, 3, 5, 9, 10, 15, so the set list will have six child objects c_0,..., c_5. The third child object (that is, c_2) is correlated with facet 5, and, in general, the *i*th element of the sequence $<c_0,..., c_5>$ is correlated with the *i*th element of the sequence <0, 3, 5, 9, 10, 15>.

The index used to enumerate the facets of a multifaceted geometric object is described in the section pertaining to that object. Indices begin with zero, so that the index of the *i*+1st facet of a multifaceted object is *i*. The index used to construct an attribute set list must be standard.

PARENT HIERARCHY

Data, attribute set list.

PARENT OBJECTS

Box, mesh, trigrid.

CHILD OBJECTS

Attribute sets (required). The number of child objects is determined in the manner indicated in the description of a face attribute set list.

EXAMPLE

```
Container (
    TriGrid (...)                          #parent object

    Container (
        FaceAttributeSetList ( )
        6                       #nObjects          (parent has six facets;
        exclude                 #packing           (exclude
        4                       #nIndices          (four of them:
        0   2   3   5           #indices[]         (these four.)
        #begin list
        Container (
            AttributeSet                           #apply to facet 1
            DiffuseColor (...)
        )
        Container (
            AttributeSet                           #apply to facet 4
            DiffuseColor (...)
        #end list
        )
    )
)
```

Vertex Attribute Set Lists

LABELS

ASCII	`VertexAttributeSetList`
Binary	`vasl (= 0x7661736C )`

DATA FORMAT

```
Uns32           nObjects
PackingEnum     packing
Uns32           nIndices
Uns             indices[nIndices]
```

Field descriptions

`nObjects`	The number of vertices of the parent geometric object.
`packing`	See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of this field.
`nIndices`	Size of the following array. See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of this field.
`indices[]`	An array of vertex indices. See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of this field.

DATA SIZE

```
16 + nIndices * sizeof(Uns) + padding
```

DESCRIPTION

A vertex attribute set list is used to assign sets of attributes separately and selectively to the vertices of a verticed geometric object. Among the primitive metafile geometric objects, the following have vertices: general polygons, lines, meshes, polygons, polylines, triangles, and trigrids.

The index used to enumerate the vertices of an object of one of these types is described in the section on objects of that type. To recapitulate, in all cases the

vertices are enumerated in the order of their occurrence in the specification of the parent geometric object. In the case of a general polygon, the index does not distinguish between contours.

PARENT HIERARCHY

Data, attribute set list.

PARENT OBJECTS

General polygon, line, mesh, polygon, polyline, triangle, trigrid. A vertex attribute set list always has a parent object.

CHILD OBJECTS

Attribute sets (required). See the section "Face Attribute Set Lists" on page 1-137 for a complete explanation of how child objects are correlated with aspects of the parent geometric object.

EXAMPLE

```
Container (
   GeneralPolygon (          # parent geometric object
      2                      # nContours
      #contour 0
      3                      # nVertices, contour 0
      -1 0 0                 # vertex 0
      1 0 0                  # vertex 1
      0 1.7 0                # vertex 2
      #contour 1
      3                      # nVertices, contour 1
      -1 0.4 0               # vertex 3
      1 0.4 0                # vertex 4
      0 2.1 0                # vertex 5
   )
   Container (
      VertexAttributeSetList ( 6 Exclude 2 0 4 ) # set list
      Container (                                # child objects
```

```
            AttributeSet ( )              # vertex 1 (contour 0)
            DiffuseColor ( 0 0 1 )
         )
         Container (
            AttributeSet ( )              # vertex 2 (contour 0)
            DiffuseColor ( 0 1 1 )
         )
         Container (
            AttributeSet ( )              # vertex 3 (contour 1)
            DiffuseColor ( 1 0 1 )
         )
         Container (
            AttributeSet ( )              # vertex 5 (contour 1)
            DiffuseColor ( 1 1 0 )
         )
      )
      Container (
         AttributeSet ( )
         DiffuseColor ( 1 1 1 )
      )
)
```

Styles

Backfacing Styles

LABELS

ASCII	`BackfacingStyle`
Binary	`bckf ( = 0x62636B66 )`

BACKFACING STYLES

```
Both            0x00000000
Culled          0x00000001
Flipped         0x00000002
```

Constant descriptions

`Both` A renderer should draw shapes that face toward and away from the camera. If a shape has only frontfacing attributes, those attributes are used for both sides of the shape.

`Culled` A renderer should not draw shapes that face away from the camera (this is not the same as hidden surface removal).

`Flipped` A renderer should draw shapes that face toward and away from the camera. If a shape has only frontfacing attributes, those attributes are used for both sides of the shape, but the normals of backfacing shapes are inverted, so that they face toward the camera.

DATA FORMAT

```
BackfacingEnum              backfacing
```

`backfacing` The value in this field must be one of the three constants defined above.

DESCRIPTION

A scene's backfacing style determines whether or not a renderer draws shapes that face away from a scene's camera. This style object defines some of the characteristics of a renderer and generally applies to all of the objects in a model.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( OrderedDisplayGroup ( ) )
   Matrix ( ... )
   BackfacingStyle ( Both )
   Mesh ( ... )
   Mesh ( ... )
EndGroup ( )
```

Interpolation Styles

LABELS

ASCII	`InterpolationStyle`
Binary	`intp (= 0x696E7470)`

INTERPOLATION STYLES

`None`	`0x00000000`
`Vertex`	`0x00000001`
`Pixel`	`0x00000002`

Constant descriptions

`None`	No interpolation is to occur. The renderer is to apply each effect uniformly across a surface.
`Vertex`	The renderer is to interpolate values linearly across a verticed surface, using the values at the vertices.
`Pixel`	The renderer is to calculate a value of each effect for every pixel in the image.

DATA FORMAT

```
InterpolationStyleEnum        interpolationStyle
```

Field descriptions

`interpolationStyle`
The value in this field must be one of these constants: `None`, `Vertex`, or `Pixel`.

DATA SIZE

4

DESCRIPTION

A scene's interpolation style determines the method of interpolation a renderer uses when applying lighting or other shading effects to a surface. A value of `None` causes the surfaces of a model to have a faceted appearance; the other two values cause its surfaces to be rendered smoothly.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
   InterpolationStyle ( Vertex )
   Container (
      Triangle ( ... )
      VertexAttributeSetList ( ... )
```

```
        .
        .
        .
    )
EndGroup ( )
```

Fill Styles

LABELS

ASCII	`FillStyle`
Binary	`fist (= 0x66697374 )`

FILL STYLES

`Filled`	`0x00000000`
`Edges`	`0x00000001`
`Points`	`0x00000002`
`Empty`	`0x00000003`

Constant descriptions

`Filled`	The renderer should draw shapes as solid filled objects.
`Edges`	The renderer should draw shapes as the sets of lines that define the edges of surfaces.
`Points`	The renderer should draw shapes as the sets of points that define the vertices of surfaces.
`Empty`	[To be supplied.]

DATA FORMAT

`FillStyleEnum`	`fillStyle`

Field descriptions

`fillStyle`	The value of this field must be one of these constants: `Filled`, `Edges`, `Points`, `Empty`.

