Open Inventor™ 2.1 Porting and Performance Tips

Document Number 007-3078-001

CONTRIBUTORS

Written by Renate Kempf and Josie Wernecke
Edited by Cindy Kleinfeld
Production by Gloria Ackley
Engineering contributions by Gavin Bell, Alan Norton, Helga Thorvaldsdóttir.
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

Open Inventor™ 2.1 Porting and Performance Tips
Document Number 007-3078-001

Contents

About This Guide ix
What This Guide Contains ix
What You Should Know Before Reading This Manual x
Background Reading x
Conventions Used in This Guide x

1. Porting to Open Inventor 2.1: Getting Started 1
Porting Applications With No Custom Classes 1
Porting Custom Classes 3

2. Incompatible API Changes 5
Overview of Changes By Class Name 6
Changes to Shape Hints 8
Changes to Complexity 9
Changes to Materials and Colors 10
Changes to Normals 12
Changes to Texture Coordinates 12
Changes to Vertex-Based Shapes 13
Changes to Viewers 14
Miscellaneous Changes 16

3. Scene Appearance Changes in Inventor 2.1 19

4. New Features 21
Version-Related Changes 21
File Version 21
Inventor Revision Symbols 22
DSO Directories and Versions 22
File Reading and Writing 23
Support for VRML Files 23
User-Defined File Headers 23
SoOutput::setFloatPrecision 23
New Nodes 23
SoVertexProperty Node 24
SoLOD Node 26
SoLocateHighlight Node 26
VRML Nodes 26
New Fields 27
The “fields” Field 27
The “isA” Field 27
Miscellaneous Additions 28
SoGLRenderAction Render Abort Callback Changes 28
New Manipulator for Transformations 29
OpenGL Texture Object Extension 29
SoXtViewer Changes 29
New and Updated Utilities 30
File Downgrade Utility 30
Program for Converting Files to Use Vertex Properties 30
Program for Optimizing Scene Graphics 31
Program for Analyzing Rendering Performance 31

5. Incompatible Extender API Changes 33
Methods 33
The GLRender() Method 34
readInstance() Method 35
copy() Method 36
Elements 37
Changes in Shape Hints Element Methods 37
Changes in Binding Elements 37
Changes in Texture Elements 37
Changes in Material Elements 38
Implementation of Node Override 39
Shape Nodes 40
Generating Normals 40
Managing the Material State 40
Using the Material Bundle 44
Getting a Bounding Box 45
Material Property Nodes 45
Engine Evaluation 46

6. Optimizing Open Inventor Applications 47
Benchmarking Tips 48
Setting Performance Goals 48
Measuring Performance 49
Determining Bottlenecks 49
Modifying Your Application to Reduce Bottlenecks 50
Are You Finished Yet? 51
The Five Performance Commandments 52

Optimizing Rendering 52
Determining Whether Rendering Is the Problem 52
Isolating Rendering 53
Using the ivperf Utility to Analyze Rendering Performance 53
Correcting Window Clear Bottlenecks 55
Improving Traversal Performance 55
Organizing the Scene for Caching 56
Improving Material Change Bottlenecks 57
Optimizing Transformations 58
Performance Tip for Face Sets 58
Optimizing Textures 59
Optimizing Texture Management 59
Using Lights Efficiently 61
Optimizing Vertex Transformations 61
Optimizing Pixel Fill Operations 63
Correcting Problems ivperf Does Not Measure 63
Optimizing Everything Else 68
Useful Tools 69
Optimizing Memory Usage 69
Looking at CPU Usage 70
Optimizing Action Construction and Setup 70
Decreasing Notification Overhead 71
Picking and Handling Events 72

A. Creating a Node 73
Overview 74
Initializing the Node Class 75
Enabling Elements in the State 75
Inheritance Within the Element Stack 76
Defining the Constructor 76
Setting Up the Node's Fields 77
Defining Enumerated Values for a Field 77

Implementing Actions 78
The doAction() Method 78
Changing and Examining State Elements 80
Element Bundles 81
The SoLazyElement 82
Creating a Property Node 82
Creating a Shape Node 88
Generating Primitives 88
Rendering 90
Picking 91
Getting a Bounding Box 93
Pyramid Node 94
Creating a Group Node 108
Child List 108
Hidden Children 108
Using the Path Code 109
What Happens If an Action Is Terminated? 111
Alternate Node 111
Using New Node Classes 116
Creating an Abstract Node Class 120
The copyContents() Method 120
The affectsState() Method 121
Uncacheable Nodes 121
Creating an Alternate Representation 122
Generating Default Normals 122

Index 123

About This Guide

Open Inventor 2.1 Porting and Performance Tips explains how to port your Open Inventor™ 2.0 application to Open Inventor 2.1. Emphasis is on making the porting process go smoothly and on helping you make your application run as efficiently as possible, using the various performance enhancements provided by Open Inventor 2.1.

What This Guide Contains

This guide consists of six chapters and one appendix:

Chapter 1, "Porting to Open Inventor 2.1: Getting Started," steps you through the porting process, pointing to other relevant information as appropriate.

Chapter 2, "Incompatible API Changes," helps you find and correct for incompatible API changes by first providing an overview list, then explaining each change in more detail.

Chapter 3, "Scene Appearance Changes in Inventor 2.1," is useful if you run your Open Inventor application after it has compiled successfully and find that the scene appearance isn't the same it was in Inventor 2.0. The chapter provides a table that correlates scene appearance changes and the underlying API changes.

Chapter 4, "New Features," describes the major new features in Open Inventor 2.1.

Chapter 5, "Incompatible Extender API Changes," is for users who have customized existing Inventor 2.0 classes or created subclasses of them. It explains the new features and changes to the protected and extender methods.

Chapter 6, "Optimizing Open Inventor Applications," explains how to determine what is limiting the performance of your Open Inventor application, and provides suggestions for improving its performance.

Appendix A, "Creating a Node," provides an update to Chapter 2 of *The Inventor Toolmaker* based on changes to Inventor in 2.1.

What You Should Know Before Reading This Manual

To work successfully with this manual, you should know how to write and debug Open Inventor applications. Most readers of this book probably port applications they wrote themselves. If you haven't worked with Open Inventor before, you are urged to read the materials listed here and to gain some experience programming with Open Inventor before you attempt to port an application.

