How to Write an IRIS Inventor™ File Translator

Release 1.0

Copyright © 1992, Silicon Graphics, Inc. All rights reserved.

Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks, and Inventor, Graphics Library, and Showcase are trademarks of Silicon Graphics. UNIX is a registered trademark of UNIX System Laboratories.

Table of Contents

Section 1 Introduction 1

1.1 What Is in This Document? 1

1.2 Things You Need to Know About Inventor 1

1.2.1 Scene Database 1
1.2.2 Nodes and Fields 2
1.2.3 Node Icons 2
1.2.4 Scene Graphs 3
1.2.5 Group Nodes 3

Section 2 Translating Files into Inventor Format 4

2.1 General Steps 4

2.2 SGO File Format 4

2.2.1 SGO Quadrilateral List 5
2.2.2 SGO Triangle List 6
2.2.3 SGO Triangle Mesh 6

2.3 Reading the File Header 7

2.4 Initializing the Inventor Database 8

2.4.1 Initializing the Database 8
2.4.2 Creating the Root Node 8
2.4.3 Setting Up Default Attributes 8

2.5 Entering the Object Read Loop 9

2.6 Writing the Database to a File 12

2.7 Complete Sample Program: Translating from SGO to Inventor 12

2.8 Sample Results 21

2.9 Using the File Translator in Another Program 21

2.10 Alternate Method Using printf() 22

Section 3 Translating from Inventor to Other Formats 29

3.1 Translating from Inventor to Your Format 29

3.1.1 Reading a File into the Database 29
3.1.2 Traversing the Database 29

3.2 Traversing the Database in Order 29

3.2.1 Using the Callback Action for Automatic Traversal 30
3.2.2 Traversing the Database Manually 32

3.3 Using the Search Action 34

3.3.1 Example: Translating from Inventor to SGO 34

Section 4 Tips and Guidelines 41

4.1 Tips 41

4.1.1 Testing the Results 41
4.1.2 Creating an Efficient Scene Graph 41
4.1.3 Verifying Values 42
4.1.4 Automatic Normal Generation 42

4.2 Guidelines for Writing an Inventor File Translator 43

4.2.1 File Suffix 43
4.2.2 Application Name and Command Line Syntax 43
4.2.3 Error Handling 43
4.2.4 Manual Page 44

4.3 Conventions Used in Inventor Files 44

Section 5 File Format Syntax 45

5.1 File Header 45

5.2 Writing a Node 46

5.3 Writing Values within a Field 46

5.4 Ignore Flag 48

5.5 Sample Scene Graph and Inventor File 49

5.6 Instancing 50

5.7 Including Other Files 51

5.8 ASCII and Binary Versions 51

Section 1 Introduction

An important feature in IRIS 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.

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 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.

1.1 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 IRIS 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
- Details about Inventor's file format syntax

For a more complete description of Inventor objects, creating a scene database, and applying actions, see the *Inventor Programming Guide, Volume I: Using the Toolkit*, Chapters 1 through 5 and Chapter 9.

1.2 Things You Need to Know about Inventor

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

1.2.1 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.

1.2.2 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:

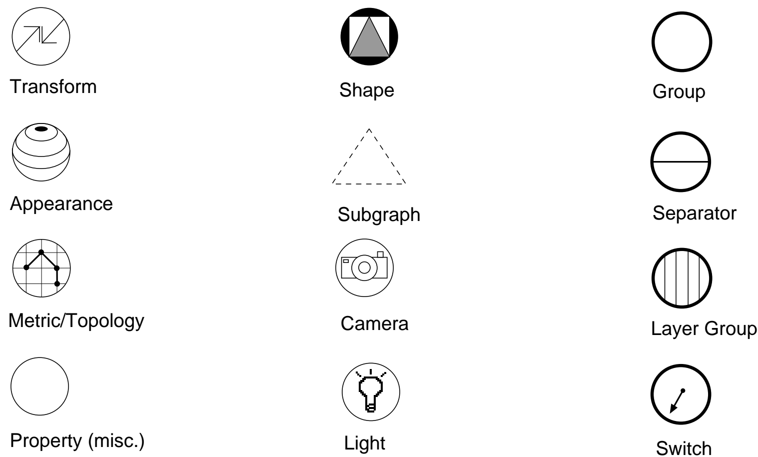
Field	Default Value	Meaning
`intensity`	1.0	value between 0.0 and 1.0 (maximum illumination)
`color`	1.0 1.0 1.0	red, green, blue color of light
`location`	0.0 0.0 1.0	position in *x*, *y*, *z*

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

1.2.3 Node Icons

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

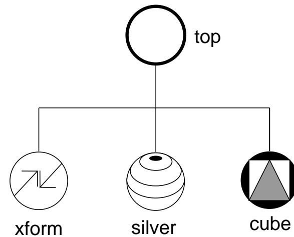
Figure 1 Node icons



1.2.4 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 2 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 2 Simple scene graph

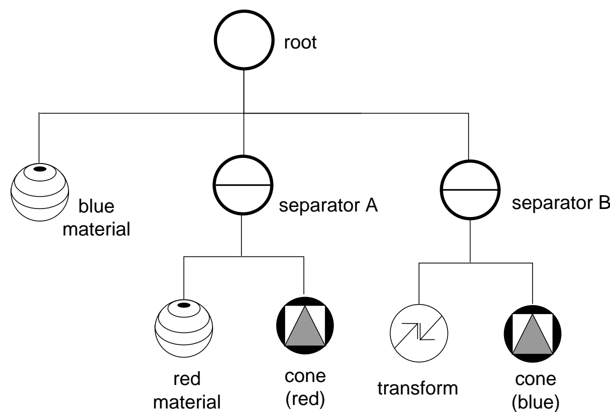


1.2.5 Group Nodes

A group node contains an ordered list of children that are traversed from left to right. A *separator* group, shown in Figure 3, is a special type of group node. Nodes under the separator group do not affect nodes in the graph after the separator.

When a scene database is written out, 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 3; the red material in the separator A group does not affect the cube in the separator B group). The root node of a scene graph is usually a separator.

Figure 3 Example of separator groups



Section 2 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 IRIS 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. Section 3 describes translating from Inventor format to other formats.

2.1 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 Section 2.7. An alternate version using `printf()` is shown in Section 2.10.

2.2 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. For more information on the SGO file format, see the *IRIS Showcase User's Guide*, Appendix C.

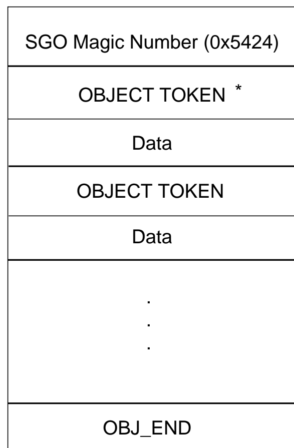
Figure 4 shows the basic structure of the SGO file format. The first word in the file must be the SGO magic number, a code number used to identify the file (0x5424). The magic number is 4 bytes long.

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.

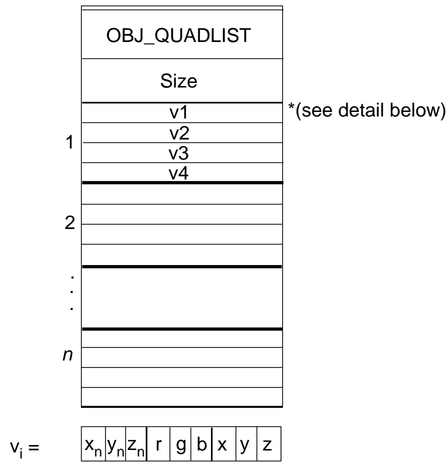
Figure 4 SGO file format



2.2.1 SGO Quadrilateral List

Figure 5 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 5, 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.