DATA SIZE

4

DESCRIPTION

A scene's fill style determines whether an object is drawn as a solid filled object or is decomposed into a set of edges or points.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
   FillStyle ( Edges )
   Container (
      Mesh ( ... )
      VertexAttributeSetList ( ... )
   )
   Torus ( ... )
EndGroup( )
```

Highlight Styles

LABELS

ASCII `HighlightStyle`

Binary `high (= 0x68696768 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A highlight style object is used to specify attributes to be applied to selected geometric objects during rendering. Any renderer that supports highlighting will use the attributes specified by a highlight style object to override incompatible attributes assigned to affected geometric objects in other ways. The attributes specified by a highlight style object are applied to a geometric object only if that geometric object also has a highlight state attribute that is set to `True`. See the section "Highlight State" on page 1-126 for complete details on highlight state attributes.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (required).

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
   Container (
      HighlightStyle ( )              # highlight style object
```

```
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )        # highlight attribute
        )
    Container (
        Polygon ( ... )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
            HighlightState ( True )# polygon will be highlighted
        )
    )
    Container (
        Box
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 1 0 )
            HighlightState ( False )# box will not be highlighted
        )
    )
    Container (
        Line ( ... )              # line will not be highlighted
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 1 1 )
        )
    )
)
EndGroup ( )
```

Subdivision Styles

LABELS

ASCII	`SubdivisionStyle`
Binary	`sbdv (= 0x7364636C )`

SUBDIVISION METHOD ENUM DATA TYPE

`Constant`	`0x00000000`
`WorldSpace`	`0x00000001`
`ScreenSpace`	`0x00000002`

Note
There are two data formats. ◆

FIRST DATA FORMAT

`SubdivisionMethodEnum`	`subdivisionMethod`
`Float32`	`value1`

Field descriptions

`subdivisionMethod`
The value in this field must be one of the specifiers `WorldSpace` or `ScreenSpace`. A value of `WorldSpace` indicates that the renderer subdivides a curve (or surface) into polylines (or polygons) whose sides have a world-space length that is at most as large as the value specified in the `value1` field. A value of `ScreenSpace` indicates that the renderer subdivides a curve (or surface) into polylines (or polygons) whose sides have a length that is at most as large as the number of pixels specified in the `value1` field.

`value1`
For world-space subdivision, the maximum length of a polyline segment (or polygon side) into which a curve (or surface) is subdivided. For screen-space subdivision, the maximum number of pixels in a polyline segment (or polygon side) into which a curve (or surface) is subdivided. The value in this field should be greater than 0.

DATA SIZE

8

SECOND DATA FORMAT

```
SubdivisionMethodEnum        subdivisionMethod
Uns32                        value1
Uns32                        value2
```

Field descriptions

`subdivisionMethod`
The value in this field must be the specifier `Constant`. This value indicates that the renderer subdivides a curve into a number of polyline segments and a surface into a mesh of polygons.

`value1` The number of polylines into which a curve should be subdivided, or the number of vertices in the *u* parametric direction of the polygonal mesh into which a surface is divided. The value in this field should be greater than 0.

`value2` The number of vertices in the *v* parametric direction of the polygonal mesh into which a surface is divided. The value in this field should be greater than 0.

DATA SIZE

12

DESCRIPTION

A scene's subdivision style determines how a renderer decomposes smooth curves and surfaces into polylines and polygonal meshes for display purposes. Different specifiers and numerical values determine different degrees of fineness of approximation.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( )
   SubdivisionStyle ( Constant 32 32 )
   Ellipsoid ( ... )
   )
   Container (
      SubdivisionStyle ( WorldSpace 12 )
      Box ( ... )
   )
EndGroup ( )
```

Orientation Styles

LABELS

ASCII	`OrientationStyle`
Binary	`ornt (= 0x6F726E74 )`

ORIENTATION STYLES

`CounterClockwise`	`0x00000000`
`Clockwise`	`0x00000001`

Constant descriptions

`CounterClockwise`
The front face of a polygonal shape is defined using the righthand rule.

`Clockwise`
The front face of a polygonal shape is defined using the lefthand rule.

DATA FORMAT

```
OrientationEnum                 orientation
```

Field descriptions

`orientation` The value in this field must be one of these constants: `CounterClockwise`, `Clockwise`.

DATA SIZE

4

DESCRIPTION

A scene's orientation style determines which side of a planar surface is considered (by the renderer) to be the "front" side. This style may be changed in order to change the orientation of a polygonal shape.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup( DisplayGroup ( ) )
   OrientationStyle ( Clockwise )
   .
   .
   .
EndGroup ( )
```

Receive Shadows Styles

LABELS

ASCII	`ReceiveShadowsStyle`
Binary	`rcsh` (`= 0x72637368`)

DATA FORMAT

`Boolean`	`receiveShadows`

Field descriptions

`receiveShadows`	A value of `True` indicates that objects are to receive shadows; a value of `False` indicates that objects are not to receive shadows.

DATA SIZE

4

DESCRIPTION

A scene's receive shadows style specifies whether or not obscured objects shall receive shadows in rendering.

Note
Some lights also specify whether or not the objects they illuminate shall cast shadows. However, objects in the scope of a receive shadows style set to `False` do not receive shadows, whether or not they are also appropriately situated to receive shadows from a light set to cast shadows. ◆

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
   PointLight ( ... )
   ReceiveShadows ( True )
   Mesh ( ... )
   Mesh ( ... )
   Mesh ( ... )
EndGroup ( )
```

Pick ID Styles

LABELS

ASCII	`PickIDStyle`
Binary	`pkid (= 0x706B6964 )`

DATA FORMAT

```
Uns32        id
```

Field descriptions

`id`	An integer, supplied by your application.

DATA SIZE

4

DESCRIPTION

A pick ID style object is used to correlate the class of objects within its scope with an integer. This integer may be included in the specification of a picking operation to restrict that operation to the objects in that class. A pick ID style object must be placed in a group or container to have effect; the scope of a pick ID style object placed in a group (or container) is the class of objects located between that style object and the end of that group (or container).

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
PickIDStyle ( 8 )
```

Pick Parts Styles

LABELS

ASCII	`PickPartsStyle`
Binary	`pkpt (= 0x706B7074 )`

PICK PARTS STYLES

`Object`	`0x00000000`
`Face`	`0x00000001`
`Edge`	`0x00000002`

```
Vertex          0x00000003
```

Constant descriptions

`Object`	The hit list for picking contains only whole objects.
`Face`	The hit list for picking contains faces of objects.
`Edge`	The hit list for picking contains edges of objects.
`Vertex`	The hit list for picking contains vertices of objects.

DATA FORMAT

```
PickPartsFlags      pickParts
```

Field descriptions

`pickParts`	The value in this field must be one of the four flags specified in the `PickPartsFlags` data enumeration.

DATA SIZE

4

DESCRIPTION

A pick parts style object is used to specify the sort of object to be picked during the operation of picking. The flags `Face`, `Edge`, and `Vertex` are used to pick meshes; the flag `Object` is used to pick all other objects. The default flag is `Object`.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
PickPartsStyle ( Object )
```

Transforms

Translate Transforms

LABELS

ASCII	`Translate`
Binary	`trns (= 0x74726E73 )`

DATA FORMAT

`Vector3D`	`translate`

Field descriptions

`translate`	A translation in three dimensions, specified by a vector.

DATA SIZE

12

DESCRIPTION

A translate transform moves an object along the *x*, *y*, and *z* axes by the values specified by its translation vector. Thus, the transform `Translate ( i j k )` moves each point P = <P_x, P_y, P_z> in its scope to the point P′ = <P_{x+i}, P_{y+j}, P_{z+k}>.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Translate ( -1 1 0 )
```

Scale Transforms

LABELS

ASCII	`Scale`
Binary	`scal( = 0x7363616C )`

DATA FORMAT

`Vector3D`	`scale`

Field descriptions

`scale`	A scaling vector.