Background Reading

The following books provide background and complementary information for this manual. Books available in hard copy as well the IRIS InSight™ online viewer are marked with **(I)**:

- Wernecke, Josie, and the Open Inventor Architecture Group. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Release 2, Menlo Park: Addison-Wesley Publishing Company, 1994.**(I)**
- Open Inventor Architecture Group. *Open Inventor C++ Reference Manual.* Menlo Park: Addison-Wesley Publishing Company, 1994.
- Wernecke, Josie, and the Open Inventor Architecture Group. *The Inventor Toolmaker. Extending Open Inventor, Release 2*. Menlo Park: Addison-Wesley Publishing Company, 1994.**(I)**

Conventions Used in This Guide

This guide uses the following typographical conventions:

Italics	Filenames, IRIX command names, command-line options, function parameters, flags, and book titles.
`Fixed-width`	Code examples and system output.
Bold	C++ class names and fields. Function or method names with parentheses following the name, for example **getPackedValue()**.

Chapter 1

Porting to Open Inventor 2.1: Getting Started

This chapter gets you started porting to Open Inventor 2.1. It steps you through the porting process, pointing to other relevant information as appropriate.

The process of porting to Open Inventor 2.1 is much simpler if your application has no custom classes derived from the Inventor classes. This chapter therefore discusses the process in two separate sections:

- "Porting Applications With No Custom Classes" on page 1
- "Porting Custom Classes" on page 3

Porting Applications With No Custom Classes

To port an application without custom classes, follow these steps:

1. Compile your application under Inventor 2.1.

   If you have no errors, go to Step 4. Otherwise, continue with Step 2.

2. If you have compile or link problems, go to "Overview of Changes By Class Name" on page 6 and from there to the more detailed description.

3. Fix all your problems in sequence and recompile, or recompile after each fix. The approach you take depends on your application and your personal preference.

4. Run the application with the debug version of the library as follows:

   - Set the environment variable LD_LIBRARY_PATH to */usr/lib/libInventorDebug* to run your application with the debug version of the library.
   - Note any debug messages that occur while running your application and fix problems. (The messages should be self-explanatory.)

5. When your application compiles and links without errors, run it and watch carefully whether there are changes to the appearance of your scenes. Because some defaults changed, things may not look exactly as they did before.
6. If you find problems, go to Chapter 3, "Scene Appearance Changes in Inventor 2.1." The chapter provides a table that lets you access the API problem description based on the appearance problem description.

After the application compiles and runs satisfactorily, you are urged to also prepare it for the future by removing obsolete nodes, fields, and so on, that are supported for compatibility only.

In some cases, the old version of the API is supported in addition to the new version. This is done only to make the transition from 2.0 to 2.1 easier. To flush out cases where your code is still using the old version, compile your code with -DIV_STRICT as an option to *cc*.

Finally, optimize the application to really get the most out of Inventor 2.1 as follows:

1. Change the program and the scene graphs to take advantage of the new **vertexProperty** field of vertex-based shapes.
2. Look at the other new features discussed in Chapter 4, "New Features," and make sure you're taking advantage of the available enhancements and optimizations.
3. Look at the information in Chapter 6, "Optimizing Open Inventor Applications." It has been updated for 2.1 from the booklet of the same name (which was not previously available online). Optimize your program using that information as appropriate.

Porting Custom Classes

If your application has custom classes, porting becomes more complex. Consider doing the following:

1. Attempt to update your classes as appropriate before you compile the application for the first time. See Chapter 5, “Incompatible Extender API Changes,” for information that’s useful when updating custom classes.
2. When you believe you’ve fixed most problems, go through the steps in “Porting Applications With No Custom Classes” on page 1.
3. If compiling or linking uncovers further problems with your custom classes, see also the *include* files in */usr/include/Inventor*.

Chapter 2

Incompatible API Changes

The 2.1 release of Open Inventor provides significant performance enhancements over previous releases. In some cases an infrequently used feature was slowing down rendering, even when the feature wasn't being used. In Open Inventor 2.1, some of those features have therefore been slightly modified and some have been removed altogether.

This chapter helps you find these incompatible changes, providing the following information:

- "Overview of Changes By Class Name" on page 6
- "Changes to Shape Hints" on page 8
- "Changes to Complexity" on page 9
- "Changes to Materials and Colors" on page 10
- "Changes to Normals" on page 12
- "Changes to Texture Coordinates" on page 12
- "Changes to Complexity" on page 9
- "Changes to Viewers" on page 14
- "Miscellaneous Changes" on page 16

Note: This chapter only lists incompatible API changes. For additions to the API, see Chapter 4, "New Features."

Overview of Changes By Class Name

This section provides an overview of the incompatible API changes. If you're using the online version of this manual, you can access a change description directly by clicking on it:

- SoShapeHints—creaseAngle default value changed.
- SoShapeHints, SoDrawStyle, SoClipPlane—Wireframe and clipped objects behavior changed.
- SoShapeHints, SoCube, SoCone, SoSphere, SoCylinder—Primitive shapes behavior changed.
- SoComplexity, nodes derived from SoVertexShape—Geometry is no longer dropped for complexity values smaller than 0.5.
- SoComplexity, SoTexture2—textureQuality field behavior changed.
- SoComplexity, SoTexture2—Default texture quality is point sampling on some systems.
- SoComplexity, SoTexture2—Textures sometimes use only 12 bits of color.
- SoMaterial—Multiple materials support changed.
- SoMaterialIndex—Obsolete.
- SoColorIndex, SoLightModel—Indexed colors with PHONG light model no longer supported.
- SoPackedColor—Field for storing packed colors changed.
- SoPackedColor—getPackedValue(), setPackedValue() methods changed.
- SoMaterialBinding, SoMaterial—Correct number of diffuse colors required.
- SoMaterialBinding, SoMaterial—Correct number of transparencies required.
- SoMaterialBinding—DEFAULT and NONE bindings obsolete.
- SoNormal—Default normal no longer provided.
- SoNormalBinding, SoNormal—Correct number of normals required.
- SoNormalBinding, SoNormal—Normals generated automatically.

