Open Inventor:

How to Write an Open Inventor File Translator

CONTRIBUTORS

Written by Josie Wernecke and Eleanor Bassler
Engineering contributions by Rikk Carey and Paul Strauss

Contents

Figures

Chapter 1

Introduction

An important feature in Open Inventor is its **3D Interchange File Format**, which provides a much-needed standard for exchanging 3D data among applications. Inventor's file format supports both ASCII and binary files.

- The ASCII file format is a simple, human-readable representation of a 3D scene database.
- Binary files are written in a machine-independent format. Although the binary format is not public, tools are readily available for converting an Inventor ASCII file to binary format.

Translator programs make possible the flow of 3D data between application programs using Open Inventor and non-Inventor. Figure 1-1 illustrates this data flow.

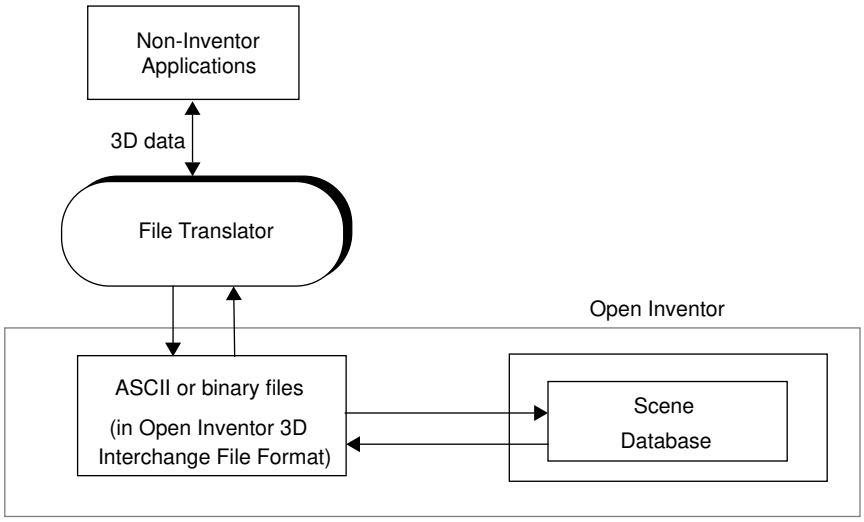

Figure 1-1 Exchanging 3D data between Inventor and non-Inventor applications

Other types of data have been converging on standard formats such as TIFF, GIF, PICT, and PostScript, but there is currently no clear winner for a standard 3D format. Once a standard emerges, all 3D developers and users can benefit from a common file format that allows data exchange among members of a world-wide audience. With a shared file format, users can cut and paste among a variety of applications on the desktop, and developers have access to a wide range of tools and code to boost their productivity. In addition, developers need only write one translator to and from Inventor format, rather than a whole collection of file translators for each 3D file format in the industry.

What Is in This Document?

This document provides the necessary basic information to enable you to write a program that translates existing graphics files into Open Inventor format. It includes the following sections:

- Background information on the Inventor scene database
- An example of a simple file translator
- Tips and suggestions for writing file translators

For a more complete description of Inventor objects, creating a scene database, and applying actions, see *The Inventor Mentor,* Chapters 1 through 5 and Chapter 9.

Things You Need to Know about Inventor

The following paragraphs outline basic concepts of an Inventor scene database.

Scene Database

An Inventor *scene database* is a collection of 3D objects and properties arranged appropriately to represent a 3D scene or data set. Inventor programs create or read their own copies of scene databases each time they execute. A scene database resides in the program's memory while the

program is running, unlike a traditional database that resides on disk and is shared by many running programs.

Nodes and Fields

A node is an object that represents a 3D shape, property, or group. *Shape nodes* represent 3D geometric objects. *Property nodes* represent appearance and other qualitative characteristics of the scene. *Group nodes* are containers that collect nodes into graphs. Other important nodes include camera, light, and callback nodes.

Nodes contain both data and functions. The data elements contained in a node are called *fields*. When you create a node, all the fields within that node are created as well, and each field automatically contains a default value. For example, a point light node contains these fields:

Field	Default Value	Meaning
on	TRUE	whether light is active
intensity	1.0	value between 0.0 and 1.0 (maximum illumination)
color	1.0 1.0 1.0	red, green, blue color of light
location	0.0 0.0 1.0	position in *x*, *y*, *z*

See the *Open Inventor Nodes Quick Reference* for a complete alphabetical listing of all Inventor nodes as well as their fields and default values.

Node Icons

Figure 1-2 contains the legend for nodes used in the diagrams in this document.