DATA SIZE

12

DESCRIPTION

A scale transform scales an object along the *x*, *y*, and *z* axes by the values specified by its scaling vector.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Scale ( 2 2 2 )
```

Matrix Transforms

LABELS

ASCII	`Matrix`
Binary	`mtrx ( = 0x6D747278 )`

DATA FORMAT

`Matrix4x4`	`matrix`

Field descriptions

`matrix`	A 4-by-4 array specifying a custom matrix transformation. The specified matrix should be invertible.

DATA SIZE

64

DESCRIPTION

A matrix transform is a transform by an arbitrary invertible 4-by-4 matrix.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Matrix (
   1 0 0 0
   0 1 0 0
   0 0 1 0
   0 0 0 1
)
```

Rotate Transforms

LABELS

ASCII	`Rotate`
Binary	`rott (= 0x726F7474)`

AXIS ENUM DATA TYPE

`X`	`0x00000000`
`Y`	`0x00000001`
`Z`	`0x00000002`

DATA FORMAT

```
AxisEnum        axis
Float32         radians
```

Field descriptions

`axis` The axis of rotation. The value in this field must be one of these constants: `X`, `Y`, or `Z`.

`radians` The number of radians to rotate around the axis of rotation.

DATA SIZE

8

DESCRIPTION

A rotate transform rotates an object about the *x*, *y*, or *z* axis by a specified number of radians.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Rotate (            # rotate about the z axis by -1.57 radians
   Z
   -1.57
   )
```

Rotate-About-Point Transforms

LABELS

ASCII	`RotateAboutPoint`
Binary	`rtap ( = 0x72746170 )`

DATA FORMAT

`AxisEnum`	`axis`
`Float32`	`radians`
`Point3D`	`origin`

Field descriptions

`axis`	The axis of rotation.
`radians`	The number of radians to rotate about the axis of rotation.
`origin`	The point at which the rotation is to occur.

DATA SIZE

20

DESCRIPTION

A rotate-about-point transform rotates an affected object by the specified number of radians about the line parallel to the value in the `axis` field and passing through the point specified in the `origin` field.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Rotate (
   Y                    # axis
   1.0                  # radians
   2 3 4                # origin
)
```

Rotate-About-Axis Transforms

LABELS

ASCII	`RotateAboutAxis`
Binary	`rtaa ( = 0x72746161 )`

DATA FORMAT

`Point3D`	`origin`
`Vector3D`	`orientation`
`Float32`	`radians`

Field descriptions

`origin`	The origin of the axis of rotation.
`orientation`	The orientation of the axis of rotation. This vector should be normalized.
`radians`	The number of radians by which an affected object is rotated.

DATA SIZE

28

DESCRIPTION

A rotate-about-axis transform rotates an object about an arbitrary axis in space by a specified number of radians. The value in the `origin` field and the orientation vector are used to define the axis of rotation.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
RotateAboutAxis (
   20 0 0              # origin
   .33 .33 .34         # orientation
   1.57                # radians
)
```

Quaternion Transforms

LABELS

ASCII	`Quaternion`
Binary	`qtrn(= 0x7174726E)`

DATA FORMAT

```
Float32         w
Float32         x
Float32         y
Float32         z
```

Field descriptions

`w`	The *w* component of the quaternion transform.
`x`	The *x* component of the quaternion transform.
`y`	The *y* component of the quaternion transform.
`z`	The *z* component of the quaternion transform.

DATA SIZE

16

DESCRIPTION

A quaternion transform rotates and twists an object in a manner determined by the mathematical properties of quaternions.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Quaternion ( 0.2 0.7 0.2 1.57 )
```

Shader Transforms

LABELS

ASCII	`ShaderTransform`
Binary	`sdxf (= 0x73647866)`

DATA FORMAT

`Matrix4x4`	`shaderTransform`

Field descriptions

`shaderTransform`	A 4-by-4 matrix.

DATA SIZE

64

DESCRIPTION

A shader transform transforms a shaded object into a distinct world-space coordinate system. A shader transform does not affect the current transformation hierarchy and does not affect the manner in which the object to which it is applied is drawn.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Any shader. A shader transform always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   CustomShader ( )
   ShaderTransform (
      1 0 0 0
      0 1 0 0
      0 0 1 0
      2 3 4 1
   )
)
```

Shader UV Transforms

LABELS

ASCII	`ShaderUVTransform`
Binary	`sduv ( = 0x73647576 )`

DATA FORMAT

`Matrix3x3`	`matrix`

Field descriptions

`matrix`	A 3-by-3 matrix.

DATA SIZE

36

DESCRIPTION

A shader *uv* transform may be used to transform the surface *uv* parameterization of a geometric object prior to shading. A shader *uv* transform may be used to rotate a texture map.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Any shader. A shader *uv* transform always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   TextureShader ( )
   ShaderUVTransform (
      1 0 0
      0 1 0
      0.2 0.3 1
   )
   PixmapTexture ( ... )
)
```

Lights

Attenuation and Fall-Off Values

Some lights suffer attenuation; that is, a loss of intensity over distance. The application determines the degree of attenuation of a light by specifying substituends for three distinct variables in a complex term that occurs in whatever formula it uses to compute the intensity of that light at a given distance from its source. The choice of constants determines whether the light suffers attenuation and, if so, the degree to which its intensity diminishes as a function of distance. These constants are specified in a data structure of the following type.

ATTENUATION DATATYPE

```
Float32         c0
Float32         c1
Float32         c2
```

DESCRIPTION

The attenuation factor determined by an attenuation data type is expressed by the result of replacing the variables c_0, c_1, c_2 by the values of the fields `c0`, `c1`, `c2` in the complex term

$$\left(\frac{1}{c_0 + c_1 d_{l,\,p} + c_2 d_{l,\,p}^{\,2}} \right)$$

(Here l is the location of the light source, p is the illuminated point, and $d_{l,\,p}$ is the distance from l to p.)

The initial intensity of a light is multiplied by its attenuation factor when the intensity of the light at a point is computed. Thus, if $c_0 = 1$ and $c_1 = c_2 = 0$, then the light does not suffer attenuation over distance. If $c_1 = 1$ and $c_0 = c_2 = 0$, then the intensity of the light at a point p diminishes in proportion to the distance between p and the light source, provided that that distance is at least one unit. If $c_2 = 1$ and $c_0 = c_1 = 0$, then the intensity of the light at p diminishes in proportion to the square of the distance between p and the light source, again provided that that distance is at least one unit. If $c_0 = c_2 = 1$ and $c_1 = 0$, then the intensity of the light at p diminishes in proportion to the sum of 1 and the square of the distance between p and the light source.

The attenuation factor is not clamped to a maximum value. Thus, for some choices of c_0, c_1, c_2, the intensity of a light may exceed its source intensity at distances of less than one unit, driving the RGB color values of the light toward the maximum of (1, 1, 1), or pure white.

The amount of illumination that a point illuminated by a light receives from that light also depends on several other factors. Among these factors are the diffuse and specular reflection characteristics of the surface that contains that point and the relative positions of the light source, the illuminated point, and the viewer (the camera).

LIGHT FALL-OFF VALUES

A spot light specifies a cone of light emanating from a source location. Within the inner cone defined by the hot angle of a spot light, the light may suffer attenuation over distance from the light source. Within the outer section of the cone between the hot angle and the outer angle of a spot light, the light may suffer further attenuation.

Spot lights have a fall-off value that determines the manner of attenuation of the light from the edge of the cone defined by the hot angle to the edge of the cone defined by the outer angle. The direction of fall off is perpendicular to the ray from the source location through the center of the cone. The amount of additional attenuation determined by any fall-off value is the same along all rays from the location of the light source forming the same angle with the axis of the cone.

The following constants specify four fall-off values a spot light may have.

FALLOFF VALUES

```
None            0x00000000
Linear          0x00000001
Exponential     0x00000002
Cosine          0x00000003
```

Constant descriptions

`None` The intensity of the light is not affected by the distance from the center of the cone to the edge of the cone.

`Linear` The intensity of the light at the edge of the cone falls off at a constant rate from the intensity of the light at the center of the cone.