- SoNormalBinding—DEFAULT and NONE bindings obsolete.
- SoTextureCoordinate2—Default texture coordinate no longer provided.
- SoTextureCoordinate2—Automatic generation of texture coordinates as needed.
- SoTextureCoordinate2—Correct number of texture coordinates required.
- SoTextureCoordinateBinding—DEFAULT binding obsolete.
- Shapes derived from SoNonIndexedShape—startIndex field obsolete.
- Shapes derived from SoNonIndexedShape—USE_REST_OF_VERTICES not fully supported.
- SoXtViewer—Texture mapping interactive draw style change.
- SoXtPlaneViewer, SoXtExamineViewer, SoXtWalkViewer—Key bindings changed.
- Shapes derived from SoNonIndexedShape—USE_REST_OF_VERTICES not fully supported.
- All nodes—Limited support for Override flag.
- SoOutput—isASCIIHeader() and isBinaryHeader() no longer supported.
- SoCallback, SoEventCallback—Reset required after making direct OpenGL calls.
- SoLineHighlightRenderAction, SoBoxHighlightRenderAction, SoGLRenderAction—Constructor arguments changed.
- SoSFLong, SoMFLong, SoSFULong, SoMFULong—long changed to int32.
- All draggers and manipulators—Control key change.

Changes to Shape Hints

- **SoShapeHints**—**creaseAngle** default value changed.

  The **creaseAngle** field is used when default normals are generated. The default value has been changed from 0.5 radians to 0.0 radians. As a result, default normals are generated faster and use less memory. However, objects that were smoothly shaded by default by previous versions of Inventor will have faceted normals in Inventor 2.1.

- **SoShapeHints**, **SoDrawStyle**, **SoClipPlane**—Wireframe and clipped objects behavior changed.

  Drawing in wireframe mode (using the **SoDrawStyle** node) or clipped mode (using **SoClipPlane** nodes) no longer turns off SOLID **SoShapeHints**; clipped or wireframe objects have backfaces removed if the appropriate **SoShapeHints** value is specified.

- **SoShapeHints**, **SoCube**, **SoCone**, **SoSphere**, **SoCylinder**—Primitive shapes behavior changed.

  The primitive shapes (**SoCube**, **SoCone**, **SoSphere**, **SoCylinder**) no longer automatically turn on shapeType = SOLID and vertexOrdering = COUNTERCLOCKWISE on **SoShapeHints** nodes. You must insert an **SoShapeHints** node at the top of your scenes to get the old primitive behavior. Note that if you don't insert an **SoShapeHints** node, backface culling is disabled. This may slow down the frame rate.

  On the other hand, you can now specify an **SoShapeHints** node of shapeType = UNKNOWN_SHAPE_TYPE and vertexOrdering = COUNTERCLOCKWISE if you want both the interior and exterior surfaces of primitive shapes to be lit. If you only want the inside of the object lit, for example if the viewer is always inside a sphere, you can specify an **SoShapeHints** node with shapeType = SOLID and vertexOrdering = CLOCKWISE.

Changes to Complexity

- **SoComplexity,** nodes derived from **SoVertexShape**—Geometry is no longer dropped for complexity values smaller than 0.5.

  When you applied complexity values of less than 0.5 to triangle strips and other vertex-based shapes in Inventor 2.0, some of the geometry was dropped (for example, alternate strips were deleted from a triangle strip set). In Inventor 2.1 this no longer happens. No geometry is deleted from such shapes because this was not an effective way of speeding up the rendering of complex shapes.

- **SoComplexity, SoTexture2—textureQuality** field behavior changed.

  The **textureQuality** field of the **SoComplexity** node now takes effect only when the next **SoTexture2** node is traversed, not immediately as in Inventor 2.0. As a result, mipmap creation for an **SoTexture2** node can be avoided if the texture quality is low enough. If you arranged your scene graphs so that the **textureQuality** field of **SoComplexity** followed the **SoTexture2** nodes, re-arrange them so that the **SoComplexity** node is traversed first.

- **SoComplexity**, **SoTexture2**—Default texture quality is point sampling on some systems.

  On systems that do not provide hardware support for texture mapping, the default **textureQuality** field is interpreted to mean point sampling of the texture, and mipmaps are not used.

- **SoComplexity**, **SoTexture2**—Textures sometimes use only 12 bits of color.

  If **textureQuality** is less than 0.8, only 12 bits of texture color is used (instead of 24) on systems that support the OpenGL texture extension (EXT_texture). This increases the fill rate but decreases visual quality.

Changes to Materials and Colors

- **SoMaterial**—Multiple materials support changed.

  Open Inventor 2.1 no longer supports multiple values for the following properties specified in an **SoMaterial** node:

  – **ambientColor**

  – **specularColor**

  – **emissiveColor**

  – **shininess**

  Open Inventor 2.1 continues to support multiple values for:

  – **diffuseColor**

  – **transparency**

  If you provide multiple transparencies, you must provide as many transparencies as diffuse colors. If you provide just one transparency, it applies to all materials in the shape.

  For backward compatibility, the field types for **ambientColor**, **specularColor**, **emissiveColor**, and **shininess**, have not changed. They are still **SoMF-** fields.

  Note that the new **SoVertexProperty** node provides explicit support (and accelerated performance) for changing diffuse color and transparency within a shape.

- **SoMaterialIndex**—Obsolete.

  The **SoMaterialIndex** node is no longer supported. For indexed color rendering, use the **SoColorIndex** node instead.

- **SoColorIndex**, **SoLightModel**—Indexed colors with PHONG light model no longer supported.

  The PHONG light model is no longer supported with indexed colors. Indexed colors can still be used with BASE_COLOR lighting.

- **SoPackedColor**—Field for storing packed colors changed.

  The field for storing packed colors in the **SoPackedColor** node has been changed to be more compatible with OpenGL. The field name is now **orderedRGBA**, and the packing order is 0xrrggbbaa, where aa represents the alpha value, and rr, gg, and bb represent the red, green, and blue components of the color, respectively. In previous releases, the field name was rgba and the packing order was 0xaabbggrr.

- **SoPackedColor—getPackedValue()**, **setPackedValue()** methods changed.

  The **getPackedValue()** and **setPackedValue()** methods on the **SbColor** class have changed.

  – **getPackedValue()** now takes a transparency value to pack with the color.

  – **setPackedValue()** returns a transparency value along with the **SbColor**. The transparency value, like the transparency field in **SoMaterial**, ranges from 0.0 to 1.0, where 0.0 is fully opaque, and 1.0 is fully transparent.

- **SoMaterialBinding**, **SoMaterial**—Correct number of diffuse colors required.

  Open Inventor 2.1 assumes the correct number of diffuse colors is specified. When the material binding of a vertex-based shape is not OVERALL, provide one diffuse color for each vertex, face, or part, depending on whether PER_VERTEX, PER_FACE, or PER_PART binding is used. If you don't provide enough diffuse colors, you may see unexpected results. Note that in previous releases, Inventor cyclically reused the colors provided.