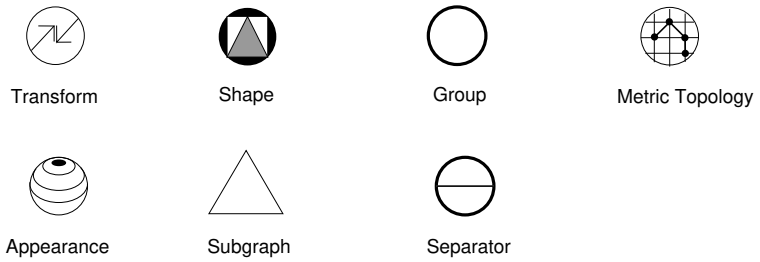

Figure 1-2 Node icons

Scene Graphs

A *scene graph* is an ordered collection of nodes. Hierarchical scene graphs are created by adding nodes as *children* of group nodes. Figure 1-3 shows a simple scene graph. The top node of the graph (in this figure, "top") is called the *root* node. The Inventor scene database can contain any number of scene graphs, each consisting of a related set of 3D objects and attributes. Typically, a 3D scene or a set of object files contains only one scene graph. However, this is not a restriction.

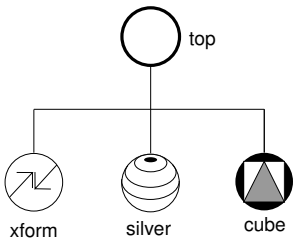

Figure 1-3 Simple scene graph

Group Nodes

A group node contains an ordered list of children that are traversed from left to right. A *separator* group, shown in Figure 1-4, is a special type of group node. Nodes under the separator group do not affect nodes in the graph after the separator.

When you write a scene database, the objects in the database are written from top to bottom and from left to right. Objects lower (and to the right) in the scene graph *inherit* the attributes and values set by objects that precede them. If you do not want subsequent objects to inherit certain properties or values, use a *separator* group, which pushes and pops properties during traversal (see Figure 1-4; the red material in the separator A group does not affect the cone in the separator B group). The root node of a scene graph is usually a separator.

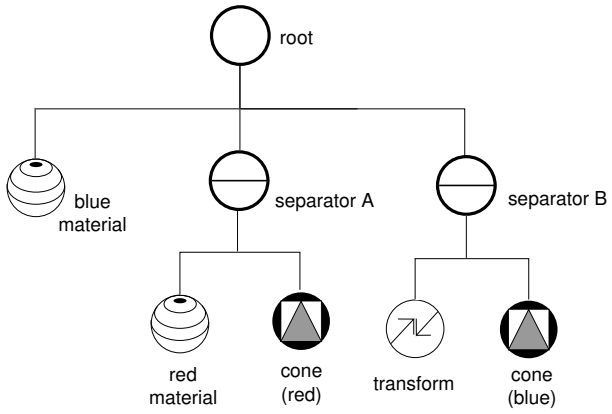

Figure 1-4 Example of separator groups

Translating Files into Inventor Format

This section outlines a general methodology for writing an Inventor file translator and presents a sample file translation program. The sample program translates Silicon Graphics Object (SGO) data files into Open Inventor files. This example is presented as one possible way to write a file translator. There are, of course, many other possible solutions, depending on your needs, the type of data you are working with, and the structure of the files you are translating. Chapter 11 in *The Inventor Mentor* describes the Inventor file format in great detail.

General Steps

The basic steps in an Inventor file translation program can be summarized as follows:

1. Read and verify the file header of the input file (if applicable).
2. Initialize the Inventor database. This step includes creating the root node or nodes of the database and setting up nodes containing any default attributes that will be used by other nodes in the scene.
3. Enter the object read loop. Read the first object from the input file, generate an Inventor object, and put it into the database. Continue reading objects from the input file until all objects are read and translated.
4. Clean up the Inventor database. Reorganize it for maximum efficiency.
5. Write the Inventor database to a file.

The following subsections discuss each step in more detail. The complete program is shown in "Complete Sample Program" on page 16.

SGO File Format

Since the program example translates files from SGO file format, you'll need to know something about this format before you look at the example in detail.

Figure 2-1 shows the basic structure of the SGO file format. The first word in the file must be the SGO magic number, a code number used to identify the file (0x5424). The magic number is 4 bytes long.

The objects in SGO files are constructed from three data types: quadrilateral lists, triangle lists, and triangle meshes. One SGO file can contain any number of objects of differing type. An identifying token (4 bytes) precedes the data for each object:

```
OBJ_QUADLIST  (= 1)         quadrilateral list

OBJ_TRILIST   (= 2)         triangle list

OBJ_TRIMESH   (= 3)         triangle mesh
```

The end of data token is `OBJ_END` (= 4). This token is placed after the data for the last object in the file.