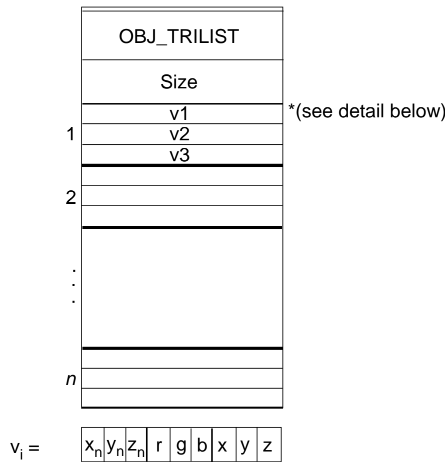
Figure 5 SGO quadrilateral list object



2.2.2 SGO Triangle List

Figure 6 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 6, 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 6 SGO triangle list object



2.2.3 SGO Triangle Mesh

The SGO triangle mesh object is the most complicated of the three SGO object data types. As shown in Figure 7, the object begins with the object token, followed by the size (in 4-byte words) of the data for this object. The next word is a *long* containing the number of words required for the object's vertex data. The data for a triangle mesh object can be divided into

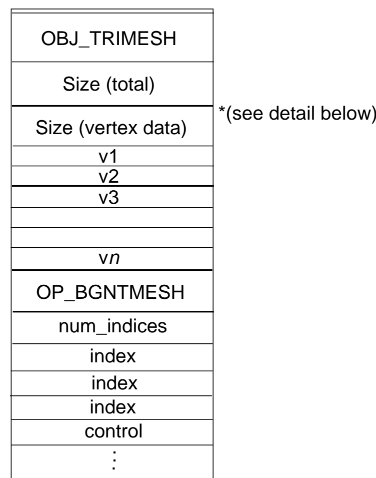
two parts. The first part of the data consists of a set of vertex data (in the same nine-word format as that of the other SGO objects). The second part of the data consists of a set of mesh control commands indicating how the vertices are connected.

The mesh controls (one word each, of type *long*) are

```
OP_BGNTMESH          (= 1)
OP_SWAPTMESH         (= 2)
OP_ENDBGNTMESH       (= 3)
OP_ENDTMESH          (= 4)
```

Each control command is followed by a *long* indicating how many vertex indices follow. For more information on triangle meshes, see the *Graphics Library Programming Guide*.

Figure 7 SGO triangle mesh object



2.3 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);
}
```

2.4 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

2.4.1 Initializing the Database

The following code initializes the Inventor database:

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

2.4.2 Creating the Root Node

The root node for the database is a separator node (see Section 1.2.5). 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();
```

2.4.3 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

The SGO triangle mesh object presents a special case because it lists colors and normals in an arbitrary order and then indexes into the list. For this object, we need to 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). The sample program conveniently uses per vertex indexed binding for all objects. For objects that do *not* index into the material and normal lists, such as Inventor face sets, the binding simply reverts to per vertex binding.

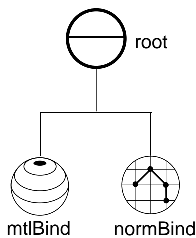
See the *IRIS Inventor Programming Guide*, Volume I, Chapter 5, for a detailed description of binding nodes.

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_INDEXED;
normBind->value = SoNormalBinding::PER_VERTEX_INDEXED;
root->addChild(mtlBind);
root->addChild(normBind);
```

At this point, the Inventor database looks like Figure 8.

Figure 8 Nodes created during initialization



2.5 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 three SGO objects are translated into the Inventor shape nodes shown at the right:

```
OBJ_QUADLIST  ──────▶  SoFaceSet
OBJ_TRILIST   ──────▶  SoFaceSet
OBJ_TRIMESH   ──────▶  SoIndexedTriangleMesh
```

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 9. 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 9.

The `readTriMesh()` function in the sample program translates an SGO triangle mesh into the group of nodes shown in Figure 10. The Inventor `SoIndexedTriangleMesh` shape node was chosen since it closely matches the SGO triangle mesh object.

Figure 9 Inventor nodes for a face set object

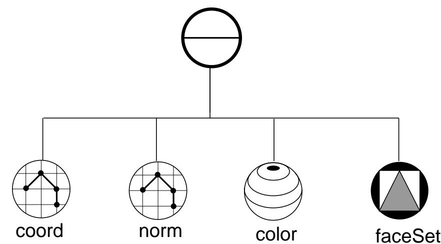
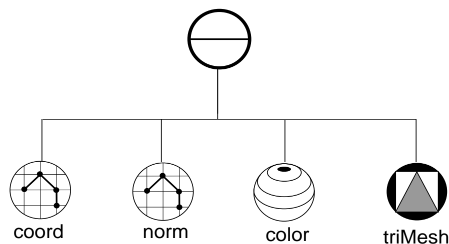


Figure 10 Inventor nodes for an indexed triangle mesh object



Each group of nodes (called a *subgraph*) is added to the database. Figure 11 shows the scene graph after reading the first SGO object. Figure 12 shows the scene graph after reading a second SGO object.

The sample program continues reading SGO objects and creating subgraphs of Inventor nodes until all SGO objects have been translated and added to the Inventor scene database.

Figure 11 Inventor database after reading one SGO object (for example, TRILIST)

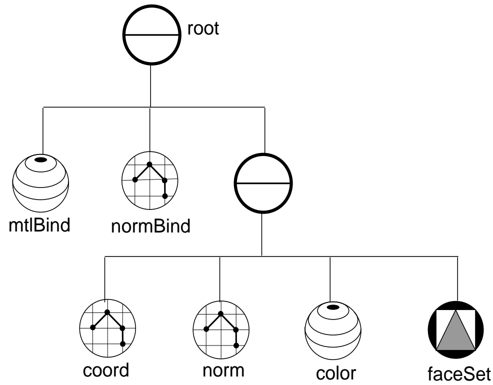
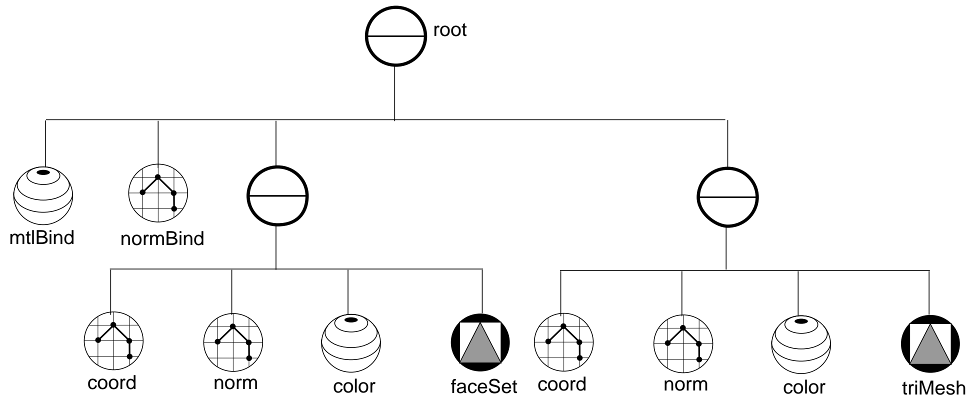


Figure 12 Inventor database after reading two SGO objects (for example, TRILIST and TRIMESH)



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:
         obj = readTriMesh();
         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);
  }
 }
```

2.6 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 Section 4 for ways to test the resulting Inventor file.