- **SoMaterialBinding**, **SoMaterial**—Correct number of transparencies required.

  If not enough transparencies are provided, the first one is used.

- **SoMaterialBinding**—DEFAULT and NONE bindings obsolete.

  The DEFAULT and NONE material bindings are obsolete. Files containing these bindings are transparently upgraded as if they specified OVERALL materials. Applications that use DEFAULT or NONE continue to work, unless they are compiled with -DIV_STRICT.

Changes to Normals

- **SoNormal**—Default normal no longer provided.

  In previous releases, an **SoNormal** node contained one normal by default. In Open Inventor 2.1 the default is not to provide any normals.

- **SoNormalBinding**, **SoNormal**—Correct number of normals required.

  If any normals are provided in a scene, Open Inventor 2.1 assumes the correct number is specified. There should be one normal for each shape, part, face, or vertex, depending on whether the normal binding is specified as OVERALL, PER_PART, PER_FACE, or PER_VERTEX.

- **SoNormalBinding**, **SoNormal**—Normals generated automatically.

  In previous releases, normals were automatically generated only if the normal binding was DEFAULT. In Open Inventor 2.1, normals are always automatically generated if they are needed and if the user does not provide any normals in the scene. The correct number of normals is generated, depending on the normal binding.

- **SoNormalBinding**—DEFAULT and NONE bindings obsolete.

  The DEFAULT and NONE normal bindings are obsolete. Files containing these bindings are transparently upgraded as if they specified PER_VERTEX_INDEXED normals. Applications that use the DEFAULT or NONE enums continue to work, unless they are compiled with -DIV_STRICT.

Changes to Texture Coordinates

- **SoTextureCoordinate2**—Default texture coordinate no longer provided.

  In previous releases, an **SoTextureCoordinate2** node contained one texture coordinate by default. In Open Inventor 2.1 the default is not to provide any texture coordinates at all.

- **SoTextureCoordinate2**—Automatic generation of texture coordinates as needed.

  In Open Inventor 2.1, texture coordinates are always automatically generated if they are needed and no texture coordinates are provided in the scene.

- **SoTextureCoordinate2**—Correct number of texture coordinates required.

  If any texture coordinates are provided in a scene, Open Inventor 2.1 assumes there is one for each vertex. You will see unexpected results if there are fewer texture coordinates than Inventor expected.

- **SoTextureCoordinateBinding**—DEFAULT binding obsolete.

  The DEFAULT for the **SoTextureCoordinateBinding** node is obsolete. You can still specify PER_VERTEX or PER_VERTEX_INDEXED.

Changes to Vertex-Based Shapes

- Shapes derived from **SoNonIndexedShape**—**startIndex** field obsolete.

  The **startIndex** field of a vertex-based shape is replaced by the new **SoVertexProperty** node.

  Previously you could specify the vertex properties for multiple shape nodes with one set of property nodes by having a different startIndex for each shape. However, a nonzero **startIndex** field may not work as expected when used with coordinates in an **SoVertexProperty** node. Furthermore, confusion arises if some of the vertex properties come from the **vertexProperty** field of the shape and others are inherited from nodes in the scene.

  To facilitate transition from Inventor 2.0, Inventor 2.1 continues to support use of **startIndex** when all vertex properties are specified in the state. You are urged to remove dependencies on this capability.

- Shapes derived from **SoNonIndexedShape**—USE_REST_OF_VERTICES not fully supported.

  In previous releases, when a value of -1 was found in the **numVertices** field of non-indexed shapes—for example, **SoTriangleStripSet**—all the remaining values in the current coordinates were used. In Inventor 2.1, this is not supported when the coordinates are specified through the new **vertexProperty** field of the shape. Inventor 2.1 continues to support the use of -1 for **numVertices** when the coordinates are specified in the inherited state. You are urged to remove dependencies on the old behavior.

Changes to Viewers

The following changes to viewers were made in Inventor 2.1:

- **SoXtViewer**—Texture mapping interactive draw style change.

  On systems that do not have hardware support for texture mapping, the viewers now by default set the interactive draw style to move without textures. In other words, by default, the viewers show the texture on these systems when the camera is not moving, but when you interact with the camera, the texture is disabled. Use the **setDrawStyle()** method of the **SoXtViewer** class to change the viewer draw style. End users of the viewer can change the draw style using the right-mouse popup menu.

- **SoXtPlaneViewer**, **SoXtExamineViewer**, **SoXtWalkViewer**—Key bindings changed.

  The key bindings for the three viewers have changed to make them more consistent with each other and to match WebSpace Navigator Viewers.

The new bindings for **SoXtPlaneViewer** are the following:

Action	Result
Left mouse	Zoom in and out.
Ctrl + left mouse Middle mouse	Translate up, down, left, right.
Ctrl + middle mouse	Rotate around the viewer in forward direction (roll action).
<s> click	Alternative to Seek button. Press (but don't hold) the <s> key, then click on a target object.
Left mouse	Zoom in and out.
Ctrl + left mouse, Middle mouse	Translate up, down, left, right.
Right mouse	Open pop-up menu.

The key bindings for **SoXtExaminerViewer** have changed to be the following:

Action	Result
Left mouse	Tumbles the virtual trackball.
Middle mouse Ctrl + left mouse	Translate up, down, left, and right
Ctrl + middle mouse Left and middle mouse	Zoom in and out.
<s> click	Alternative to Seek button. Press (but don't hold) the <s> key, then click on a target object.
Right mouse	Open pop-up menu.

The key bindings for **SoXtWalkViewer** have changed to be the following:

Action	Result
Left mouse	Walk mode. Hold the mouse button and move up/down for forward/backward motion. Move right and left for turning. Speed increases exponentially with the distance from the mouse-down origin.
Ctrl + left mouse Middle mouse	Translate up, down, left, right.
Ctrl + middle mouse	Tilt the camera up/down and right/left. This lets you look around while stopped.
<s> click	Alternative to Seek button. Press (but don't hold) the <s> key, then click on a target object.
<u> click	Press (but don't hold) the <u> key, then click on a target object to set the up direction to the surface normal. By default, positive y is the up direction.
Right mouse	Open pop-up menu.

Miscellaneous Changes

- All nodes—Limited support for *Override* flag.