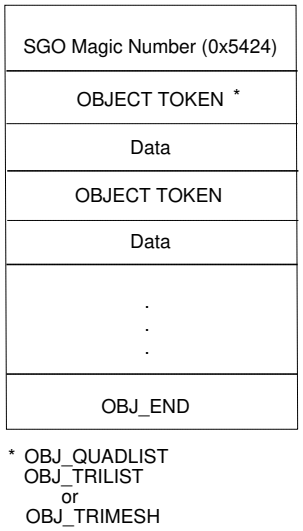

Figure 2-1 SGO file format

SGO Quadrilateral List

Figure 2-2 shows the structure of an SGO quadrilateral list object. The object begins with the object token, followed by the size (in 4-byte words) of the data for this object. Next follow nine words of data for each vertex in the object. As shown at the bottom of Figure 2-2, the first three words are the *xyz* components of the normal vector at the vertex. The next three words are the *RGB* color components at the vertex. The last three words are the *xyz* coordinates of the vertex itself.

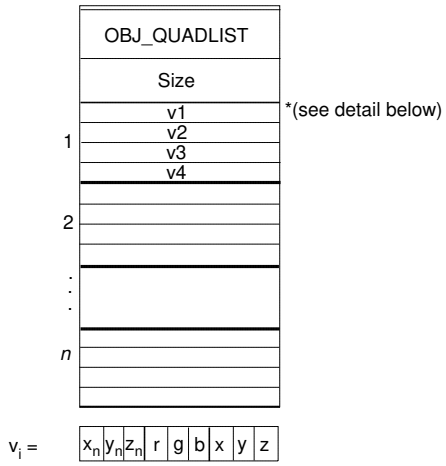

Figure 2-2 SGO quadrilateral list object

SGO Triangle List

Figure 2-3 shows the structure of an SGO triangle list object. The object begins with the object token, followed by the size (in 4-byte words) of the data for this object. Next follow nine words of data for each vertex in the object. As shown at the bottom of Figure 2-3, the first three words are the *xyz* components of the normal vector at the vertex. The next three words are the *RGB* color components at the vertex. The last three words are the *xyz* values of the vertex itself.

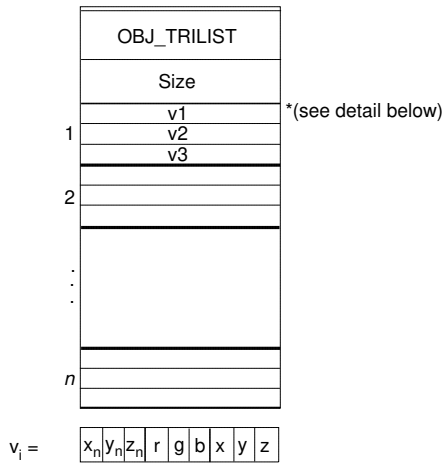

Figure 2-3 SGO triangle list object

SGO Triangle Mesh

In the interest of brevity, the SGO triangle mesh object is not discussed in this document or illustrated in the example.

Reading the File Header

The first step in the file translation program is to check that the header of the input file is in the expected format. SGO files, used in our example, are identified by a special code number, 0x5424, known as the SGO "magic number." If the file header does not contain the magic number, the function returns FALSE.

```
static SbBool
readHeader()
{
   long   magic;

   return (fread(&magic, sizeof(long), 1, stdin) == 1 &&
           magic == SGO_MAGIC);
}
```

Initializing the Inventor Database

This step comprises three parts:

- initializing the Inventor database
- creating a root node for the database
- creating nodes with default attributes that will be used by other nodes in the database

Initializing the Database

The following code initializes the Inventor database:

```
SoDB::init();
```

Creating the Root Node

The root node for the database is typically a separator node (see“Group Nodes” on page 5). The root node is always *referenced*, as shown below. This ensures that the root node is not accidentally deleted. In most cases, you will use a separator as your root node. If in doubt, use a separator.

```
root = new SoSeparator;
root->ref();
```

Setting Up Default Attributes

This step involves setting up default or global attributes that are the same for all objects in the scene graph.

SGO objects include values for normals and colors along with each index. In Inventor, you need to specify how this information is applied to the shape objects in the scene graph. For example, a color can be applied, or “bound,” to an entire shape, to each face in the shape, or to each vertex in the shape.