`Exponential` The intensity of the light at the edge of the cone falls off exponentially from the intensity of the light at the center of the cone.

`Cosine` The intensity of the light at the edge of the cone falls off as the cosine of the outer angle from the intensity of the light at the center of the cone.

Light Data

LABELS

ASCII	`LightData`
Binary	`lght ( = 0x6C676874 )`

DATA FORMAT

`Boolean`	`isOn`
`Float32`	`intensity`
`ColorRGB`	`color`

Field descriptions

`isOn`	A value of `True` indicates that the parent light is active (is on). A value of `False` indicates that the parent light is inactive (is off).
`intensity`	The intensity of the parent light at its source. The value in this field must be in the closed interval [0, 1]. 0 is the minimum value; 1 is the maximum value.
`color`	The RGB color of the parent light.

DATA SIZE

20

DESCRIPTION

A light data object specifies the color and source intensity of a parent light, and whether that light is currently active or inactive. A light object that does not have a light data object as a child object should be given the default values indicated below.

Note
A value of less than 1.0 in the intensity field of a light data object affects the color of the parent light. ◆

PARENT HIERARCHY

Data.

PARENT OBJECTS

A light data object always has a parent object; the parent object is always a light object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   AmbientLight ( )
   LightData (
      True                    # is on
      0.75                    # intensity
      0.7 0.3 0.4             # color
   )
)
```

DEFAULT SETTING

```
True          # isOn
1.0           # intensity (full)
1 1 1         # color (white)
```

Ambient Light

LABELS

ASCII	`AmbientLight`
Binary	`ambn ( = 0x616D626E )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

Ambient light is a base amount of light that is added to the illumination of all surfaces in a scene. Ambient light has no apparent source or location; its intensity is constant, and it does not cast shadows.

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data (optional). If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
Container (
   ViewHints ( )
   .
   .
   .
   BeginGroup ( DisplayGroup ( ) )
      Container (
         AmbientLight ( )
         LightData ( ... )
      )
```

```
        Container (
            DirectionalLight( )
            LightData ( ... )
        )
    EndGroup ( )
)
```

Directional Lights

LABELS

ASCII	`DirectionalLight`
Binary	`drct(= 0x64726374)`

DATA FORMAT

`Vector3D`	`direction`
`Boolean`	`castsShadows`

Field descriptions

`direction` The direction of the directional light. This vector should be normalized.

`castsShadows` A value of `True` indicates that the light casts shadows; a value of `False` indicates that the light does not cast shadows.

DATA SIZE

16

DESCRIPTION

A directional light is a light that emits parallel rays in a specific direction. A directional light may be set to cast shadows.

Note
Some style objects also specify whether or not objects in a scene shall receive shadows. However, objects in the scope of a receive shadows style set to `False` do not receive shadows, whether or not they are also appropriately situated to receive shadows from a light set to cast shadows. ◆

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data (optional). If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
Container (
   DirectionalLight ( 1 0 0 True )
   LightData ( ... )
)
```

Point Lights

LABELS

ASCII	`PointLight`
Binary	`pntl( = 0x706E746C )`

DATA FORMAT

```
Point3D          location
Attenuation      attenuation
Boolean          castsShadows
```

Field descriptions

`location` The location of the source of the point light.

`attenuation` This structure determines the amount that the intensity of the light diminishes over distance. See the section "Attenuation and Fall-Off Values" on page 1-170 for a description of this structure.

`castsShadows` A value of `True` specifies that objects illuminated by the light are to cast shadows; a value of `False` specifies that objects illuminated by the light are not to cast shadows.

DATA SIZE

20

DESCRIPTION

A point light is a light that emits rays in all directions from a specific point source. A point light may suffer attenuation over distance, and may cast shadows.

Note
Some style objects also specify whether or not objects in a scene shall receive shadows. However, objects in the scope of a receive shadows style set to `False` do not receive shadows, whether or not they are also appropriately situated to receive shadows from a light set to cast shadows. ◆

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data (optional). If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
   Triangle ( ... )
   Box ( ... )
   Container (
      PointLight (
         -10, 1, -1              # location
         1 0 1                   # attenuation
         True                    # casts shadows
      )
      LightData ( ... )
   )
EndGroup ( )
```

Spot Lights

LABELS

ASCII	`SpotLight`
Binary	`spot ( = 0x73706F74 )`

DATA FORMAT

`Point3D`	`location`
`Vector3D`	`orientation`
`Boolean`	`castsShadows`
`Attenuation`	`attenuation`

```
Float32         hotAngle
Float32         outerAngle
FallOffEnum     fallOff
```

Field descriptions

`location` The location of the source of the spot light.

`orientation` The orientation of the cone of light emitted by the spot light. The direction of this vector is toward the light source. This vector should be normalized.

`castsShadows` A value of `True` specifies that objects illuminated by the light are to cast shadows; a value of `False` indicates that objects illuminated by the light are not to cast shadows.

`attenuation` This structure determines the amount that the intensity of the light diminishes over distance. See the section "Attenuation and Fall-Off Values" on page 1-170 for a description of this structure.

`hotAngle` The half-angle (specified in radians) from the center of the cone of light within which the light remains at constant full intensity. The value in this field should be in the half-open interval [0, $\pi/2$).

`outerAngle` The half-angle (specified in radians) from the center to the edge of the cone of the spot light. The value in this field should be in the half-open interval [0, $\pi/2$), and should not be less than the value in the `hotAngle` field.

`fallOff` The fall-off value for the spot light. The value in this field determines the manner of attenuation of the light from the edge of the hot angle to the edge of the outer angle. See the section "Attenuation and Fall-Off Values" on page 1-170 for a description of the constants that can be used in this field.

DATA SIZE

44

DESCRIPTION

A spot light is a light source that emits a circular cone of light in a specific direction from a specific location. Every spot light has a hot angle and an outer

angle that together define the shape of the cone of light and the amount of attenuation that occurs from the center of the cone to the edge of the cone. The attenuation of the light's intensity from the edge of the hot angle to the edge of the outer angle is determined by the light's fall-off value.

Note
Some style objects also specify whether or not objects in a scene shall receive shadows. Thus, conflicting shadowing instructions can be sent to a renderer. The outcome in such a case is renderer specific, application specific, or both. ◆

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data. If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
Container (
   SpotLight (
      0 9 0                  # location
      0 1 0                  # orientation
      True                   # castsShadows
      0 0 1                  # attenuation
      0.3                    # hotAngle
      0.5                    # outerAngle
      Linear                 # fallOff
   )
   LightData ( ... )
)
```

Cameras

Camera Placement

LABELS

ASCII	`CameraPlacement`
Binary	`cmpl (= 0x636D706C)`

DATA FORMAT

`Point3D`	`location`
`Point3D`	`pointOfInterest`
`Vector3D`	`upVector`

Field descriptions

`location`	The location (in world-space coordinates) of the eye point of the parent camera.
`pointOfInterest`	The point at which the parent camera is aimed, in world-space coordinates.
`upVector`	The up vector of the parent camera. This vector should be perpendicular to the viewing direction defined by the values in the `location` and `pointOfInterest` fields. This vector should be normalized.

DATA SIZE

36

DESCRIPTION

A camera placement object defines the location, point of interest, and orientation of its parent camera, in world-space coordinates. The camera vector (also called the view vector) is defined to be the vector `pointOfInterest - location`. This vector is normal to the projection plane and to the clipping

planes, and the distances from the camera to those planes are measured along this vector.

A camera placement object determines the coordinate system of the projection plane as follows. The origin of the projection plane is the point at the intersection of the projection plane and the line through the location and point of interest. The *y* axis of the projection plane coincides with the projection onto the projection plane of the up vector, and the *x* axis of the projection plane is the axis such that it, the *y* axis of the projection plane, and the inverse of the camera vector form a righthanded coordinate system.

If no camera placement object is specified for a camera, that camera should receive the default camera placement values specified below.

PARENT HIERARCHY

Data.

PARENT OBJECTS

View angle aspect camera, view plane camera, orthographic camera. A camera placement object always has a camera as a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   ViewAngleAspectCamera ( ... )
   CameraPlacement (
      0 0 30                # location on z axis
      0 0 0                 # point of interest is the origin
      0 1 0                 # up vector aligned with yaxis
   )
)
```

DEFAULT VALUES

```
0 0 1        # location
0 0 0        # pointOfInterest
0 1 0        # upVector
```

Camera Range

LABELS

ASCII	`CameraRange`
Binary	`cmrg ( = 0x636D7267 )`

DATA FORMAT

```
Float32          hither
Float32          yon
```

Field descriptions

`hither` The distance from the location of the parent camera to the near clipping plane. The value in this field should be greater than 0.