  In Open Inventor 2.1, not all of the nodes support the *Override* flag. The following nodes do support override:

  – **SoColorIndex**
  – **SoComplexity**
  – **SoDrawStyle**
  – **SoFont**
  – **SoLightModel**
  – **SoMaterial**
  – **SoMaterialBinding**
  – **SoPackedColor**
  – **SoPickStyle**
  – **SoShapeHints**
  – **SoTexture2**

  Other node classes ignore the *Override* flag. Note that you can still set the flag for those classes, but it has no effect.

  An *Override* flag on an **SoMaterial** node also overrides the **diffuseColor** and **transparency** fields of **SoVertexProperty** nodes (in the scene graph or in the **vertexProperty** field of shapes). Likewise an *Override* flag in a **SoMaterialBinding** node overrides the **materialBinding** field in a vertex property node. However, you cannot override just the diffuse color or just the transparency. If you override one, the other is obtained from the state when the override occurs.

- **SoOutput—isASCIIHeader()** and **isBinaryHeader()** no longer supported.

  The **SoOutput** methods **isASCIIHeader()** and **isBinaryHeader()** are no longer supported. Use **SoDB::isValidHeader()** to verify that a header is valid, or **SoDB::getHeaderData()** to see if a particular header is for ASCII or binary files.

- **SoCallback**, **SoEventCallback**—Reset required after making direct OpenGL calls.

  You must notify the Inventor material state management mechanism by calling **SoGLLazyElement::reset()** if you directly make any of the following OpenGL calls from a callback node:

  – **glColor()**

  – **glMaterial()**

  – **glColorMaterial()**

  – **glEnable()** or **glDisable()**, with a GL_COLOR_MATERIAL, GL_BLEND, GL_LIGHTING, or GL_POLYGON_STIPPLE argument.

  – **glPushAttrib()** or **glPopAttrib()** with GL_ENABLE_BIT, GL_LIGHTING_BIT, GL_POLYGON_BIT, GL_POLYGON_STIPPLE_BIT

  After the last OpenGL call in your callback node, you must call **SoGLLazyElement::reset(state, bitmask).** The value of the bitmask depends on which of the above OpenGL calls you made.

  If you invoke **glColor()** or **glMaterial()**, logical-OR one or more of the following bitmasks as argument to **SoGLLazyElement::reset()** (depending on which components of the OpenGL material state you are altering):

  – **SoLazyElement::DIFFUSE_MASK**

  – **SoLazyElement::EMISSIVE_MASK**

  – **SoLazyElement::SPECULAR_MASK**

  – **SoLazyElement::SHININESS_MASK**

  – **SoLazyElement::AMBIENT_MASK**

  If you invoke **glColorMaterial()**, or enable or disable GL_COLOR_MATERIAL, the corresponding mask is **SoLazyElement::COLOR_MATERIAL_MASK**.

  If you enable or disable GL_BLEND, use **SoLazyElement::BLENDING_MASK**.

If you alter the GL stipple transparency, either with **glEnable()**, **glDisable()**, **glPushAttrib()**, or **glPopAttrib()**, use **SoLazyElement::TRANSPARENCY_MASK**.

If you alter lighting, either with **glEnable()**, **glDisable()**, **glPushAttrib()**, or **glPopAttrib()**, use **SoLazyElement::LIGHT_MODEL_MASK**

- **SoLineHighlightRenderAction, SoBoxHighlightRenderAction, SoGLRenderAction**—Constructor arguments changed.

  The constructor argument no longer takes *useCurrentGLValues* as an argument.

- **SoSFLong**, **SoMFLong**, **SoSFULong**, **SoMFULong**—long changed to int32.

  The **So*Long** fields have been changed to **So*Int32** fields as follows:

  – SoSFLong -> SoSFInt32

  – SoSFULong -> SoSFUInt32

  – SoMFLong -> SoMFInt32

  – SoMFULong -> SoMFUInt32

  You need to change your program to use the new fields, and to use the types int32_t and uint32_t, defined in *inttypes.h* in place of long and unsigned long. Use the *longToInt32* utility program provided in the *inventor_dev.src.sample* subsystem to help you do this.

  To ease the transition, a typedef links the **So*Long** fields to **SoInt32** fields and, where possible and feasible, methods taking arguments of type long are supported.

  Since the C++ compiler cannot distinguish overloaded functions by return type alone, some methods are no longer supported. If you compile your program with -DIV_STRICT, the **So*Long** fields cause a compile time error.

- All draggers and manipulators—Control key change.

  All draggers and manipulators now use the <Ctrl><Meta> key where they used the <Alt> key in Inventor 2.0.

Chapter 3

Scene Appearance Changes in Inventor 2.1

Because certain fields in some nodes changed their defaults, and because of other changes described in Chapter 2, your Inventor 2.0 application may look different in Inventor 2.1 even if it compiles and runs fine.

Table 3-1 lists changes in appearance and points to information that helps you fix the problem. If you're working with the online version of the manual, click on any "Reason" for more detailed information.

Table 3-1 Scene Appearance Changes in Inventor 2.1

Scene Appearance Change	Reason
Texture quality looks different.	SoComplexity, SoTexture2—textureQuality field behavior changed. SoComplexity, SoTexture2—Textures sometimes use only 12 bits of color. SoComplexity, SoTexture2—Default texture quality is point sampling on some systems.
When you view a texture-mapped object, the texture seems to disappear.	SoXtViewer—Texture mapping interactive draw style change.
Shapes that were smooth shaded look faceted.	SoShapeHints—creaseAngle default value changed.
When you set the draw style to wireframe, some of the lines are missing.	SoShapeHints, SoDrawStyle, SoClipPlane—Wireframe and clipped objects behavior changed.
A scene that has many spheres (and/or cones, cubes, cylinders) seems to render slower than before.	SoShapeHints, SoCube, SoCone, SoSphere, SoCylinder—Primitive shapes behavior changed.

Table 3-1 Scene Appearance Changes in Inventor 2.1

Scene Appearance Change	Reason
Vertex-based shapes look the same, regardless of the SoComplexity value.	SoComplexity, nodes derived from SoVertexShape—Geometry is no longer dropped for complexity values smaller than 0.5.
The material looks different than it did in Inventor 2.0 for objects that had a different material for every part or every vertex specified.	SoMaterial—Multiple materials support changed.
Colors in a shape look wrong.	SoMaterial—Multiple materials support changed. SoMaterialBinding, SoMaterial—Correct number of diffuse colors required.
A vertex-based shape used several colors repeatedly for the different parts and this no longer looks right.	SoMaterialBinding, SoMaterial—Correct number of diffuse colors required.
In a program that contains direct calls to OpenGL, the materials look wrong.	SoCallback, SoEventCallback—Reset required after making direct OpenGL calls.