In our example, we want the normals and materials to be bound to each vertex in the shape object. This is termed *per-vertex binding*. The sample

program creates the following two Inventor nodes and adds them to the scene graph:

SoMaterialBinding	tells how to bind the specified materials to the shape node
SoNormalBinding	tells how to bind the specified normals to the shape node

Note: While the sample program does not translate SGO triangle mesh objects, your program may require this translation. If so, you should note that this object presents a special case because it lists colors and normals in an arbitrary order and then indexes into the list. For triangle mesh objects, you should use Inventor's *per-vertex indexed* binding. When no indices are present, this binding defaults to per-vertex binding (normals and materials are used in order).

See *The Inventor Mentor*, Chapter 5, for a detailed description of binding nodes.

Here is the code that creates the material and normal binding nodes and specifies per-vertex binding for both nodes:

```
mtlBind = new SoMaterialBinding;
normBind = new SoNormalBinding;
mtlBind->value = SoMaterialBinding::PER_VERTEX;
normBind->value = SoNormalBinding::PER_VERTEX;
root->addChild(mtlBind);
root->addChild(normBind);
```

At this point, the Inventor database looks like Figure 2-4.

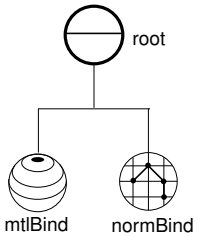

Figure 2-4 Nodes created during initialization

Entering the Object Read Loop

The object read loop is where the majority of the database objects are created. This loop reads the first object from the input file. It determines which type of object follows and calls one of three functions: **readQuadList()**, **readTriList()**, or **readTriMesh()**. Then it generates Inventor nodes to represent the corresponding data and puts those objects into the database that was initialized in step 1. This loop continues reading objects from the input file until all objects have been read and translated into Inventor nodes.

The SGO objects OBJ_QUADLIST and OBJ_TRILIST are translated into the Inventor shape node **SoFaceSet**. The OBJ_TRIMESH object is not implemented for this example.

As described earlier, SGO objects contain the following nine words of data for each vertex:

- Normals (*x*, *y*, *z*)
- Colors (*r*, *g*, *b*)
- Coordinates (*x*, *y*, *z*)

In Inventor, each of these three sets of information is contained in a separate node as follows:

- **SoNormal** - contains all vertex normals for a shape
- **SoBaseColor** - contains the red/green/blue values for the base color of the vertices (An **SoMaterial** node could be used here, but **SoBaseColor** is more efficient since only the diffuse color is changing.)
- **SoCoordinate3** - contains the coordinates for the vertices

The sample program reads an SGO object, checks the number of vertices, and then makes the appropriate amount of room in the corresponding Inventor nodes to hold all the data for that object. Note that the SGO object groups the normals, colors, and coordinates for each vertex. The sample program unpacks this combined vertex data and reorganizes it into three separate Inventor nodes (all normals for the shape go into the normal node, all colors go into the base color node, and so on).

For example, the **readQuadList()** function in the sample program creates the four Inventor nodes shown in Figure 2-5. These nodes are then added as

children of an **SoSeparator** node. The **readTriList()** function in the sample program translates an SGO triangle list into the same group of nodes shown in Figure 2-5.

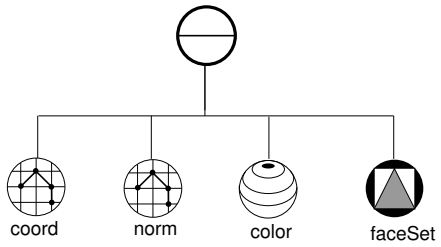

Figure 2-5 Inventor nodes for a face set object

Each group of nodes (called a *subgraph*) is added to the database. Figure 2-6 shows the scene graph after reading the SGO object TRILIST.

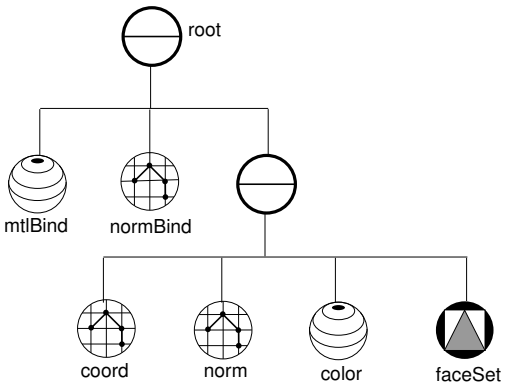

Figure 2-6 Inventor database after reading the SGO object TRILIST

Here is the code for the object read loop.

```
// Keep reading objects until we are done or we have an error
while (! endFound) {
    switch (readObjectType()) {

        case SGO_OBJ_QUADLIST:
            obj = readQuadList();
            break;

        case SGO_OBJ_TRILIST:
            obj = readTriList():
            break;

        case SGO_OBJ_TRIMESH:
            // Not implemented in this example
            break;

        case SGO_OBJ_END:
            endFound = TRUE;
            break;

        default:
            error("Missing or invalid object type");
            break;
    }

    if (! endFound) {
        if (obj == NULL)
            error("Bad object data");

        root->addChild(obj);
    }
}
```