`yon` The distance from the location of the parent camera to the far clipping plane. The value in this field should be greater than the value in the `hither` field.

DATA SIZE

8

DESCRIPTION

A camera range object is used to set the near and far clipping planes of its parent camera. Distances are measured in the direction defined by the camera vector, which is normal to both clipping planes.

PARENT HIERARCHY

Data.

PARENT OBJECTS

View angle aspect camera, view plane camera, orthographic camera. A camera placement object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   ViewPlaneCamera ( ... )
   CameraPlacement ( ... )
   CameraRange (
      .01                          # hither
      75                           # yon
   )
)
```

Camera Viewport

LABELS

ASCII	`CameraViewPort`
Binary	`cmvp ( = 0x636D7670 )`

DATA FORMAT

```
Point2D         origin
Float32         width
Float32         height
```

Field descriptions

`origin`	The origin of the view port of the parent camera. The abscissa and ordinate of this point should lie in the closed interval [–1, 1]. The value in this field is the upper-left corner of the view port.
`width`	The width of the view port of the parent camera. The value in this field should lie in the half-open interval (0, 2], and should not be greater than the absolute value of the difference between 1 and the abscissa of the origin.
`height`	The height of the view port of the parent camera. The value in this field should lie in the half-open interval (0, 2], and should not be greater than the difference between –1 and the ordinate of the origin.

DATA SIZE

16

DESCRIPTION

Every camera specifies the dimensions of the largest (rectangular) image that that camera can produce (called the *parent image*), either explicitly or implicitly. The parent image may be specified by giving the coordinates of its vertices, by giving the height to width ratio of its sides, or in some other fashion. The camera view port object specifies the subregion of the parent image that is actually to be drawn. The value in the `origin` field defines the upper left corner of the view port; the values in the other two fields determine the lengths of the sides of the view port.

The default setting specified below sets the view port equal to the parent image. Other settings may be used to clip the parent image to desired specifications.

Camera view port specifications are made in a coordinate system in which the height-to-width ratio of the parent image is one to one, and the coordinates of the upper-left and lower-right corners of that image are (–1, 1) and (1, –1), respectively. The actual height-to-width ratio of the parent image may not be one to one. If not, then view port specifications should be made under the assumption that the view port will be rescaled by the inverse of the height-to-width ratio of the parent image after the view port specifications have been made. Thus, if the height-to-width ratio of the parent image is *i*/*j*,

and the height-to-width ratio of the image actually to be drawn is *i'/j'*, then the height-to-width ratio of the rectangle specified in the view port should be *i'j/ij'*. Any view port having a different height-to-width ratio will result in a distorted image.

PARENT HIERARCHY

Data.

PARENT OBJECTS

View angle aspect camera, view plane camera, orthographic camera. A camera viewport object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
CameraViewPort (
   -0.5 0.5
   1.0
   1.0
)
```

DEFAULT VALUES

```
-1 1          # origin at upper left corner of the parent image
2             # width is the entire width of the parent image
2             # height is the entire height of the parent image
```

Orthographic Cameras

LABELS

ASCII	`OrthographicCamera`
Binary	`orth (= 0x6F727468)`

DATA FORMAT

```
Float32         left
Float32         top
Float32         right
Float32         bottom
```

Field descriptions

`left` The *x* coordinate (in the camera's coordinate system) of the upper left corner of the front face of the view volume. Or, the distance from the center of the camera lens (that is, the view rectangle) to the left side of the lens.

`top` The *y* coordinate (in the camera's coordinate system) of the upper left corner of the front face of the view volume. Or, the distance from the center of the camera lens (that is, the view rectangle) to the top side of the lens.

`right` The *x* coordinate (in the camera's coordinate system) of the lower right corner of the front face of the view volume. Or, the distance from the center of the camera lens (that is, the view rectangle) to the right side of the lens.

`bottom` The *y* coordinate (in the camera's coordinate system) of the lower right corner of the front face of the view volume. Or, the distance from the center of the camera lens (that is, the view rectangle) to the left side of the lens.

DATA SIZE

16

DESCRIPTION

An orthographic camera is a parallel projection camera that employs an orthographic projection to obtain its image. The direction of projection is the opposite of the camera vector (that is, `location - pointOfInterest`), the projection plane is the near clipping plane, and the projection is thus along a normal to the projection plane. The origin of the projection plane is the point at `hither` (camera vector); if the absolute values of the fields `top` and `bottom` are equal, and the absolute values of the fields `left` and `right` are equal, then the origin of the projection plane is at the center of the front face of the view volume.

PARENT HIERARCHY

Shared, shape, camera.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

Camera placement, camera range, camera view port (optional). If a camera does not have one of these child objects, then it should be assigned the default values specified in the section on that child object.

EXAMPLE

```
OrthographicCamera (
    -10
    -10
    10
    10
)
```

View Plane Cameras

LABELS

ASCII	`ViewPlaneCamera`
Binary	`vwpl (= 0x7677706C)`

DATA FORMAT

```
Float32          viewPlane
Float32          halfWidthAtViewPlane
Float32          halfHeightatViewPlane
Float32          centerXOnViewPlane
Float32          centerYOnViewPlane
```

Field descriptions

`viewPlane` The distance from the camera location to the view plane.

`halfWidthAtViewPlane`
One half the width of the view plane window.

`halfHeightAtViewPlane`
The value in the `halfWidthAtViewPlane` field divided by the horizontal-to-vertical aspect ratio of the view port. The value in this field determines the half-height of the view plane window.

`centerXOnViewPlane`
The *x* coordinate of the center of the view plane window, specified in the view plane coordinate system.

`centerYOnViewPlane`
The *y* coordinate of the center of the view plane window, specified in the view plane coordinate system.

DATA SIZE

20

DESCRIPTION

A view plane camera is a type of perspective camera defined in terms of an arbitrary view plane. The camera vector is normal to the view plane, and the distance from the camera location to the view plane is measured in the direction defined by the camera vector. The window on the view plane and its center are defined in the projection plane coordinate system determined by the camera's camera placement object. The view volume of a view plane camera is determined by the four rays through the camera location and through the four corners of the rectangular window on the view plane, together with the two clipping planes. The view volume is the frustum whose top is the rectangle having as its vertices the intersections of these four rays with the near clipping plane and whose base is the rectangle having as its vertices the intersections of these rays with the far clipping plane.

The center of projection of a view plane camera is the camera location point. If the center of the window defined by a view plane camera is not at the origin of the view plane, then the camera yields an off-axis view. The projection determined by a view plane camera may have one, two, or three principal vanishing points.

A view plane camera may be used to obtain a close-up image of a single object by using the approximate center and dimensions of that object to specify the size and location of the window on the view plane.