2.7 Complete Sample Program: Translating from SGO to Inventor

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

```
// Description:
//
// Program to convert IRIS Showcase's SGO (Silicon Graphics Object)
// data files into IRIS Inventor files. Each SGO object (quadrilateral
// list, trilist, or triangle mesh) is read from file and is converted
// into a subgraph rooted by a Separator. The subgraph contains a
// Coordinate3 node, a Normal node, a BaseColor node,
// and a shape node. The shape node is either a FaceSet (for
// quadrilateral and triangle lists) or an IndexedTriangleMesh (for
// triangle meshes). All of the subgraphs are added under a single
// Separator root. The resulting graph is written to stdout.
//
// 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/SoIndexedTriangleMesh.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 */

#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;

   // 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.insertSpace(0, 4 * numQuads - 1);
   color->rgb.insertSpace(0, 4 * numQuads - 1);
   norm->vector.insertSpace(0, 4 * numQuads - 1);
   faceSet->numVertices.insertSpace(0, numQuads - 1);

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

      // Read 4 vertices (9 floats each)
      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.insertSpace(0, 3 * numTris - 1);
    color->rgb.insertSpace(0, 3 * numTris - 1);
    norm->vector.insertSpace(0, 3 * numTris - 1);
    faceSet->numVertices.insertSpace(0, numTris - 1);

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

       // Read 3 vertices (9 floats each)
       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;
}
```

```
///////////////////////////////////////////////////////////////////////
//
// Reads a triangle mesh 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 *
readTriMesh( FILE *file )
{
   long                 numWords, numVertWords;
   long                 controlData, numIndices, index;
   int                  numVerts, numControls, vert, control, ind, mc;
   float                verts[9];
   SoSeparator          *root;
   SoCoordinate3        *coord;
   SoNormal             *norm;
   SoBaseColor          *color;
   SoIndexedTriangleMesh   *triMesh;

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

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

   // Read number of words in vertex data (9 words each)
   if (fread(&numVertWords, sizeof(long), 1, file) != 1)
      return NULL;

   // There are 9 words per vertex
   numVerts = (int) numVertWords / 9;

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

   // Because we know how many vertices there are, we can make the
   // appropriate amount of room in the fields of the nodes.
   coord->point.insertSpace(0, numVerts - 1);
   color->rgb.insertSpace(0, numVerts - 1);
   norm->vector.insertSpace(0, numVerts - 1);

   // Process all vertices
   for (vert = 0; vert < numVerts; vert++) {

      // Read 1 vertex (9 floats)
      if (! readVertices(1, verts, file))
         return NULL;

      // Store vertex info in fields
      norm->vector.set1Value(vert, &verts[0]);
      color->rgb.set1Value(vert,   &verts[3]);
      coord->point.set1Value(vert, &verts[6]);
   }

   // The number of mesh control data words is the number of data
   // words left over (after the number of vertex words and vertex
   // data)
   numControls = (int) (numWords - 1 - numVertWords);

   // Insert room for controls in IndexedTriangleMesh. This is a
   // rough estimate
   triMesh->coordIndex.insertSpace(0, numVerts - 1);
```

```
    // Process mesh control data
    mc = 0;
    for (control = 0; control < numControls; control++) {

        // Read next control data word
        if (fread(&controlData, sizeof(long), 1, file) != 1)
            return NULL;

        switch (controlData) {

          case SGO_OP_BGNTMESH:
          case SGO_OP_ENDBGNTMESH:
          case SGO_OP_ENDTMESH:
           // The SO_END_MESH_INDEX handles all of these cases correctly
           if (mc > 0)
               triMesh->coordIndex.set1Value(mc++, SO_END_MESH_INDEX);
           break;

          case SGO_OP_SWAPTMESH:
           triMesh->coordIndex.set1Value(mc++, SO_SWAP_MESH_INDEX);
           break;

          default:
           // Error to get here
           return NULL;
        }

        // Read the number of vertex indices to follow
        if (fread(&numIndices, sizeof(long), 1, file) != 1)
            return NULL;

        // Read and store the indices. Each index is the byte offset
        // of the vertex, so we need to divide.
        for (ind = 0; ind < numIndices; ind++) {
            if (fread(&index, sizeof(long), 1, file) != 1)
            return NULL;
            triMesh->coordIndex.set1Value(mc++, index / 36);
        }

    // We skipped this many mesh control data words for indices
    control += 1 + (int) numIndices;
    }

    // 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(triMesh);

    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.setFindAll( TRUE );
   sa.apply(root);
   paths = sa.getPaths();

   for ( i = 0; i < paths.length(); 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 sgo.file [> iv.file]\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 material and normal bindings. We use per-vertex indexed
   // bindings because that's what's needed for the
   // IndexedTriangleMesh. The FaceSet will treat these bindings as
   // (non-indexed) per-vertex.
   mtlBind = new SoMaterialBinding;
   normBind = new SoNormalBinding;
   mtlBind->value = SoMaterialBinding::PER_VERTEX_INDEXED;
   normBind->value = SoNormalBinding::PER_VERTEX_INDEXED;
   root->addChild(mtlBind);
   root->addChild(normBind);

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

        case SGO_OBJ_QUADLIST:
         obj = readQuadList( file );
         break;

        case SGO_OBJ_TRILIST:
         obj = readTriList( file );
         break;

        case SGO_OBJ_TRIMESH:
         obj = readTriMesh( file );
         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);
      }
   }

   // Verify color values
   verifyColors( root );

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

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

   return 0;
}
```

2.8 Sample Results

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

```
#Inventor V1.0 ascii
# sgilogo.iv (edited down)
#
Separator {
   MaterialBinding {
      value   PER_VERTEX_INDEXED
   }
   NormalBinding {
      value   PER_VERTEX_INDEXED
   }
   Separator {
      Coordinate3 {
         point   [ 0 0 0,
                  -0.205 0.07 -0.16045,
                  ...
                  -0.16 -0.07045 -0.205 ]
      }
      Normal {
         vector   [ 0 0 0,
                    0 0 1,
                    ...
                    0 1 0 ]
      }
      BaseColor {
         rgb    [ 0 0 0,
                  0 0.07 0,
                  ...
                  0 0 0 ]
      }
      IndexedTriangleMesh {
         coordIndex  [ 1802, 1804, 1803, -2, 1805, 1806, 1807,
                       1808, 1809, 1810, 1811, 1812, 1813, 1814,
                       1815, 1816,
                       ...
                       28, 41, -1 ]
      }
   }
}
```

2.9 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);
```

2.10 Alternate Method Using printf()

The sample program in Section 2.7 uses the preferred method of creating an Inventor scene database and then writing out the database using Inventor's `SoWriteAction`. You can, however, also use `printf()` to print the Inventor ASCII format directly. This technique is more difficult to maintain and support than the method decribed in the first sample program. Any changes or improvements in Inventor's file format will be automatically supported by Inventor's `SoWriteAction`. But, if you use `printf`'s, you must pay close attention to future Inventor releases and add the changes and improvements to the file format yourself.

Here is an example of an SGO-to-Inventor file translator that uses `printf`'s to print the Inventor ASCII file format.

```
//
// Description:
//
// Program to convert SGO (Silicon Graphics Object) data files into
// IRIS Inventor files. Each SGO object (quadrilateral list, triangle
// list, or triangle mesh) is read from file and written out in
// Inventor ascii format.
// Each sgo object is written out as a subgraph rooted by a Separator.
// The subgraph contains a Coordinate3 node, a Normal node, a BaseColor
// node,and a shape node. The shape node is either a FaceSet (for
// quadrilateraland triangle lists) or an IndexedTriangleMesh (for
// triangle meshes).
// All of the subgraphs are contained under a single Separator root.
//
// usage SgoToIv file.sgo > file.iv
#include <stdio.h>
#include <malloc.h>

typedef int     Boolean;

// 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 */

#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 Boolean
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 Boolean
readVertices( FILE *file, int numVerts, float verts[])
{
   // Read vertices (9 floats each)
   return (fread(verts, sizeof(float), numVerts * 9, file) ==
      numVerts * 9);
}


//////////////////////////////////////////////////////////////////////
//
// Writes a vertex array out in Inventor format as coordinates,
// normals, and colors.
//
//////////////////////////////////////////////////////////////////////

static void
writeVertices(int numVerts, float verts[])
{
   int     vert, vertIndex;

   printf( "        Coordinate3 { point [\n" );
   for ( vert = 0; vert < numVerts; vert++ ) {
      vertIndex = vert * 9;
      printf( "\t\t\t\t %f %f %f,\n",
         verts[vertIndex+6], verts[vertIndex+7], verts[vertIndex+8] );
   }
   printf( "        ] }\n" );
```