Writing the Database to a File

Now you are ready to write the database to a file. This is the easy part:

```
SoWriteAction     wa;    // default writes in ASCII to stdout
wa.apply(root);
```

To write to a filename in binary:

```
SoWriteAction    wa;
SoOutput  *out = wa.getOutput();
```

```
if (out->openFile(filename)!=NULL) {
     out->setBinary(TRUE);
     wa.apply(root)
}
```

See Chapter 3, “Tips and Guidelines” for ways to test the resulting Inventor file.

Complete Sample Program

Here is the complete sample program that translates SGO files into Open Inventor files.

```
// usage: SgoToIv file.sgo > file.iv
//

#include <Inventor/SoDB.h>
#include <Inventor/actions/SoSearchAction.h>
#include <Inventor/actions/SoWriteAction.h>
#include <Inventor/nodes/SoBaseColor.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoMaterialBinding.h>
#include <Inventor/nodes/SoNormal.h>
#include <Inventor/nodes/SoNormalBinding.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/SoPath.h>

// SGO format codes
#define SGO_MAGIC               0x5424

#define SGO_OBJ_QUADLIST        1
#define SGO_OBJ_TRILIST         2
#define SGO_OBJ_TRIMESH         3
#define SGO_OBJ_END             4
#define SGO_OBJ_ERROR         (-1)  /* No such object type */

#define SGO_OP_BGNTMESH         1
#define SGO_OP_SWAPTMESH        2
#define SGO_OP_ENDBGNTMESH      3
#define SGO_OP_ENDTMESH         4
```

```
//////////////////////////////////////////////////////////////
// Prints an error message to stderr and exits.
//////////////////////////////////////////////////////////////

static void
error(const char *message)
{
   fprintf(stderr, "SgoToIv: %s\n", message);
   exit(1);
}

//////////////////////////////////////////////////////////////
// Reads header (magic number) from file. Returns FALSE on
// error.
//////////////////////////////////////////////////////////////

static SbBool
readHeader(FILE *file)
{
   long        magic;

   return (fread(&magic, sizeof(long), 1, file)
                          == 1 && magic == SGO_MAGIC);
}

//////////////////////////////////////////////////////////////
// Reads SGO object type from file and returns it.
//////////////////////////////////////////////////////////////

static long
readObjectType(FILE *file)
{
   long        type;

   if (fread(&type, sizeof(long), 1, file) != 1)
      return SGO_OBJ_ERROR;

   return type;
}

//////////////////////////////////////////////////////////////
// Reads N SGO vertices into the passed array of floats,
// which should be big enough to hold the data
// (N * 9 floats). Returns FALSE on a bad read.
//////////////////////////////////////////////////////////////

static SbBool
readVertices(int numVerts, float verts[], FILE *file)
```

```
{
   // Read vertices (9 floats each)
   return (fread(verts, sizeof(float), numVerts * 9, file)
                           == numVerts * 9);
}

////////////////////////////////////////////////////////////
// Reads a quadrilateral list SGO object while creating an
// Inventor graph to represent it. Returns the root of the
/ graph, or NULL if there's an error.
////////////////////////////////////////////////////////////

static SoNode *
readQuadList(FILE *file)
{
   long                numWords;
   int                 numQuads, quad, vert, fieldIndex,
                                 vertIndex;
   float               verts[36];
   SoSeparator         *root;
   SoCoordinate3       *coord;
   SoNormal            *norm;
   SoBaseColor         *color;
   SoFaceSet           *faceSet;

   fprintf(stderr, "Reading a quad list\n");

   // Read number of words in data
   if (fread(&numWords, sizeof(long), 1, file) != 1)
      return NULL;

   // There are 36 words (4 vertices of 9 words each) per
   // quadrilateral
   numQuads = (int) numWords / 36;

   // Create nodes to hold all the vertex and shape info
   coord = new SoCoordinate3;
   norm  = new SoNormal;
   color = new SoBaseColor;
   faceSet  = new SoFaceSet;

   // Because we know how many vertices there are, we can
   // make the appropriate amount of room in the fields of
   // the nodes.
   coord->point.setNum(4 * numQuads);
   color->rgb.setNum(4 * numQuads);
   norm->vector.setNum(4 * numQuads);
   faceSet->numVertices.setNum(numQuads);
```

```
   // Process each quadrilateral
   for (quad = 0; quad < numQuads; quad++) {

      // Read 4 vertices
      if (! readVertices(4, verts, file))
         return NULL;

      // Store vertex info in fields
      for (vert = 0; vert < 4; vert++) {

         // Get index into appropriate place in field and
         // vertex
         fieldIndex = 4 * quad + vert;
         vertIndex  = 9 * vert;

         norm->vector.set1Value(fieldIndex,
                   &verts[vertIndex + 0]);
         color->rgb.set1Value(fieldIndex,
                   &verts[vertIndex + 3]);
         coord->point.set1Value(fieldIndex,
                   &verts[vertIndex + 6]);
      }

      // Store number of vertices of quadrilateral
      faceSet->numVertices.set1Value(quad, 4);
   }

   // Create a root separator to hold the subgraph. We don't
   // need to ref() it because we aren't doing anything to
   // the subgraph until it is added to the main graph
   // (after this returns).
   root = new SoSeparator(4);

   // Add the nodes to the root
   root->addChild(coord);
   root->addChild(norm);
   root->addChild(color);
   root->addChild(faceSet);

   return root;
}

//////////////////////////////////////////////////////////////
// Reads a triangle list SGO object while creating an
// Inventor graph to represent it. Returns the root of the
//  graph, or NULL if there's an error.
//////////////////////////////////////////////////////////////

static SoNode *
```

```
readTriList(FILE *file)
{
   long                numWords;