PARENT HIERARCHY

Shared, shape, camera.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

Camera placement, camera view port, camera range (optional). If a camera does not have one of these child objects, then it should be assigned the default values specified in the section on that child object.

EXAMPLE

```
Container (
   ViewPlaneCamera (
      20
      15.0
      15.0
      18
      29
   )
   CameraPlacement ( ... )
   CameraRange ( ... )
   CameraViewPort ( ... )
)
```

View Angle Aspect Cameras

LABELS

ASCII	`ViewAngleAspectCamera`
Binary	`vana ( = 0x76616E61 )`

DATA FORMAT

`Float32`	`fieldOfView`
`Float32`	`aspectRatioXtoY`

Field descriptions

`fieldOfView` An angle, specified in radians, that defines the maximum field of view of the camera. The value in this field should lie in the open interval $(0, \pi)$.

`aspectRatioXtoY` The horizontal-to-vertical aspect ratio of the camera. If the value in this field is less than 1.0, the camera's field of view is vertical; otherwise, the camera's field of view is horizontal.

DATA SIZE

8

DESCRIPTION

A view angle aspect camera is a type of perspective camera defined in terms of a field of view angle and a horizontal-to-vertical aspect ratio. The aspect ratio determines the ratio of the base to the height of the rectangles that define the top and base of the camera's view volume. These rectangles lie in the near and far clipping planes, respectively, are upright in the camera's coordinate system, and are centered at the points of intersection of the line along the camera vector and the clipping planes.

If the aspect ratio is less than 1.0, then the field of view angle is in the $x = 0$ plane of the camera's coordinate system. Otherwise, the field of view angle is in the $y = 0$ plane of the camera's coordinate system. In both cases the rays that define the angle intersect in the camera location point, and the field of view angle is bisected by the ray from the camera location defined by the camera vector. The center of projection is the camera location point. The view volume of a view angle aspect camera is symmetrical about its center line. The method of projection determined by a view angle aspect camera has one principal vanishing point, located at the origin of the projection plane.

PARENT HIERARCHY

Shared, shape, camera.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

Camera placement, camera view port, camera range (optional). If a camera does not have one of these child objects, then it should be assigned the default values specified in the section on that child object.

EXAMPLE

```
Container (
   ViewAngleAspectCamera (
      1.7
      1.0
   )
   CameraPlacement ( ... )
   CameraRange ( ... )
   CameraViewPort ( ... )
)
```

Groups

Display Groups

LABELS

ASCII	`DisplayGroup`
Binary	`dspg ( = 0x6C697374 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A display group is a list of drawable objects and containers the root objects of which are drawable objects. Types of drawable objects include geometric objects, attribute sets, styles, transforms, and other display groups. A display group is delimited by begin group and end group objects.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

Display group state (optional). If no child object is specified, group state flags should be set to the default values specified in the section "Display Group States" on page 1-204.

EXAMPLE

```
BeginGroup ( Display Group( ) )
   SubdivisionStyle ( Constant 32 32 )
   Container (
      Mesh ( ... )
      VertexAttributeSetList ( ... )
      FaceAttributeSetList ( ... )
   )
   Container (
      Mesh ( ... )
      VertexAttributeSetList ( ... )
      FaceAttributeSetList ( ... )
   )
   .
   .
   .
EndGroup ( )
```

Ordered Display Groups

LABELS

ASCII	`OrderedDisplayGroup`
Binary	`ordg (= 0x6F72646C)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An ordered display group is a display group in which the objects listed are sorted by type. The elements of an ordered display group are listed in the following order: transforms, styles, attribute sets, shaders, geometric objects, other groups. An ordered display group is delimited by begin group and End group objects.

PARENT HIERARCHY

Shared, shape, group, display group.

PARENT OBJECTS

None.

CHILD OBJECTS

Display group state (optional). If no child object is specified, group state flags should be set to the default values specified in the section "Display Group States" on page 1-204.

EXAMPLE

```
BeginGroup ( OrderedDisplayGroup ( ) )
   RotateTransform ( ... )
   ScaleTransform ( ... )
   SubdivisionStyle ( ... )
   BackfacingStyle ( ... )
   BeginGroup ( DisplayGroup ( ) )
      .
      .
      .
   EndGroup ( )
EndGroup ( )
```

Light Groups

LABELS

ASCII	`LightGroup`
Binary	`lghg ( = 0x676C6768 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A light group is simply a list of light objects. A light group is delimited by begin group and end group objects.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( LightGroup ( ) )
   AmbientLight ( )
   DirectionalLight ( ... )
   SpotLight ( ... )
EndGroup ( )
```

I/O Proxy Display Groups

LABELS

ASCII	`IOProxyDisplayGroup`
Binary	`iopx (= 0x70727879)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An I/O proxy display group is used to place distinct specifications of the same model together in a group. The purpose of an I/O proxy display group is to permit a reading application that does not recognize all specifications of a model to pass over those that it does not recognize until it encounters one that

it does recognize and can use to recover the model. For example, a pentagon may be represented by either a mesh or a polygon. If both representations are placed together in an I/O proxy display Group, then a reading application that recognizes meshes but does not recognize polygons can recover the pentagon from its mesh representation.

Representations of a model in an I/O proxy display Group should appear in preferential order: any representation of a model is to be preferred to any other representation of that model occurring later in the group. While drawing, bounding, or picking, the reading application should use the first representation of the model that it recognizes and should ignore all other representations.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( IOProxyDisplayGroup ( ) )
   Polygon ( ... )                    # first preference
   GeneralPolygon ( ... )             # second preference
   Mesh                               # third preference
EndGroup ( )
```

Info Groups

LABELS

ASCII `InfoGroup`

Binary `info ( = 0x696E666F )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An info group is a list of string objects delimited by begin group and end group objects. An info group allows objects containing information in text form to be placed together in a group.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( InfoGroup ( ) )
   CString ( ... )
   .
   .
   .
   CString ( ... )
EndGroup ( )
```

Groups (Generic)

LABELS

ASCII	`Group`
Binary	`grup ( = 0x67727570 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A group (generic) is simply a list of drawable objects, delimited by begin group and end group objects.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( Group ( ) )
    .
    .
    .
EndGroup ( )
```

Begin Group Objects

LABELS

ASCII	`BeginGroup`
Binary	`bgng (= 0x62676E67)`

DESCRIPTION

A begin group object is used to declare a group and to delimit the start of that group. Every group must begin with a begin group object.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup(
   DisplayGroup ( )              # empty group
)
EndGroup ( )
```

End Group Objects

LABELS

ASCII	`EndGroup`
Binary	`endg ( = 0x656E6467 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An end group object is placed immediately after the last object in a group and is used to delimit that group.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )      # empty group
EndGroup ( )
```

Display Group States

LABELS

ASCII	`DisplayGroupState`
Binary	`dgst (= 0x64677374)`

DISPLAY GROUP STATE FLAGS

```
None                0x00000000
IsInline            0x00000001
DoNotDraw           0x00000002
NoBoundingBox       0x00000004
NoBoundingSphere    0x00000008
DoNotPick           0x00000010
```

Constant descriptions

`None`	No flags are specified.
`IsInline`	The parent group is to be executed inline (that is, without pushing the graphics state on a stack before execution and popping it after execution). This flag is used to prevent the objects in the parent group from inheriting properties specified at a higher level in a hierarchical model containing the parent group. If this flag is set, then objects in the parent group receive only those properties specified in that group.

`DoNotDraw`	The parent group is not to be drawn when rendering or picking. If this flag is set, then the parent group is not to be traversed when it is encountered in a hierarchical model.
`NoBoundingBox`	The bounding box of the parent group is not to be used for rendering.
`NoBoundingSphere`	The bounding sphere of the parent group is not to be used for rendering.
`DoNotPick`	The parent group is not eligible for inclusion in the hit list of a pick object.