```
   printf( "        Normal { vector [\n" );
   for ( vert = 0; vert < numVerts; vert++ ) {
      vertIndex = vert * 9;
      printf( " \t\t\t\t %f %f %f,\n",
         verts[vertIndex+0], verts[vertIndex+1], verts[vertIndex+2] );
   }
   printf( "        ] }\n" );

   printf( "        BaseColor { rgb [\n" );
   for ( vert = 0; vert < numVerts; vert++ ) {
      vertIndex = vert * 9;
      if ( verts[vertIndex+3] < 0.0 ) verts[vertIndex+3] = 0.0;
      else if ( verts[vertIndex+3] > 1.0 ) verts[vertIndex+3] = 1.0;
      if ( verts[vertIndex+4] < 0.0 ) verts[vertIndex+4] = 0.0;
      else if ( verts[vertIndex+4] > 1.0 ) verts[vertIndex+4] = 1.0;
      if ( verts[vertIndex+5] < 0.0 ) verts[vertIndex+5] = 0.0;
      else if ( verts[vertIndex+5] > 1.0 ) verts[vertIndex+5] = 1.0;
      printf( "\t\t\t\t %f %f %f,\n",
         verts[vertIndex+3], verts[vertIndex+4], verts[vertIndex+5] );
   }
   printf( "        ] }\n" );
}


//////////////////////////////////////////////////////////////////////
//
// 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 Boolean
readQuadList( FILE *file )
{
   long        numWords;
   int         numVerts, numQuads, quad;
   float       *verts;


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

   // Read number of words in data
   if (fread(&numWords, sizeof(long), 1, file) != 1)
      return 0;

   // There are 36 words (4 vertices of 9 words each) per quadrilateral
   numVerts = (int) numWords / 9;
   numQuads = (int) numWords / 36;
   verts = (float *) malloc( numVerts * 9 * sizeof (float) );

   if ( !readVertices(file, numVerts, verts))
      return 0;

   writeVertices( numVerts, verts );

   // Write out face connectivity of quads
   printf( "        FaceSet { numVertices [\n" );
   for ( quad = 0; quad < numQuads; quad++ ) {
      printf( "                                 4,\n" );
   }
   printf( "        ] }\n" );

   free( verts );
   return 1;
}
```

```
////////////////////////////////////////////////////////////////////////
//
// 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 Boolean
readTriList( FILE *file )
{
   long        numWords;
   int         numVerts, numTris, tri;
   float       *verts;


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

   // Read number of words in data
   if (fread(&numWords, sizeof(long), 1, file) != 1)
      return 0;

   // There are 27 words (3 vertices of 9 words each) per tri
   numVerts = (int) numWords / 9;
   numTris = (int) numWords / 27;
   verts = (float *) malloc( numVerts * 9 * sizeof (float) );

   if ( !readVertices(file, numVerts, verts))
      return 0;

   writeVertices( numVerts, verts );

   // Write out face connectivity of tris
   printf( "        FaceSet { numVertices [\n" );
   for ( tri = 0; tri < numTris; tri++ ) {
      printf( "\t\t\t\t  3,\n" );
   }
   printf( "        ] }\n" );

   free( verts );
   return 1;
}


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

static Boolean
readTriMesh( FILE *file )
{
   long        numWords, numVertWords;
   long        controlData, numIndices, index;
   int         numVerts, numControls, vert, control, ind, mc;
   float       *verts;


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

   // Read number of words in data
   if (fread(&numWords, sizeof(long), 1, file) != 1)
      return 0;
```

```
   // Read number of words in vertex data (9 words each)
   if (fread(&numVertWords, sizeof(long), 1, file) != 1)
      return 0;

   // There are 9 words per vertex
   numVerts = (int) numVertWords / 9;
   verts = (float *) malloc( numVerts * 9 * sizeof (float) );

   if ( !readVertices(file, numVerts, verts))
      return 0;

   writeVertices( numVerts, verts );
   printf( "        IndexedTriangleMesh { coordIndex [\n" );

   // The number of mesh control data words is the number of data
   // words left over (after the number of vertex words and vertex
   // data)
   numControls = (int) (numWords - 1 - numVertWords);

   // Process mesh control data
   mc = 0;
   for (control = 0; control < numControls; control++) {

      // Read next control data word
      if (fread(&controlData, sizeof(long), 1, file) != 1)
         return 0;

      switch (controlData) {

        case SGO_OP_BGNTMESH:
        case SGO_OP_ENDBGNTMESH:
        case SGO_OP_ENDTMESH:
         // The SO_END_MESH_INDEX handles all of these cases correctly
         if (mc > 0) {
         printf( " \t\t\t\t  -1,\n" );
         mc++;
         }
         break;

        case SGO_OP_SWAPTMESH:
         printf( " \t\t\t\t  -2,\n" );
         mc++;
         break;

        default:
         // Error to get here
         return 0;
      }

      // Read the number of vertex indices to follow
      if (fread(&numIndices, sizeof(long), 1, file) != 1)
         return 0;

      // Read and store the indices. Each index is the byte offset
      // of the vertex, so we need to divide.
      for (ind = 0; ind < numIndices; ind++) {
         if (fread(&index, sizeof(long), 1, file) != 1)
            return 0;
         printf( "\t\t\t\t  %d,\n", index/36 );
      }

   // We skipped this many mesh control data words for indices
   control += 1 + (int) numIndices;
   }
   printf( "        ] }\n" );

   return 1;
}
```

```
//////////////////////////////////////////////////////////////////////
//
// Mainline.
//
//////////////////////////////////////////////////////////////////////

main( int argc, char *argv[] )
{
   FILE    *file;
   Boolean     endFound = 0;

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

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

   // Write Inventor header and root separator group
   printf( "#Inventor V1.0 ascii\n" );
   printf( "Separator {\n" );

   // Set up material and normal bindings. We use per-vertex indexed
   // bindings because that's what's needed for the
   // IndexedTriangleMesh. The FaceSet will treat these bindings as
   // (non-indexed) per-vertex.

   printf( "    MaterialBinding { value PER_VERTEX_INDEXED }\n" );
   printf( "    NormalBinding { value PER_VERTEX_INDEXED }\n" );

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

      switch (readObjectType( file )) {

        case SGO_OBJ_QUADLIST:
         printf( "    Separator {\n" );
         if (! readQuadList( file ))
            error( "Bad QuadList data" );
         break;

        case SGO_OBJ_TRILIST:
         printf( "    Separator {\n" );
         if (! readTriList( file ))
            error( "Bad TriList data" );
         break;

        case SGO_OBJ_TRIMESH:
         printf( "    Separator {\n" );
         if (! readTriMesh( file ))
            error( "Bad TriMesh data" );
         break;

        case SGO_OBJ_END:
         endFound = 1;
         break;

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

```
   if (! endFound )
      printf( "    }\n" );
   }
   printf( "}\n" );
   fprintf(stderr, "sgo->iv conversion done.\n");

   return 0;
}
```

Section 3 Translating from Inventor to Other Formats

This section describes translating from Inventor to other formats. The sample program translates IRIS Inventor files into SGO files.

3.1 Translating from Inventor to Your Format

Translating an Inventor file into your file format involves two steps:

1. Read in the Inventor database.
2. Traverse the Inventor database and write out the node information in your format. Sections 3.2 and 3.3 outline several basic approaches to traversing the database.

3.1.1 Reading a File into the Database

You can read an Inventor scene graph from a file into the scene database using the `read()` method. `SoDB` reads a scene graph from the file specified by the given `SoInput` and returns a pointer to the resulting root node in `readRoot`. In this excerpt, the name of the file is `myFile.iv`.

```
SoInput          myFile;
SoNode           *readRoot;

myFile.openFile( "myFile.iv" );    // returns FALSE on error
SoDB::read( &myFile, readRoot );   // returns TRUE or FALSE
myFile.closeFile();
```

3.1.2 Traversing the Database

There are two general approaches to translating from Inventor file format to other formats:

- If your format correlates closely with Inventor objects and database structure, traverse the database in order, converting each node to your format.
- If your format uses only a limited set of Inventor objects, use the search action (`SoSearchAction`) to find specific object types in the Inventor file and then write out those objects.

Each of these methods is described in more detail in the following sections.