   int                 numTris, tri, vert, fieldIndex,
                                vertIndex;
   float               verts[27];
   SoSeparator         *root;
   SoCoordinate3       *coord;
   SoNormal            *norm;
   SoBaseColor         *color;
   SoFaceSet           *faceSet;

   fprintf(stderr, "Reading a tri list\n");

   // Read number of words in data
   if (fread(&numWords, sizeof(long), 1, file) != 1)
      return NULL;

   // There are 27 words (3 vertices of 9 words each) per
   // triangle
   numTris = (int) numWords / 27;

   // Create nodes to hold all the vertex and shape info
   coord = new SoCoordinate3;
   norm  = new SoNormal;
   color = new SoBaseColor;
   faceSet  = new SoFaceSet;

   // Because we know how many vertices there are, we can
   // make the appropriate amount of room in the fields of
   // the nodes.
   coord->point.setNum(3 * numTris);
   color->rgb.setNum(3 * numTris);
   norm->vector.setNum(3 * numTris);
   faceSet->numVertices.setNum(numTris);

   // Process each triangle
   for (tri = 0; tri < numTris; tri++) {

      // Read 3 vertices
      if (! readVertices(3, verts, file))
         return NULL;

      // Store vertex info in fields
      for (vert = 0; vert < 3; vert++) {

         // Get index into appropriate place in field and
         // vertex
         fieldIndex = 3 * tri + vert;
```

```
            vertIndex  = 9 * vert;

            norm->vector.set1Value(fieldIndex,
                     &verts[vertIndex + 0]);
            color->rgb.set1Value(fieldIndex,
                     &verts[vertIndex + 3]);
            coord->point.set1Value(fieldIndex,
                     &verts[vertIndex + 6]);
         }

         // Store number of vertices of triangle
         faceSet->numVertices.set1Value(tri, 3);
      }

      // Create a root separator to hold the subgraph. We don't
      // need to ref() it because we aren't doing anything to
      // the subgraph until it is added to the main graph
      // (after this returns).
      root = new SoSeparator(4);

      // Add the nodes to the root
      root->addChild(coord);
      root->addChild(norm);
      root->addChild(color);
      root->addChild(faceSet);

      return root;
}

//////////////////////////////////////////////////////////////
// Checks color values for correct range.
//////////////////////////////////////////////////////////////

static void
verifyColors(SoNode *root)
{
      SoBaseColor         *color;
      int                 i, j;
      SoPath              *path;
      SoPathList          paths;
      float               r, g, b;
      const SbColor       *rgb;
      SoSearchAction      sa;

      sa.setType(SoBaseColor::getClassTypeId());
      sa.setInterest(SoSearchAction::ALL);
      sa.apply(root);
      paths = sa.getPaths();
```

```
   for (i = 0; i < paths.getLength(); i++) {
      path = paths[i];
      color = (SoBaseColor *) path->getTail();
      rgb = color->rgb.getValues(0);
      for (j = 0; j < color->rgb.getNum(); j++) {
         rgb[j].getValue(r, g, b);
         if (r < 0.0)
            r = 0.0;
         else if (r > 1.0)
            r = 1.0;
         if (g < 0.0)
            g = 0.0;
         else if (g > 1.0)
            g = 1.0;
         if (b < 0.0)
            b = 0.0;
         else if (b > 1.0)
            b = 1.0;
         color->rgb.set1Value(j, r, g, b);
      }
   }
}

//////////////////////////////////////////////////////////////
// Mainline.
//////////////////////////////////////////////////////////////

main(int argc, char *argv[])
{
   SbBool              endFound = FALSE;
   FILE                *file;
   SoSeparator         *root;
   SoMaterialBinding   *mtlBind;
   SoNormalBinding     *normBind;
   SoNode              *obj;

   // Check command line syntax and open sgo file
   if (argc != 2) {
      fprintf(stderr,
                "usage: SgoToIv file.sgo [> file.iv]\n");
      exit(1);
   }
   if ((file = fopen(argv[1], "r")) == NULL)
      error("SGO file could not be opened");

   // Initialize the Inventor database
   SoDB::init();
```

```
// Read header of SGO file and check it for validity
if (! readHeader(file))
   error("Invalid SGO header");

// Set up a root group to add all the objects to
root = new SoSeparator;
root->ref();

// Set up per-vertex material and normal bindings
mtlBind = new SoMaterialBinding;
normBind = new SoNormalBinding;
mtlBind->value = SoMaterialBinding::PER_VERTEX;
normBind->value = SoNormalBinding::PER_VERTEX;
root->addChild(mtlBind);
root->addChild(normBind);

// Keep reading objects until we are done or we have an
// error
while (! endFound) {

   SbBool readError = FALSE;

   switch (readObjectType(file)) {

     case SGO_OBJ_QUADLIST:
      obj = readQuadList(file);
      if (obj == NULL)
         readError = TRUE;
      break;

     case SGO_OBJ_TRILIST:
      obj = readTriList(file);
      if (obj == NULL)
         readError = TRUE;
      break;

     case SGO_OBJ_TRIMESH:
      // If triangle mesh is to be implemented,
      // include the code here.
      // Otherwise, print an error message
      fprintf(stderr, "SgoToIv: Can't process triangle
                  meshes\n");
      readError = TRUE;
      break;

     case SGO_OBJ_END:
      endFound = TRUE;
      break;
```

```
            default:
             error("Missing or invalid object type");
             readError = TRUE;
             break;
          }

          if (! endFound) {
             if (readError)
                error("Bad object data");
             else if (obj != NULL)
                root->addChild(obj);
          }
       }

       // Verify color values
       verifyColors(root);

       // Write out the resulting graph
       SoWriteAction wa;
       wa.apply(root);

       fprintf(stderr, "sgo->iv conversion done.\n");

       return 0;
    }
```