DATA FORMAT

```
DisplayGroupStateFlags          traversalFlags
```

Field descriptions

`traversalFlags`	A bitfield expression specifying one or more display group state flags.

DATA SIZE

4

DESCRIPTION

A display group state object is used to specify a set of flags that determines how its parent display group is to be traversed during rendering or picking and whether a bounding box or bounding sphere is to be used during rendering. If a display group does not have a display group state object as a child object, that group's state flags should be set to the default state specified below.

In a text file, a display group state object should be placed together with a group object in the begin group object that immediately precedes that group.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Display group, ordered display group. A display group state object always has a parent object.

CHILD OBJECTS

None.

DEFAULT DISPLAY GROUP STATE FLAGS

```
None (= 0x00000000)
```

EXAMPLE

```
BeginGroup (
   DisplayGroup ( )
   DisplayGroupState ( DoNotPick )
)
.
.
.
EndGroup ( )
```

Renderers

Wireframe Renderers

LABELS

ASCII	`WireFrame`
Binary	`wrfr (= 0x77726672)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A wireframe renderer creates line drawings of models. Such a renderer does not decompose polylines or polygons during rendering. It can render all backfacing, point, and edge drawing styles.

PARENT HIERARCHY

Shared, renderer.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    Wireframe ( )
    ViewPlaneCamera ( ... )
    PointLight ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
)
```

Interactive Renderers

LABELS

ASCII	`InteractiveRenderer`
Binary	`ctwn (= 0x6374776E)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

The interactive renderer uses a fast and accurate depth-sorting algorithm for drawing solid, shaded surfaces as well as vectors. The interactive renderer is also capable of rendering highly detailed, complex models with very realistic surface illumination and shading, but at the expense of time and memory.

PARENT HIERARCHY

Shared, renderer.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    InteractiveRenderer ( )
    ViewPlaneCamera ( ... )
    PointLight ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
)
```

Generic Renderers

LABELS

ASCII	`GenericRenderer`
Binary	`gnrr (= 0x676E7272)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A generic renderer performs no rendering functions, but may be used to pick or to accumulate state.

PARENT HIERARCHY

Shared, renderer.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    GenericRenderer ( )
    ViewPlaneCamera ( ... )
    PointLight ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
)
```

Shaders

Shader Data Objects

LABELS

ASCII	`Shader`
Binary	`shdr` (`= 0x73686472`)

SHADER UV BOUNDARY TYPES

`Wrap`	`0x00000000`
`Clamp`	`0x00000001`

Constant descriptions

`Wrap` Values outside the valid range of *uv* values are to be wrapped. To wrap a shader effect is to replicate the entire effect across the mapped area.

`Clamp` Values outside the valid range of *uv* values are to be clamped. To clamp a shader effect is to replicate the boundaries of the effect across the portion of the mapped area that lies outside the valid range.

DATA FORMAT

```
ShaderUVBoundaryEnum        uBounds
ShaderUVBoundaryEnum        vBounds
```

Field descriptions

`uBounds` The value in this field determines whether values in the *u* parametric direction that lie outside the valid range are wrapped or clamped by the parent shader.

`vBounds` The value in this field determines whether values in the *v* parametric direction that lie outside the valid range are wrapped or clamped by the parent shader.

DATA SIZE

8

DESCRIPTION

A shader data object is a boundary-handling method specifier that determines how a parent shader handles parametric *uv* values that are outside the valid range (namely, 0 to 1).

PARENT HIERARCHY

Data.

PARENT OBJECTS

Any shader. A shader data object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    CustomShader ( ... )
    ShaderData ( Wrap Clamp )
)
```

DEFAULT VALUES

```
Wrap Wrap
```

Texture Shaders

LABELS

ASCII	`TextureShader`
Binary	`txsu (= 0x74787375)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A texture shader is used to apply a texture to a surface in shading.

PARENT HIERARCHY

Shared, shape, shader, surface shader.

PARENT OBJECTS

None.

CHILD OBJECTS

Pixmap texture object. A texture shader always has one child object.

EXAMPLE

```
Container (
   TextureShader ( )
   PixmapTexture ( ... )
)
```

Pixmap Texture Objects

LABELS

ASCII	`PixmapTexture`
Binary	`txpm (= 0x7478706D)`

ENDIAN TYPES

```
BigEndian        0x00000000
LittleEndian     0x00000001
```

Constant descriptions

`BigEndian`	Packing is to be done in a big-endian manner.
`LittleEndian`	Packing is to be done in a little-endian manner.

PIXEL TYPES

```
RGB8            0x00000000
RGB16           0x00000001
RGB24           0x00000002
RGB32           0x00000003
```

Constant descriptions

`RGB8` 8 bits are devoted to each pixel in the pixmap.

`RGB16` 16 bits are devoted to each pixel in the pixmap.

`RGB24` 24 bits are devoted to each pixel in the pixmap.

`RGB32` 32 bits are devoted to each pixel in the pixmap.

DATA FORMAT

```
Uns32           width
Uns32           height
Uns32           rowBytes
Uns32           pixelSize
PixelTypeEnum   pixelType
EndianEnum      bitOrder
EndianEnum      byteOrder
RawData         image[rowBytes * height]
```

Field descriptions

`width` The width of the pixmap. The value in this field must be greater than 0.

`height` The height of the pixmap. The value in this field must be greater than 0.

`rowBytes` The number of bytes in a row of the pixmap. The value in this field cannot be less than the product of the values in the `width` and `pixelSize` fields.

`pixelSize` The size of each pixel in the pixmap. The value in this field must be greater than 0 and less than 32.

`pixelType` The type of the pixels of the pixmap.

`bitOrder`	The order in which the bits in a byte are addressed. This field must contain one of the constants `BigEndian` or `LittleEndian`.
`byteOrder`	The order in which the bytes in a word are addressed. This field must contain one of the constants `BigEndian` or `LittleEndian`.
`image[]`	The array that defines the pixmap.

DATA SIZE

```
28 + rowBytes * height + padding
```

DESCRIPTION

A pixmap texture object is a generic method of transferring pixmap data that is used in conjunction with a texture shader.

PARENT HIERARCHY

Shared, texture.

PARENT OBJECTS

Texture shader. A pixmap texture object sometimes, but not always, has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
PixmapTexture (
   256 256                      # width/height
   128                          # rowBytes
   32                           # pixelSize
   RGB24
   BigEndian BigEndian
```

```
    0x00123232...
    0x...
)
```

View Objects

View Hints

LABELS

ASCII	`ViewHints`
Binary	`vwhn ( = 0x7677686E )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

The view hints object is used to group together all of the objects needed to render an image from a model (that is, a renderer, a camera, lights, and any additional information to be supplied to the renderer). These other objects occur as child objects to the view hints object; a container may be used to group them together. The container holding a view hints object and its associated rendering specifications should be placed immediately before the models to be rendered according to those specifications.

A metafile may contain more than one view hints object. If a metafile contains more than one view hints object, the specifications associated with each view hints object are inherited by all subsequent view hints objects, unless overridden by contrary specifications. Accordingly, a subsequent view hints object need have as child objects only those specifications that differ from those

of its predecessors. For example, you may wish to render the same model using different cameras, while keeping the lights and other specifications intact. Once the initial specifications have been made, you need only specify a different camera together with a new view hints object. The model may be placed in the scope of a subsequent view hints object through the use of a reference object; the specification of the model need not be repeated.

PARENT HIERARCHY

Shared.

PARENT OBJECTS

None.

CHILD OBJECTS

Renderer, camera, lights (as many as desired), attribute set, image dimensions, image mask, image clear color (all optional).

EXAMPLE