3.2 Traversing the Database in Order

If your format has equivalents for a variety of Inventor objects, you will probably use the method of traversing the Inventor database in order, translating one node at a time into your format. In some cases, you will need to collect objects and write the information out later. For example, Inventor separates coordinates (`SoCoordinate3`, and so on) from polygon objects

(SoFaceSet, and so on). You may need to collect the coordinate, normal, color, and polygon information and then combine them in your format.

If your format supports *instancing* (multiple references to a single object), you'll need to keep track of how many times an object has been traversed and maintain instancing in your format.

3.2.1 Using the Callback Action for Automatic Traversal

There are several methods for traversing the database. The callback action, SoCallbackAction, provides automatic traversal of the database in the correct order. It invokes your callback function at each node. See the SoCallbackAction man page for more information.

The following code fragment illustrates the use of the SoCallbackAction to help translate an Inventor database to an external file format:

```
//
// usage: IvToExt file.iv > file.ext
//

#include <stdio.h>
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/nodes/SoGroup.h>

extern void  writeLeafNode( SoNode * );

//////////////////////////////////////////////////////////////////////
////
//
// This routine writes out the data for the given leaf node if it
// is recognizable by the external data file format.
//
//////////////////////////////////////////////////////////////////////
////
void
writeLeafNode( SoNode *node )
{
   if ( node->getTypeId() == SoCoordinate3::getClassTypeId() ) {

      // Get the coordinates and write them into the file
      SoCoordinate3 *coord = (SoCoordinate3 *)node;
      const SbVec3f *pts   = coord->point.getValues(0);
      long numCoords       = coord->point.getNum();

      // ...
   }
```

```
   else if ( node->getTypeId() == SoMaterial::getClassTypeId() ) {

      // Get the material and write it into the file
      SoMaterial *mtl            = (SoMaterial *)node;
      const SbColor &ambient     = mtl->ambientColor[0];
      const SbColor &diffuse     = mtl->diffuseColor[0];
      const SbColor &specular    = mtl->specularColor[0];
      const SbColor &emission    = mtl->emissiveColor[0];
      float shininess            = mtl->shininess[0];
      float transparency         = mtl->transparency[0];

      // ...
   }

   // Continue for all node types you are interested in.
   // ...

}

//////////////////////////////////////////////////////////////////////
////
//
// This callback routine checks the node to see if it is a group node
// or a leaf node.  For group nodes, hierarchy information may be
// written to the external format.  For leaf nodes, write them out.
//
//////////////////////////////////////////////////////////////////////
////

static SoCBResponse
callbackRoutine( void *data, SoCallbackAction *act,
                 const SoNode *node )
{
   // If the node is a group node, you might want to write out any
   // data relating to the hierarchy.

   if ( node->isOfType( SoGroup::getClassTypeId() ) ) {

        // ...
   }
   else
      writeLeafNode( (SoNode *)node );

   // Return status telling action to continue traversing the database.
   return SoCallbackAction::CONTINUE;
}

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

main( int argc, char *argv[] )
{
   SoNode       *inputRoot = NULL;
   SoInput      inputFile;

   if( argc != 2 ) {
      fprintf( stderr, "usage: IvToExt file.iv [> file.ext]\n" );
      exit( 1 );
   }
```

```
    SoDB::init();

    // Open the Inventor file and read in the database
    inputFile.openFile( argv[1] );
    SoDB::read( &inputFile, inputRoot );
    if( inputRoot == NULL ) {
        fprintf( stderr, "Error reading %s\n", argv[1] );
        exit( 1 );
    }
    inputRoot->ref();
    inputFile.closeFile();

    // Write any header information into the output file
    // ...

    // Traverse the database using the SoCallbackAction.  Register a
    // callback routine which will be called each time a node is
    // encountered during traversal.
    SoCallbackAction cbAct;

    cbAct.addPostCallback( SoNode::getClassTypeId(), callbackRoutine,
                           NULL );
    cbAct.apply( inputRoot );

    return 0;
}
```

3.2.2 Traversing the Database Manually

An alternative is to traverse the database manually and query each node whether it is a group node:

```
if ( node->isOfType( SoGroup::getClassTypeId() ) ) {...
```

If the node is a group, find out how many children are in the group (with `getNumChildren()`) and then traverse each child.

This method allows you to traverse every node in the database. The callback action traverses only the active nodes (for example, it might only traverse selected children of a switch node).

The following code fragment illustrates traversing the database manually to translate an Inventor database to some external file format. It uses the `writeLeafNode()` routine from the code fragment in the previous section.

```
// usage: IvToExt file.iv > file.ext

#include <stdio.h>
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoSwitch.h>
#include <Inventor/nodes/SoArray.h>

extern void  writeLeafNode( SoNode * );
```

```
/////////////////////////////////////////////////////////////////////
////
//
// This routine recursively traverses the database rooted by the given
// node.  If the node is a group node of some type, the routine is
// called again for the children.  If the node is a leaf, write out the
// pertinent data to stdout.
//
/////////////////////////////////////////////////////////////////////
////
static void
traverseWrite( SoNode *node )
{

   // If the node is a group node, traverse its children and
   // write them out.  For each type of group, you may have
   // to add additional code to correctly convert the hierarchy
   // information to your external format

   if ( node->isOfType( SoGroup::getClassTypeId() ) ) {

      if (node->getTypeId() == SoGroup::getClassTypeId() ) {

         // Traverse each child.  No additional operations are
         // performed by the group node.
         SoGroup *group = (SoGroup *)node;

          for (int child = 0; child < group->getNumChildren(); child++)
               traverseWrite( group->getChild( child ) );
      }
      else if (node->getTypeId() == SoSeparator::getClassTypeId() ) {

         // Inventor graphics state is pushed before visiting the
         // children and popped after all of the children have been
         // visited.  Make sure either your external format performs
         // the same push and pop, or you manually push and pop the
         // attributes during translation of the file format.

         // ...
      }
      else if (node->getTypeId() == SoSwitch::getClassTypeId() ) {

         // If your format recognizes levels of detail, then you
         // should traverse all of the children.  Otherwise, just
         // traverse the active child, as shown here.
          int child = (int)(((SoSwitch *)node)->whichChild.getValue());

           if ( child >= 0 )
                traverseWrite( ((SoGroup *)node)->getChild( child ) );
      }
      else if (node->getTypeId() == SoArray::getClassTypeId() ) {

         // The children are repeated several times, offset by a
         // certain amount.  You must manually keep track of the
         // offset for each set of children
         SoArray *array = (SoArray *)node;
         int i, j, k;

         for ( i = 0; i < array->numElements1.getValue(); i++ )
             for ( j = 0; j < array->numElements2.getValue(); j++ )
                for ( k = 0; k < array->numElements3.getValue(); k++ ){

                   // Save separation distances and traverse children
                   // ...
                }
       }

       // Continue for other types of group nodes.
```

```
        // ...
    }

    // Check for any leaf nodes you are interested in converting to
    // your external format.

    else
       writeLeafNode( node );
}


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

main( int argc, char *argv[] )
{
    SoNode        *inputRoot = NULL;
    SoInput       inputFile;

    if( argc != 2 ) {
       fprintf( stderr, "usage: IvToExt file.iv [> file.ext]\n" );
       exit( 1 );
    }

    SoDB::init();

    // Open the Inventor file and read in the database
    inputFile.openFile( argv[1] );
    SoDB::read( &inputFile, inputRoot );
    if( inputRoot == NULL ) {
       fprintf( stderr, "Error reading %s\n", argv[1] );
       exit( 1 );
    }
    inputRoot->ref();
    inputFile.closeFile();

    // Write any header information into the output file
    // ...

    // Traverse the database using the traverseWrite routine.
    traverseWrite( inputRoot );

    return 0;
}
```

3.3 Using the Search Action

If your file format makes limited use of the available Inventor objects, use the `SoSearchAction` to find the specific object types in the Inventor file that your format supports. Then write out these objects, along with any related objects.

3.3.1 Example: Translating from Inventor to SGO

This approach is appropriate for translating files from Inventor to SGO format and is described in the following steps. Code for the translator follows this description.