Sample Results

The following code shows the results of using the sample program to translate an SGO file with one trilist object into Inventor file format.

```
# Inventor V2.0 ascii
# greetings.iv (edited down)
#

Separator {
   MaterialBinding {
      value   PER_VERTEX
   }
   NormalBinding {
      value   PER_VERTEX
   }
   Separator {
      Coordinate3 {
         point       [ -0.455535 0.0715609 0,
                       -0.456693 0.0780454 0,
```

```
                        ...
                        0.465511 0.00737759 0.0567392 ]
      }
      Normal {
         vector      [ 0 0 -1,
                       0 0 -1,
                       ...
                       -0.155748 -0.689741 0.707107 ]
      }
      BaseColor {
         rgb         [ 0 0.0715609 0,
                       0 0.0780454 0,
                       ...
                       0.465511 0.00737759 0.0567392 ]
      }
      FaceSet {
         numVertices [ 3, 3, 3, 3, 3, 3, 3, 3,
                       ...
                       3, 3, 3, 3, 3, 3, 3, 3 ]
      }
   }
}
```

Using the File Translator in Another Program

The following excerpt shows how you could read another file format from within an Inventor application using an external translator program.

The program that translates the file into Inventor format is named **SgoToIv**. The file being translated is *myfile.sgo*. The translator program writes to *stdout* which has been piped via **popen()** to *fp*. Inventor's file reader is set to read from *fp*.

```
FILE *fp = popen("SgoToIv myfile.sgo", "r");

SoNode *root;
SoInput in;
in.setFilePointer(fp);
if (! SoDB::read(&in, root))
   fprintf(stderr, "Read error\n");
in.closeFile();

pclose(fp);
```

Tips and Guidelines

This section presents tips for testing the results of your translator program, for creating efficient scene graphs, and for verifying the file translation. It also suggests guidelines for writing an Inventor file translator.