```
3DMetafile ( 1 0 Normal toc> )
Container (
   ViewHints ( )
   Container (
      ViewAngleAspectCamera ( 0.73 1.0 )
      CameraPlacement (
         0 0 30
         0 0 0
         0 1 0
      )
   )
   DirectionalLight ( -0.7 -0.7 -0.65 )
   Container (
      AttributeSet ( )
      DiffuseColor ( 0.2 0.2 0.2 )
      SpecularControl ( 3 )
```

```
    )
    ImageDimensions ( 200 200 )
)
ref1:
BeginGroup ( DisplayGroup ( ) )
.
.
.
EndGroup ( )
Container (
    ViewHints ( )
    Container (
        ViewAngleAspectCamera ( 0.73 1.0 )
        CameraPlacement (
            0 10 0
            0 0 0
            0 1 0
        )
    )
)
Reference ( 1 )
```

Image Masks

LABELS

ASCII	`ImageMask`
Binary	`immk (= 0x696D6D6B )`

DATA FORMAT

`Uns32`	`width`
`Uns32`	`height`
`Uns32`	`rowBytes`
`RawData`	`image[rowBytes * height]`

Field descriptions

`width`	The width, in bits, of the bitmap whose bits are listed in the array `image[]`. The value in this field should be greater than 0.
`height`	The height, in bits, of the bitmap whose bits are listed in the array `image[]`. The value in this field should be greater than 0.
`rowBytes`	The number of bytes in a row of the bitmap.
`image[]`	An array of bit specifications.

DATA SIZE

```
12 + (rowBytes * height) + padding
```

DESCRIPTION

An image mask is a bitmap that is used to mask out certain portions of an image. The values in the `width` and `height` fields of an image mask specify the boundaries of the rectangular subregion of an image that is actually to be drawn. (Width and height are measured from the upper-left corner of the image to which a mask is applied.) Each bit listed in the array `images[]` corresponds to 1 pixel in the rectangle defined by the width and height of the mask. If a bit is set, then the corresponding pixel is drawn with the color determined by the underlying image. If a bit is clear, then the corresponding pixel is drawn black. Normally, an image mask is applied to an image after that image has been rasterized.

An image dimensions object may be used together with an image mask: the former may be used to clip an image, and the latter may be used to filter the clipped image.

PARENT HIERARCHY

Data, view hints data.

PARENT OBJECTS

View hints. An image mask always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
3DMetafile ( 1 0 Normal toc> )
Container (
   ViewHints ( )
   ImageDimensions ( 32 32 )
   ImageClearColor ( 1 1 1 )
   ImageMask (
      32 32                                          # width, height
      4                                              # rowBytes
      BigEndian                                      # bitOrder
      0x000000000FFFF8000FFFF8000FFFF800
      0x0FFFF8000FFFF8000FFFF8000FFFFFE0
      0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
      0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
      0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
      0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
      0x0C61FFE00F24FFE00E64FFE00F24FFE0
      0x0F24FFE00C61FFE00FFFFFE000000000
   )
)
Rotate ( X 0.25 )
Rotate ( Y 0.23 )
Container (
   Torus ( 0 0.7 0 0 0 1 1 0 0 0 0 0 0.7 )
   Container (
      AttributeSet ( )
      DiffuseColor ( 0.2 0.9 0.9 )
   )
)
```

Image Dimensions Objects

LABELS

ASCII	`ImageDimensions`
Binary	`imdm (= 0x696D646D)`

DATA FORMAT

```
Uns32           width
Uns32           height
```

Field descriptions

`width` The preferred width, in pixels, of the displayed portion of an image.

`height` The preferred height, in pixels, of the displayed portion of an image.

DATA SIZE

8

DESCRIPTION

An image dimensions object is used to specify the height and width of the rectangular portion of an image that is to be displayed. The height and width of an image dimensions object are measured from the upper-left corner of the image to which that image dimensions object is applied. Normally, an image is rasterized before an image dimensions object is applied to it. An image dimensions object may be used together with an image mask: the former may be used to clip an image, and the latter may be used to filter the clipped image.

PARENT HIERARCHY

Data, view hints data.

PARENT OBJECTS

View hints. An image dimensions object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   ViewHints ( )
   ImageDimensions ( 32 32 )
   ImageMask ( ... )
)
```

Image Clear Color Objects

LABELS

ASCII	`ImageClearColor`
Binary	`imcc ( = 0x696D6363 )`

DATA FORMAT

`ColorRGB`	`clearColor`

Field descriptions

`clearColor`	The RGB color to be given to the visible background of a model when an image is rendered from that model.

DATA SIZE

4

DESCRIPTION

An image clear color object is used to assign color to the background of a model in a rendered image when the model does not itself completely fill that image.

PARENT HIERARCHY

Data, view hints data.

PARENT OBJECTS

View hints. An image clear color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
   ViewHints ( )
   ImageDimensions ( ... )
   ImageClearColor ( 1 1 1 )
   .
   .
   .
)
```

Unknown Objects

Unknown Text

LABELS

ASCII `UnknownText`

Binary	`uktx ( = 0x756B7478 )`

DATA FORMAT

```
String          asciiName
String          contents
```

Field descriptions

`asciiName` The object type of the unknown object, enclosed in double quotation marks.

`contents` The specification (without encapsulation) of the unknown object, enclosed in double quotation marks. Blank space and comments in the original object specification of the unknown object may be omitted when this field is written.

DATA SIZE

```
sizeof(asciiName) + sizeof(contents)
```

DESCRIPTION

An unknown text object is used to transport unknown data found in a text file. It is an encapsulated replica of that unknown data. In the usual case, an unknown text object contains an ill-formed object specification. Your file reading program may be designed to transport the data contained in an unknown text object, to validate and convert the data to a specification of a known object, or to discard the data.

An unknown text object may occur in a binary file as well as in a text file.

PARENT HIERARCHY

Shared, shape.

PARENT OBJECTS

Any object that may have a child object may be a parent object to an unknown text object.

CHILD OBJECTS

None.

EXAMPLE

```
UnknownText (
   "Sphere"                              # unknown object type
   "1 0 0    0 1 0     0 0 1     0 0 a" # illegal specification
)
```

Unknown Binary

LABELS

ASCII	`UnknownBinary`
Binary	`ukbn ( = 0x756B626E )`

DATA FORMAT

`Int32`	`objectType`
`Uns32`	`objectSize`
`EndianEnum`	`byteOrder`
`RawData`	`objectData[objectSize]`

Field descriptions

`objectType`	The binary representation of the type of the unknown object.
`objectSize`	The size of the unknown object.
`byteOrder`	The byte order of the unknown object. The information in this field is needed to transport unknown data between processors and permits parsing endian-specific primitives contained in the object data.
`objectData[]`	The specification of the unknown object in the form of raw data.

DATA SIZE

```
12 + sizeof(objectData)
```

DESCRIPTION

An unknown binary object is used to transport unknown data found in a binary file. It is an encapsulated replica of that unknown data. In the usual case, an unknown binary object contains an ill-formed object specification. Your file reading program may be designed to transport the data contained in an unknown text object, to validate and convert the data to a specification of a known object, or to discard the data.

An unknown binary object may occur in a text file as well as in a binary file.

PARENT HIERARCHY

Shared, shape.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
UnknownBinary (
   1701605476
   4
   BigEndian
   0x0AB2
)
```

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter NTX printer. Final page negatives were output directly from text files on an Agfa Large-Format Imagesetter. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Courier.

WRITER
Malcolm MacFail

LEAD WRITER
Tim Monroe

DEVELOPMENTAL EDITORS
Beverly Zegarski

ILLUSTRATOR
Sandee Karr

PROJECT MANAGER
Patricia Eastman

Special thanks to Kent Davidson, Pablo Fernicola, and Klaus Strelau.