Chapter 4

New Features

This chapter describes the major new features that distinguish Open Inventor 2.1 from Open Inventor 2.0. It discusses the following topics:

- "Version-Related Changes" on page 21
- "File Reading and Writing" on page 23
- "New Nodes" on page 23
- "New Fields" on page 27
- "Miscellaneous Additions" on page 28
- "New and Updated Utilities" on page 30

Version-Related Changes

This section discusses various issues directly related to the version change:

- "File Version"
- "Inventor Revision Symbols"
- "DSO Directories and Versions"

File Version

The file version has been updated to 2.1; files written out with Open Inventor 2.1 have the file header `#Inventor V2.1 ascii` or `#Inventor V2.1 binary`. Inventor 2.1 programs are still able to read 2.0 and 1.0 files. Older programs cannot read 2.1 files; the files must be converted to the appropriate version with the *ivdowngrade* program discussed in "File Downgrade Utility" on page 30.

Inventor Revision Symbols

C preprocessor symbols SO_VERSION and SO_VERSION_REVISION have been added to *Inventor/SbBasic.h* to identify the revision of Inventor being compiled against:

- SO_VERSION is the major version number (for Inventor 2.1, “2”)
- SO_VERSION_REVISION is the minor revision number (for Inventor 2.1, “1”)

If you use parts of the Inventor API that changed between revisions, you can use these symbols to conditionally compile (#ifdef) your code if you want the same source to compile against multiple versions of Inventor.

DSO Directories and Versions

To avoid problems with incompatible code, the places Inventor searches for DSOs (dynamic shared objects) implementing new nodes has been changed. Inventor 2.1 searches in the following order:

```
LD_LIBRARY_PATH:/usr/lib:/lib              (normal rld rules)
.:/usr/local/lib/InventorDSO:/usr/lib/InventorDSO       (new)
```

If you’re running a *setuid* or *setgid* program, or if running as root, the order is:

```
/usr/lib:/lib                              (normal rld rules)
/usr/lib/InventorDSO                                    (new)
```

The */usr/lib/Inventor* and */usr/local/lib/Inventor* directories were used for Inventor 2.0. For example, if you created a new node of type *NewNode* and created NewNode.so DSOs for both Inventor 2.0 and Inventor 2.1, you could put the Inventor 2.0 NewNode.so in the */usr/local/lib/Inventor/* directory and put the Inventor 2.1 NewNode.so in the */usr/local/lib/InventorDSO/* directory.

To avoid future problems with incompatible DSOs, Inventor 2.1 and future releases of Inventor will search for DSOs that are given the same version number as the Inventor libraries. For Inventor 2.1, DSOs for new nodes and engines must be tagged with version “sgi3.0”. This is accomplished by adding the flag *-set_version “sgi3.0”* when creating the DSO. See the reference page for *ld* for more information on DSO versioning, and the Inventor release notes for information on creating DSOs for new Inventor classes.

File Reading and Writing

This section explains changes related to file reading and writing.

Support for VRML Files

The **SoDB** read methods now also read files with the VRML 1.0 header.

User-Defined File Headers

The **SoDB** class now includes methods for registering your own file headers. This is useful, for example, if you have an application that supports a superset of the Inventor file format, and you want to use the Inventor file reading mechanism to parse the file. Similarly, you can specify your own header for output files using methods of the **SoOutput** class. See the reference pages for more details.

SoOutput::setFloatPrecision

You can now specify the precision Inventor uses when writing floating point numbers in the ASCII file format. The new **setFloatPrecision()** method on **SoOutput** takes one argument, an integer specifying the desired precision. For example, a precision of 2 limits the output of floating point numbers to 2 significant digits.

New Nodes

This section discusses new nodes that don't result in incompatible API changes or scene appearance changes. It provides information about:

- "SoVertexProperty Node"
- "SoLOD Node"
- "SoLocateHighlight Node"
- "VRML Nodes"

SoVertexProperty Node

A new node class, **SoVertexProperty**, has been added. The **SoVertexProperty** node is a performance feature of Inventor 2.1; it does not add functionality. Open Inventor applications that use this node may show a speed improvement, particularly for non-cached rendering.

Description

The **SoVertexProperty** node allows you to specify in one node all the vertex data for a vertex-based shape (that is, any shape derived from **SoVertexShape**). This is more efficient than inheriting the data from several different nodes. An **SoVertexProperty** node has the following fields:

vertex the coordinates, specified as **SoMFVec3f**

normal the normal vectors, specified as **SoMFVec3f**

normalBinding the normal binding, specified as **SoSFEnum**

orderedRGBA the packed color and alpha values, specified as **SoMFUInt32**

materialBinding the material binding, specified as **SoSFEnum**

texCoord the texture coordinates, specified as **SoMFVec2f**

When the **SoVertexProperty** node is used, it replaces the **SoCoordinate3**, **SoNormal**, **SoNormalBinding**, **SoTextureCoordinate2**, **SoMaterial** (when used to specify multiple colors), and **SoMaterialBinding** nodes.

Defaults

The **vertex**, **normal**, **orderedRGBA**, and **texCoord** fields are all NULL by default. If any of these fields are not specified, the shape inherits the values from other nodes in the scene.

The default **normalBinding** is PER_VERTEX, and the default **materialBinding** is OVERALL. However, the **normalBinding** is ignored if the **normal** field is not used. Similarly, the **materialBinding** field is ignored if the **orderedRGBA** field is not used.

Usage

The **SoVertexProperty** node can be used by the shape in either of two ways:

- The **SoVertexProperty** node is placed in the scene and inherited by the shape node.

  This is the same model used for inheritance for other property nodes in Inventor.

- The vertex-based shapes have a new **SoSFNode** field, **vertexProperty**, that allows you to directly include the **SoVertexProperty** node in the shape.

  This is a significant change from the inheritance model used for other properties. The properties specified in the node apply only to the shape in which it was included; they do not affect any subsequent shapes.