Tips

The following sections offer general tips for writing an Inventor file translator.

Testing the Results

One way to test the Inventor file produced by your file translator is to perform a read test using the *ivcat* command:

```
ivcat filename.iv
```

This command prints the specified Inventor file in ASCII to `stdout`. If there are any syntax errors in the file, *ivcat* prints error messages for them. Use the -b option to print the specified Inventor file in binary to *stdout*.

As the saying goes, “Seeing is believing.” Another way to test the results of your file translator program is to use the *ivview* application to read your new Inventor scene graph and display the results. To use it type:

```
ivview filename.iv
```

Creating an Efficient Scene Graph

You can use **ivquicken** to improve the rendering performance of the scene graphs created by your translator program. See the reference page for **ivquicken** to learn more about this utility program.

You can also attempt to increase performance by having your program make a second pass through the database, condensing redundant nodes into fewer nodes. For example:

- If a number of nodes share the same material, for example, you can insert a material node in the scene graph so that multiple nodes will inherit the same material value. In the **SgoToIv** example program, if all the vertices of an object have the same color, you can isolate the material and use OVERALL material binding.
- If you are changing only the diffuse color attribute, use an Inventor **SoBaseColor** node rather than an **SoMaterial** node (as shown in the example program **SgoToIv**).
- If you know that a shape is solid or has ordered vertices, specify this information in an **SoShapeHints** node. In general, the more information you specify with **SoShapeHints**, the faster the rendering speed. The exception to this rule is that when you specify ordered vertices, but not a solid shape, rendering may be slower because two-sided lighting is automatically turned on and backface culling is turned off.

Verifying Values

You may also need to check the resulting Inventor scene graph to be sure that all values fall into the appropriate range. Many SGO files, for example, have color values that are out of range. You might want to add code that resets colors to valid values. The **verifyColors()** function in the "Complete Sample Program" in Chapter 2 provides an example of this. This code uses an Inventor search action to locate the base color nodes in the scene graph. It obtains the number of values in the base color node, loops through the values, and checks them. If the value is out of range, it resets the value.

If your translator generates nodes other than base color that have color fields (such as lights and materials), make sure their values are valid as well.

Automatic Normal Generation

If your data does not contain normals, Inventor can generate them automatically during rendering (but not in the file format). Inventor generates normals automatically if DEFAULT normal binding is used and you do not specify any normals. You can also use **ivquicken** to generate normals.

Guidelines for Writing an Inventor File Translator

The following guidelines are suggested so that applications can access Inventor translators in a consistent manner. These guidelines include

- Inventor file suffix
- application name and command line syntax
- general conventions for error handling
- manual page

File Suffix

It is recommended that *.iv* be used as the filename extension for Inventor files.

Application Name and Command Line Syntax

The recommended name for the translator application is

Xxx`ToIv` or `IvTo`*Xxx*

where *Xxx* represents the name of the non-Inventor file format and *Iv* represents Inventor file format.

The command line syntax is:

```
XxxToIv    filename     [ > filename.out]
```

where *filename* is the name of the file to be translated. By default, the output is directed to `stdout`. If desired, *filename* can default to *stdin*. However, be sure to implement the explicit *filename* option to guarantee compatibility with other file translators.

Error Handling

The application should return 0 if there are no errors. It should return 1 (or any other nonzero code) if errors occur. Error messages should be sent to *stderr*.

Manual Page

Write a manual page for the translator program, in standard UNIX man page format. Here is an example for the **SgoToIv** program.

```
SgoToIv(1)       Silicon Graphics                SgoToIv(1)

NAME
  SgoToIv - translates an SGO object file to an Inventor file

SYNOPSIS
  SgoToIv file.sgo [> file.iv]

DESCRIPTION
  SgoToIv reads a single SGO file, translates to Inventor,
  and writes the result to stdout.

NOTES
  SgoToIv does not handle SGO triangle meshes.

Page 1             Release 1.0               February 1994
```

Conventions Used in Inventor Files

The following conventions are used by the Inventor file format.

- The meter is the default unit for all data. (Use the **SoUnits** node to scale to other units.)
- The positive *y* axis points up. The positive *z* axis extends towards the viewer's eye (out of the screen).
- Colors are expressed as red, green, blue values.
- All fields within nodes have default values. See the *Open Inventor Nodes Quick Reference* for a list of these values.
- The vertices of polygons in vertex-based shapes should be specified consistently in either clockwise or counter-clockwise order.