In this case, you are searching the Inventor file for face sets and indexed triangle mesh objects only, since these are the only objects supported by the SGO file format. The basic procedure is

1. Search the Inventor file for all face sets and obtain a list of them.

   a. For each face set, search backwards in the file for the coordinates, colors, and normals corresponding to that face set.

   b. Use `getMatrixAction()` to compute the world space transformation matrix for the coordinates. Apply this matrix to the coordinates.

   c. Check the number of vertices in the face set.

   – If there are 4 vertices per face, write the object out as an SGO `OBJ_QUADLIST`.

   – If there are 3 vertices per face, write the object out as an SGO `OBJ_TRILIST`.

   In either case, check that the number of vertices is the *same* for all faces in the face set.

2. Search the Inventor file for all indexed triangle meshes. For each triangle mesh, complete steps *a* and *b*, above. Then write the object out as an SGO `OBJ_TRIMESH`.

Here is the code for the Inventor-to-SGO file translator:

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

#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/actions/SoSearchAction.h>
#include <Inventor/actions/SoGetMatrixAction.h>
#include <Inventor/nodes/SoNode.h>
#include <Inventor/nodes/SoBaseColor.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoIndexedTriangleMesh.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 */

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

// Globals
static    SoSearchAction        *search;
static    SoGetMatrixAction     *getMx;
```

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

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


///////////////////////////////////////////////////////////////////////
//
// Write a vertex list for an SGO object.
//
///////////////////////////////////////////////////////////////////////

static void
writeVertexList( int numVerts, SbVec3f *verts,
            const SbVec3f *norms, const SbColor *colors )
{
   int         i;
   long        numWords;
   float       vert[9];

   numWords = numVerts * 9;
   fwrite( &numWords, sizeof(long), 1, stdout );
   for ( i = 0; i < numVerts; i++ ) {
      verts[i].getValue( vert[6], vert[7], vert[8] );
      norms[i].getValue( vert[0], vert[1], vert[2] );
      colors[i].getValue( vert[3], vert[4], vert[5] );
      fwrite( vert, sizeof(float), 9, stdout );
   }

   return;
}

///////////////////////////////////////////////////////////////////////
//
// Write an SGO Triangle or Quad list object.
//
///////////////////////////////////////////////////////////////////////

void
writeTriQuadLists( SoPathList fsPaths )
{
   int                 i, code;
   SoPath              *path;
   SoPathList          paths;
   SbVec3f             *xpts = NULL, *xnorms = NULL;
   const SbVec3f       *pts, *norms;
   const SbColor       *colors;
   SbMatrix            mx;
   SbBool              valid = TRUE;
   SoCoordinate3       *coords;
   SoFaceSet           *faceSet;
   const long          *numVerts;

   for ( i = 0; i < fsPaths.length(); i++ ) {
      // For each Face Set found, find the coords, norms, and colors
      path = fsPaths[i];
      search->setType( SoCoordinate3::getClassTypeId() );
      search->apply( path );
      paths = search->getPaths();
      if ( paths.length() == 0 ) {
```

```
            fprintf( stderr, "No coords found, skip this face...\n" );
            continue;
         }
         else coords = (SoCoordinate3 *) paths[paths.length()-1]
                       ->getTail();

         search->setType( SoNormal::getClassTypeId() );
         search->apply( path );
         paths = search->getPaths();
         if ( paths.length() == 0 ) norms = NULL;
         else norms = ((SoNormal *) paths[paths.length()-1]->getTail())
                          ->vector.getValues(0);

         search->setType( SoBaseColor::getClassTypeId() );
         search->apply( path );
         paths = search->getPaths();
         if ( paths.length() == 0 ) colors = NULL;
         else colors = ((SoBaseColor *) paths[paths.length()-1]
                       ->getTail()) ->rgb.getValues(0);

        // Transform coordinates and normals by world space matrix.
         getMx->apply( path );
         mx = getMx->getMatrix();
         pts = coords->point.getValues(0);
         xpts = new SbVec3f[coords->point.getNum()];
         xnorms = new SbVec3f[coords->point.getNum()];

         for ( i = 0; i < coords->point.getNum(); i++ ) {
            mx.multVecMatrix( pts[i], xpts[i] );
            mx.multDirMatrix( norms[i], xnorms[i] );
         }

         faceSet = (SoFaceSet *) path->getTail();
         numVerts = faceSet->numVertices.getValues(0);
         if ( numVerts[0] == 3 ) {
           for ( i = 0; i < faceSet->numVertices.getNum() && valid;i++ ) {
               if ( numVerts[i] != 3 ) {
                  fprintf( stderr, "Invalid tri list found\n" );
                  valid = FALSE;
               }
            }
            if ( valid ) {
               code = SGO_OBJ_TRILIST;
               fwrite( &code, sizeof(long), 1, stdout );
               writeVertexList( coords->point.getNum(), xpts, xnorms,
                                colors);
            }
         }
         else if ( numVerts[0] == 4 ) {
           for ( i = 0; i < faceSet->numVertices.getNum() && valid;i++ ) {
               if ( numVerts[i] != 4 ) {
                  fprintf( stderr, "Invalid quad list found\n" );
                  valid = FALSE;
               }
            }
            if ( valid ) {
               code = SGO_OBJ_QUADLIST;
               fwrite( &code, sizeof(long), 1, stdout );
               writeVertexList( coords->point.getNum(), xpts, xnorms,
                                colors);
            }
         }
      }
      return;
   }
```

```
////////////////////////////////////////////////////////////////////
//
// Write an SGO Triangle Mesh object.
//
////////////////////////////////////////////////////////////////////

void
writeTriMeshes( SoPathList fsPaths )
{
   int                 i, j, code, size, num, numControls;
   SoPath              *path;
   SoPathList          paths;
   SbVec3f             *xpts = NULL, *xnorms = NULL;
   const SbVec3f       *pts, *norms;
   const SbColor       *colors;
   SbMatrix            mx;
   const long          *indices;
   long                control;
   SbBool              valid = TRUE;
   SoCoordinate3       *coords;
   SoIndexedTriangleMesh *triMesh;

   for ( i = 0; i < fsPaths.length(); i++ ) {
      // For each Tri Mesh found, find the coords, norms, and colors
      path = fsPaths[i];
      search->setType( SoCoordinate3::getClassTypeId() );
      search->apply( path );
      paths = search->getPaths();
      if ( paths.length() == 0 ) {
         fprintf( stderr, "No coords found, skip this mesh...\n" );
         continue;
      }
      else coords = (SoCoordinate3 *) paths[paths.length()-1]
                      ->getTail();

      // Find normals for this tri mesh
      search->setType( SoNormal::getClassTypeId() );
      search->apply( path );
      paths = search->getPaths();
      if ( paths.length() == 0 ) norms = NULL;
      else norms = ((SoNormal *) paths[paths.length()-1]->getTail())
                      ->vector.getValues(0);

      // Find colors for this tri mesh
      search->setType( SoBaseColor::getClassTypeId() );
      search->apply( path );
      paths = search->getPaths();
      if ( paths.length() == 0 ) colors = NULL;
      else colors = ((SoBaseColor *) paths[paths.length()-1]
                       ->getTail()) ->rgb.getValues(0);

     // Transform coordinates and normals by world space matrix.
      getMx->apply( path );
      mx = getMx->getMatrix();
      pts = coords->point.getValues(0);
      xpts = new SbVec3f[coords->point.getNum()];
      xnorms = new SbVec3f[coords->point.getNum()];
      for ( i = 0; i < coords->point.getNum(); i++ ) {
         mx.multVecMatrix( pts[i], xpts[i] );
         mx.multDirMatrix( norms[i], xnorms[i] );
      }

      triMesh = (SoIndexedTriangleMesh *) path->getTail();
      indices = triMesh->coordIndex.getValues(0);

      // Write tri mesh
      code = SGO_OBJ_TRIMESH;
      fwrite( &code, sizeof(long), 1, stdout );
```

```
// Count the number of controls
for ( numControls = 0, i = 0; i <
      triMesh->coordIndex.getNum(); i++ ) {
   switch( indices[i] ) {
     case SO_END_MESH_INDEX:
     case SO_SWAP_MESH_INDEX:
      if ( i == triMesh->coordIndex.getNum()-1 ) ++numControls;
      else numControls += 2;
      break;

     default:
      ++numControls;
   }
}

// Write total size of mesh object and vertices
size = coords->point.getNum() * 9 + numControls;
fwrite( &size, sizeof(long), 1, stdout );
writeVertexList( coords->point.getNum(), xpts, xnorms, colors);

// Write mesh controls
for ( i = 0; i < triMesh->coordIndex.getNum(); i++ ) {
   switch( indices[i] ) {
     case SO_END_MESH_INDEX:
      code = ( i == triMesh->coordIndex.getNum() - 1 )
                 ? SGO_OP_ENDTMESH : SGO_OP_ENDBGNTMESH;
      fwrite( &code, sizeof(long), 1, stdout );
      for( j=i+1, num=0; j<triMesh->coordIndex.getNum(); j++,
           num++)
         if ( indices[j] == SO_END_MESH_INDEX
               || indices[j] == SO_SWAP_MESH_INDEX
               || j == triMesh->coordIndex.getNum() - 1 ) {
            fwrite( &num, sizeof(long), 1, stdout );
            break;
         }
      break;

     case SO_SWAP_MESH_INDEX:
      code = SGO_OP_SWAPTMESH;
      fwrite( &code, sizeof(long), 1, stdout );
      for ( j=i+1, num=0; j<triMesh->coordIndex.getNum(); j++,
            num++)
         if ( indices[j] == SO_END_MESH_INDEX
               || indices[j] == SO_SWAP_MESH_INDEX
               || j == triMesh->coordIndex.getNum() - 1 ) {
            fwrite( &num, sizeof(long), 1, stdout );
            break;
         }
      break;

     default:
      if ( i == 0 ) {
         // First time, need to write out a BGNTMESH
         code = SGO_OP_BGNTMESH;
         fwrite( &code, sizeof(long), 1, stdout );
         for( j=i,num=0; j<triMesh->coordIndex.getNum();
              j++,num++)
            if ( indices[j] == SO_END_MESH_INDEX
                  || indices[j] == SO_SWAP_MESH_INDEX
                  || j == triMesh->coordIndex.getNum() - 1 ) {
               fwrite( &num, sizeof(long), 1, stdout );
               break;
            }
      }
      control = indices[i] * 9 * 4;
      fwrite( &control, sizeof(long), 1, stdout );
   }
}
```

```
    }
    return;
}

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

main( int argc, char *argv[] )
{
    SoInput             inFile;
    SoNode              *inData;
    long                magic = SGO_MAGIC;
    SoPathList          paths;
    long                code;
    SbBool              empty = TRUE;

    // Check command line syntax
    if ( argc != 2 ) {
        fprintf( stderr, "usage: IvToSgo iv.file > sgo.file\n" );
        exit( 1 );
    }

    // Initialize the Inventor database and read the file into memory
    SoDB::init();
    search = new SoSearchAction();
    getMx = new SoGetMatrixAction();
    if ( !inFile.openFile( argv[1] ) ) error( "Could not open input
          file" );
    if ( !SoDB::read( &inFile, inData ) )
        error( "Could not read Inventor file" );
    inData->ref();

    // Write SGO magic number
    fwrite( &magic, sizeof(long), 1, stdout );

    // Find all occurrences of Inventor FaceSet objects and translate
    // to SGO Tri or Quad List objects
    search->setFindAll( TRUE );
    search->setType( SoFaceSet::getClassTypeId() );
    search->apply( inData );
    paths = search->getPaths();
    if ( paths.length() > 0 ) {
        writeTriQuadLists( paths );
        empty = FALSE;
    }

    // Find all occurrences of Inventor TriMesh objects and translate
    // to SGO Tri Mesh objects
    search->setType( SoIndexedTriangleMesh::getClassTypeId() );
    search->apply( inData );
    paths = search->getPaths();
    if ( paths.length() > 0 ) {
        writeTriMeshes( paths );
        empty = FALSE;
    }

    // Write out end of file marker
    code = SGO_OBJ_END;
    fwrite( &code, sizeof(long), 1, stdout );

    return 0;
}
```