Performance Considerations

For maximum performance, observe the following rules of thumb:

- Use the **SoVertexProperty** node to specify properties for the vertex-based shapes if you have a scene that cannot be cached.
- Use the **vertexProperty** field of the shapes to directly include the **SoVertexProperty** node; don't place the **SoVertexProperty** node in the scene and inherit by the shape node.
- Use the fields in the **SoVertexProperty** node to specify all the data you need for a shape; don't mix and match data from the **SoVertexProperty** node and inherited values from other nodes.
- Specify normals and texture coordinates if you need them; don't rely on the automatic generation, which is expensive if the scene cannot be cached.
- Don't use the convenience feature of specifying the number of vertices as -1 (that is, use all remaining vertices); specifying the actual number is more efficient.

SoLOD Node

Open Inventor 2.1 has an improved version of the node that provides switching between different levels of detail in the scene. The new node is named **SoLOD**, to avoid conflict with the old node, **SoLevelOfDetail**. For compatibility, the old node is still supported.

The new **SoLOD** node switches between different levels, depending on the distance of the object from the eye. The old **SoLevelOfDetail** node used screen area to determine when to switch levels. Using distance is much faster than using screen area.

SoLocateHighlight Node

SoLocateHighlight is a new, special separator that performs locate highlighting. When the window cursor is over the contents of this separator, it efficiently redraws itself in a different color. This is useful for indicating hot spots in a scene. See the reference page for more details.

VRML Nodes

The Open Inventor file format formed the basis for the Virtual Reality Modeling Language (VRML), the industry-standard, platform-independent file format for 3D graphics on the Internet. VRML 1.0 includes a few nodes that were not in Inventor 2.0; these nodes have been added to Open Inventor 2.1, which is now a superset of VRML 1.0. The new nodes in Open Inventor 2.1 are:

- **SoWWWAnchor**
- **SoWWWInline**
- **SoAsciiText**
- **SoFontStyle**

See the reference pages for more information on these nodes. For more information on VRML, visit the World Wide Web page at *http://vrml.wired.com*.

New Fields

This section looks at two new fields, “The “fields” Field” and “The “isA” Field.”

The “fields” Field

Inventor writes custom nodes with an extra field named **fields** that lists the names of all fields in the node along with their types. Inventor 2.0 files could not be read if a **fields** field was included in a node that was a standard part of the library. To be fully compliant with VRML 1.0, Inventor 2.1 files now support a **fields** field in all nodes.

For example, the following code fragment is legal in an Inventor 2.1 file:

```
Cube {
    fields [SFFloat width,
            SFFloat height,
            SFFloat depth,]
    width 10
    height 4
    depth 3
}
```

The “isA” Field

When a new node type is a superset of an existing node, and an implementation for the new node type cannot be found, the new node type can be treated as the existing node it is based on (with some loss of functionality).

To support this, new node types can define an **MFString** field called **isA** containing the names of the types of which it is a superset.

For example, assume a new subset of **Material** called **ExtendedMaterial** adds the index of refraction as a material property. This type can be defined as follows:

```
ExtendedMaterial {
  fields [ MFString isA, MFFloat indexOfRefraction,
           MFColor ambientColor, MFColor diffuseColor,
           MFColor specularColor, MFColor emissiveColor,
           MFFloat shininess, MFFloat transparency ]
  isA [ "Material" ]
  indexOfRefraction .34
  diffuseColor .8 .54 1
}
```

If the **ExtendedMaterial** node is not known, an **alternateRep** field containing an **SoMaterial** node is automatically created.

Miscellaneous Additions

This section briefly discusses:

- "SoGLRenderAction Render Abort Callback Changes"
- "New Manipulator for Transformations"
- "OpenGL Texture Object Extension"
- "SoXtViewer Changes"

SoGLRenderAction Render Abort Callback Changes

The render abort callback in Open Inventor 2.1 no longer returns a simple Boolean value. Instead, it returns an abort code (similar to the **SoCallbackAction::Response** code) that is one of CONTINUE, ABORT, PRUNE, or DELAY.

- CONTINUE is the same as returning FALSE in previous versions of Inventor; it means that rendering should continue as usual.
- ABORT is the same as returning TRUE in previous versions; it terminates the current render traversal.
- PRUNE indicates that traversal should skip the current node and all nodes under it.

- DELAY postpones traversal of the current node until after all other nodes have been traversed, just like the **SoAnnotation** node.

PRUNE and DELAY are new features that allow applications to modify Inventor's standard rendering order.

New Manipulator for Transformations

A new manipulator, **SoTransformerManip,** has been added to the set of Inventor manipulators. It provides a full interface for rotation, translation, and scale in three dimensions. This new manipulator provides better feedback than the older manipulators, and uses the new locate highlighting feature to indicate which of its parts to pick. See the reference page for more details.

SoTransformerManip is used in IRIS Annotator™ and WebSpace Author™.

OpenGL Texture Object Extension

Open Inventor 2.1 makes use of the OpenGL extension for texture objects. Consequently, you see improved texture mapping performance on systems where this OpenGL extension is available.

SoXtViewer Changes

There are two changes to **SoXtViewer**:

- When a viewer is in pick mode, you can temporarily switch to view mode by holding down the <ALT> key.
- The method **SoXtViewer::setCursorEnabled()** allows you to specify whether the viewer is allowed to change the cursor over the render area window. Disabling the cursor enables the application to set the cursor directly. This is useful, for example, if you want to set a busy cursor in the window.

New and Updated Utilities

The following new and updated utilities are available in Inventor 2.1 and are discussed in this section:

- "File Downgrade Utility"
- "Program for Converting Files to Use Vertex Properties"
- "Program for Optimizing Scene Graphics"
- "Program for Analyzing Rendering Performance"

Note: The utility program *ivquicken* is no longer supported.

File Downgrade Utility

A new utility program, */usr/sbin/ivdowngrade*, takes an Inventor file and converts it to Inventor version 2.0 or 1.0. See the reference page for more details.

Program for Converting Files to Use Vertex Properties

The *ivAddVP* utility program helps convert Inventor files into 2.1 files that use the **vertexProperty** field for the vertex-based shapes. The source code, including a Makefile, is distributed in the *inventor_dev.src.sample* subsystem. It is installed in the directory */usr/share/src/Inventor/tools/ivAddVP*.