Section 4 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.

4.1 Tips

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

4.1.1 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 SceneViewer sample application to read your new Inventor scene graph and display the results. The SceneViewer is installed as part of the `inventor_eoe.demo` subsystem. To use it type:

```
SceneViewer filename.iv
```

4.1.2 Creating an Efficient Scene Graph

Your program can also make a second pass through the database, condensing redundant nodes into fewer nodes. In some cases, this will increase performance. 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, ordered, and/or has convex faces, specify this information in the `hints` field of `SoShapeHints`. In general, the more information you specify with `SoShapeHints`, the faster the rendering speed. The exception to this rule is that when you

specify (SURFACE | ORDERED), rendering may be slower because two-sided lighting is automatically turned on and backface culling is turned off.

4.1.3 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. For example:

```
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.setFindAll( TRUE );
    sa.apply(root);
    paths = sa.getPaths();

    for ( i = 0; i < paths.length(); 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 );
       }
    }
}
```

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.

4.1.4 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.

4.2 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

4.2.1 File Suffix

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

4.2.2 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`.

4.2.3 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`.

4.2.4 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                       IvToSgo(1)


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

SYNOPSIS
     SgoToIv sgo.file [> inventor.file]
     IvToSgo inventor.file [> sgo.file]

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

NOTES
    IvToSgo assumes the Inventor file is in a form very similar to the
SGO format.  This means that each object should contain coordinates,
normals, colors, and a Triangle Strip, Quad Mesh, or Indexed Triangle
Mesh.  All other Inventor shapes (for example, sphere) are ignored and
lost during the translation.


Page 1                       Release 1.0                  November 1992
```

4.3 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 *IRIS Inventor Nodes Quick Reference* for a list of these values.
- The vertices of polygons in vertex-based shapes should be specified in counter-clockwise order.

Section 5 File Format Syntax

This section outlines details of the syntax for the Inventor ASCII file format. Inventor's file format ignores extra white space created by spaces, tabs, and new lines (except within quoted strings). Comments begin with a number sign (#) anywhere on a line and continue to the end of the line:

```
# this is a comment in the Inventor file format
```

See the *IRIS Inventor Nodes Quick Reference* for individual descriptions of the file format for each Inventor class.

5.1 File Header

Every Inventor data file must have a standard header to identify it. This header has the following form:

```
#Inventor V1.0 ascii
```

or

```
#Inventor V1.0 binary
```

To determine whether a file is an Inventor file or not, use the `SoDB::isValidHeader()` method and pass in the first line of the file in question. Although the header may change from version to version, it is guaranteed that it will begin with a `#` sign, will be no more than 80 characters, and will end at a newline. Therefore, the C `fgets`() routine can be used. The `isValidHeader()` method returns TRUE if the file contains an Inventor header.

An example of using `fgets`() to check the header is

```
FILE *fp = fopen ( filename, "r" );
char headerString [80];
fgets( headerString, 80, fp );

if( SoDB::isValidHeader( headerString ))
   printf( "File has valid Inventor header\n");
else
   printf( "Invalid Inventor header\n");
```

5.2 Writing a Node

A node is written as:

nodename **{**
    field1name *value*
    field2name *value*
    .
    .
    .
    [*child nodes*]
    [. . .]
}

For example:

```
DrawStyle {
   style             LINES
   lineWidth         3
   linePattern       0x00ff
}
```

5.3 Writing Values within a Field

Fields within a node are written as the name of the field, followed by the value or values contained in the field. If the field value has not been changed from its default value, that field is not written out. Fields within a node can be written in any order. An example of writing field values is

```
Transform {
   translation       0 -4 0.2
}

LightModel {
   model             BASE_COLOR
}
```

Use brackets to surround multiple-value fields, with commas separating the values, as shown below for the `diffuseColor` field. It's all right to have a comma after the last value as well:

```
[ value1, value2, value3, ]
```

For example:

```
Material {
   ambientColor       .3 .1 .1
   diffuseColor      [.8 .7 .2,
                       1 .2 .2,
                      .2  1 .2,
                      .2 .2  1]
   specularColor      .4 .3 .1
   emissiveColor      .1  0 .1
}
```

Single-value fields (SF) do not contain any brackets or commas. Multiple-value fields (MF) usually have brackets, but they are not necessary if only one value is present:

```
specularColor    .4 .3 .1
```

or

```
specularColor   [.4 .3 .1 ]
```

The value that is written depends on the type of the field, as follows:

Type of Field	*Acceptable Formats*
longs, shorts, unsigned shorts	integers

floats	integer or floating point number. For example:
	`13`
	`13.0`
	`13.123`
	`1.3e-2`

names, strings	double quotation marks (`" "`) around the name if it is more than one word, or just the name (with no white space) if it is a single word. For example:
	
	`label   " front left leg "`
	`label   car`
	
	You can have any character in the string, including newlines and backslashes, except for double quotation marks. To include a double quotation mark in the string, precede it with a backslash (`\"`).

enums	either the mnemonic form of the enum or the integer equivalent. (The mnemonic form is recommended, both for portability and readability of files.) For example:
	
	`MaterialBinding {`
	`   value   PER_FACE`
	`}`

bit mask	one or more mnemonic flags, separated by a vertical bar (|) if there are multiple flags. When more than one flag is used, parentheses are required.
	
	`Cylinder {`
	`   parts   SIDES`
	`}`
	
	`Cylinder {`
	`   parts   (SIDES | TOP)`
	`}`

Type of Field	*Acceptable Formats (cont.)*

vectors (SbVec*n*f, where *n* is the number of components of the vector)	*n* floats separated by white space:
	
	`PerspectiveCamera {`
	`   position   0 0 9.5`
	`}`

colors	3 floats (RGB) separated by white space:
	
	`BaseColor {`
	`   rgb   0.3 0.2 0.6`
	`}`

rotation	a 3-vector of floats for the axis, followed by a float for the angle (in radians), separated by white space:
	
	`Transform {`
	`   rotation   0 1 0       1.5708`
	`             # y axis ... π/2 radians`
	`}`

matrix	16 floats, separated by white space (row major order)
path	An `SoSFPath` has one value, a pointer to a path. To write this value, write the path (see Section 12.3.5 of the *Inventor Programming Guide*, Volume I). An `SoMFPath` has multiple values, which are all pointers to paths. To write this value, enclose the path list in brackets, and use commas to separate each path: [first_path, second_path, ... *n*th_path]
node	An `SoSFNode` has one value, a pointer to a node. To write this value, write the node using the standard format for all nodes. An `SoMFNode` has multiple values, which are all pointers to nodes. To write this value, enclose the node list in brackets, and use commas to separate each node: [node1, node2, ... node*n*]

5.4 Ignore Flag

The ignore flag for a node (see Chapter 3 of the *Inventor Programming Guide*, Volume I) is written as a tilde (~), either after or in place of the field value or values. For example:

```
transparency [ .9, .1 ] ~
```

or

```
transparency ~
```

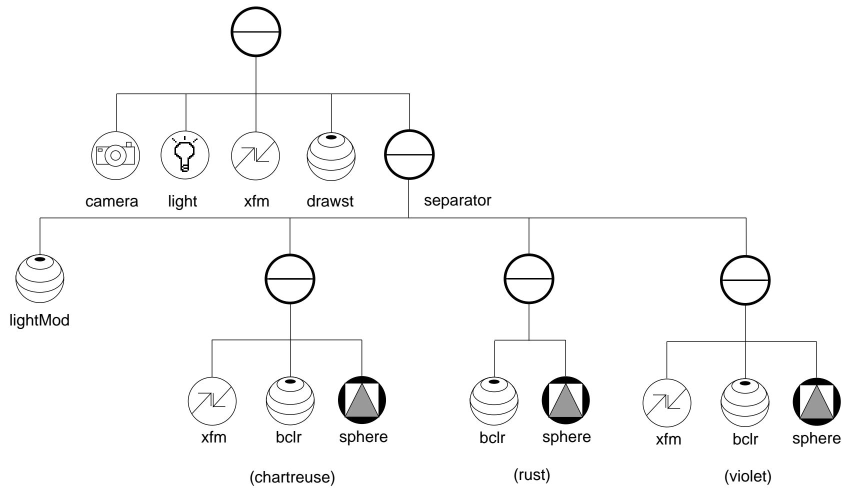
The first case preserves the values even though the field is ignored. The second case uses the default value but ignores the field.

5.5 Sample Scene Graph and Inventor File

Here is the file for a sample scene graph, which creates chartreuse, rust, and violet wireframe spheres. Figure 13 shows the scene graph for this file.

```
Separator {
   PerspectiveCamera {
      position 0 0 9.53374
      aspectRatio 1.09446
      nearDistance 0.0953375
      farDistance 19.0675
      focalDistance 9.53374
   }
   DirectionalLight { }     # note:  default fields for this light
   Transform {
      rotation -0.189479 0.981839 -0.00950093 0.102051
      center 0 0 0
   }
   DrawStyle {
      style LINES
   }
   Separator {
      LightModel {
         model   BASE_COLOR
      }
      Separator {
         Transform {
            translation -2.2 0 0
         }
         BaseColor {
            rgb .2 .6 .3   # chartreuse
         }
         Sphere { }
      }
      Separator {
         BaseColor {
            rgb .6 .3 .2   # rust
         }
         Sphere { }
      }
      Separator {
         Transform {
            translation 2.2 0 0
         }
         BaseColor {
            rgb .3 .2 .6   # violet
         }
         Sphere { }
      }
   }
}
```

Figure 13 Scene graph for a scene with three spheres



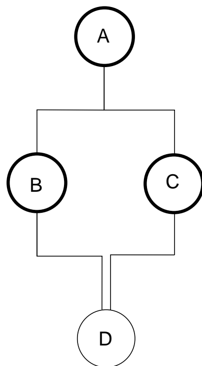
5.6 Instancing

When a scene graph contains shared instances of a node, Inventor defines a temporary name for the node and uses that name when writing subsequent occurrences of that node in the scene graph. It inserts the letters `DEF` (for define), followed by the name for the first use of the node. Thereafter, Inventor simply writes `USE` plus the name defined for that node.

For example, the scene graph shown in Figure 14 would be written as follows (without the comments):

```
Group {                  # group A
   Group {               # group B
      DEF d Cube { }     # define D and use it
   }
   Group {               # group C
      USE  d             # use D again
   }
}
```

Figure 14 Defining a node for multiple uses



Inventor uses this DEF/USE technique for paths as well as for nodes. For files to be read into Inventor, the name for a node or path can be any valid identifier. This name is thrown away after the file is read. (Identifiers start with an underscore or a letter and must consist entirely of letters, numbers, and underscores.) Names used *within* a file must be unique. A name can, however, be used in multiple files, since each file maintains its own list of names.

5.7 Including Other Files

To include a file within another file, use an `SoFile` node. This node is written as

```
File {
   name "myFile.iv"
}
```

where the `name` field is the name of the file to be included. The contents of the file `myFile.iv` are added as children of `SoFile`.

The `SoFile` node has an associated WriteBack flag. If this flag is TRUE (the default), then the children of `SoFile` are written back to their original file if anything below the `SoFile` node changed. If this flag is FALSE, then the original file remains untouched, even if changes occurred to the subgraph below the node. Use the `setWriteBack()` and `getWriteBack()` methods to change the value of this flag and to obtain its current value. If the file cannot be written, Inventor prints a warning.

5.8 ASCII and Binary Versions

The `SoOutput` object in an `SoWriteAction` has a `setBinary()` method, which sets whether the output should be in ASCII (default) or binary format. The `getOutput()` method returns a pointer to the `SoOutput`. When a file is written, Inventor inserts a header line that indicates whether the file is in ASCII or binary format, as well as the Inventor version number.

For example, to write in ASCII to a file:

```
SoWriteAction w;

w.getOutput()->setBinary( FALSE );
if ( w.getOutput()->openFile( "myFile.iv" )) {
   w.apply( root );
   w.getOutput()->closeFile();
}
```