Program for Optimizing Scene Graphics

A new utility program, */usr/sbin/ivfix*, restructures Inventor scene graphs for improved rendering performance. *ivfix* first analyzes the organization of the input scene graph and tries to sort it to take advantage of coherence. For example, it tries to organize subgraphs by common textures, since switching textures is expensive. Once sorting is complete, *ivfix* also tries to combine subgraphs so that the final result has fewer nodes. Finally, *ivfix* "flattens" the subgraphs, tessellating all shapes into triangles that are then organized into triangle strips. For example, two spheres may be combined into one triangle strip.

The source code to *ivfix*, including a Makefile, is distributed in the *inventor_dev.src.sample* subsystem. It is installed in the directory */usr/share/src/Inventor/tools/ivfix*.

Program for Analyzing Rendering Performance

The utility program *ivperf*, for analyzing rendering performance, is provided in source code format in the *inventor_dev.src.sample* subsystem. It is installed in the directory */usr/share/src/Inventor/tools/ivperf*. For information on *ivperf*, see Chapter 6, "Optimizing Open Inventor Applications."

Chapter 5

Incompatible Extender API Changes

This chapter is for developers who have customized existing Inventor 2.0 classes or have created subclasses of them. If you have problems porting an application with custom classes or subclasses, this chapter helps you solve them by explaining the new features and changes to the protected and extender methods.

Note: To make your transition to Inventor 2.1 easier, the old version of the API is still supported in some cases. To flush out cases where your code is still using the old version, compile your program with -DIV_STRICT.

This chapter discusses the following topics:

- "Methods" on page 33
- "Elements" on page 37
- "Shape Nodes" on page 40
- "Material Property Nodes" on page 45
- "Engine Evaluation" on page 46

Methods

This section provides information about changed methods, discussing the following topics:

- "The GLRender() Method"
- "readInstance() Method"
- "copy() Method"

The GLRender() Method

There are two potential changes when working with a **GLRender()** method:

- "Overriding Existing GLRender() Methods"
- "Implementing SoSeparator GLRender() Methods Efficiently"

For additional information, see "How to Convert GLRender() Methods" on page 42 in "Managing the Material State."

Overriding Existing GLRender() Methods

Overriding an existing node's **GLRender()** method by registering a method using **SoGLRenderAction::addMethod()** no longer works. **SoGLRenderAction** no longer uses the static action/method table for its traversal, but instead calls **GLRender()** methods directly (see "Implementing SoSeparator GLRender() Methods Efficiently").

Instead of calling **SoGLRenderAction::addMethod()**, implement a subclass of the node(s) with the **GLRender()** methods you wish to override and use the **SoType::overrideType()** method in its **initClass()** routine instead of the normal SO_NODE_INIT_CLASS() macro. See the comments in *SoType.h* for more information on **SoType::overrideType()**.

Implementing SoSeparator GLRender() Methods Efficiently

To make rendering traversal faster, **SoNode** now defines four different rendering methods: **GLRender()**, **GLRenderBelowPath()**, **GLRenderInPath()**, and **GLRenderOffPath()**. By default, the three new methods just call **GLRender()**. However, for optimal rendering speed, group nodes can redefine **GLRender()** to call one of the other methods based on the current path code (the **SoAction::getPathCode()** method returns the current path code).

For example, **SoSeparator** defines a **GLRender()** method that calls **SoSeparator::GLRenderBelowPath()** or **SoSeparator::GLRenderInPath()**, based on the path code. Once **GLRenderBelowPath()** is called, traversal is faster, since the path code is guaranteed not to change underneath the separator, and the separator can just call its children's **GLRenderBelowPath()** methods.

If you have a class derived from **SoSeparator** and you implemented a **GLRender()** method for that class, you need to implement **GLRenderBelowPath()** and **GLRenderInPath()** methods; otherwise, the methods of the **SoSeparator** are called and your special rendering code is not executed.

readInstance() Method

This section discusses two changes to the **readInstance()** method: "Additional Argument to readInstance()" and "Implementation of readInstance() for Group Nodes Changed."

Additional Argument to readInstance()

If you have node or engine subclasses with **readInstance()** methods, you have to add an extra argument to your **readInstance()** method; **SoFieldContainer::readInstance()** now takes the following arguments:

```
SoInput *in, unsigned short flags
```

If you call your base class's **readInstance()** method, you must pass it the *flags* argument. Otherwise, you can simply ignore this argument.

Implementation of readInstance() for Group Nodes Changed

If a group node reads children directly, you should have **readInstance()** code that emulates the **readInstance()** code of **SoGroup** and checks to see if children were written in the binary file format, as follows:

```
// Reads stuff into instance of SoGroup. Returns FALSE on
// error. Also deals with field data (if any), so this method
// should also be useful for most subclasses of SoGroup.
//
// Use: protected
SbBool
SoGroup::readInstance(SoInput *in, unsigned short flags)
{
    SbBool readOK = TRUE;
    // First, turn off notification for this node
    SbBool saveNotify = enableNotify(FALSE);
    // Read field info. We can't just call
    // SoNode::readInstance() to read the fields here
```

```
    // because we need to tell the SoFieldData that it's ok
    // if a name is found that is not a valid field name -
    // it could be the name of a child node.
    SbBool notBuiltIn; // Not used
    readOK = getFieldData()->read(in, this, FALSE,
                                  notBuiltIn);
    if (!readOK) return readOK;
    // If binary BUT was written without children (which can
    // happen if it was read as an unknown node and then
    // written out in binary), don't try to read children:
    if (!in->isBinary() || (flags & IS_GROUP))
        readOK = readChildren(in);

    // Re-enable notification
    enableNotify(saveNotify);
    return readOK;
}
```

This change was made so that binary files containing unknown group nodes with no children are correctly read.

copy() Method

Node copying now works correctly. For example, multiple references to a node (through children, fields, or connections) under the node to be copied are no longer copied individually; multiple instances of a single copy are created instead.

These changes required a change to the extender API. The changes affect you only if you have created a new node class and have implemented a **copy()** method for it that differs from that of its base class.

In Inventor 2.1, the **SoNode::copy()** method is no longer virtual. Instead, the **copy()** method creates an empty node of the correct type and then calls the virtual **copyContents()** method on it to copy from the existing node. The default implementation of **copyContents()** for **SoNode** copies the field values. The implementation for **SoGroup** copies the children as well. If your node has other (non-field, non-child) information that needs to be copied, redefine **copyContents()** for it. See "The copyContents() Method" on page 120 for more information.

Elements

This section looks at element changes and related information in Inventor 2.1, discussing the following topics:

- “Changes in Shape Hints Element Methods”
- “Changes in Binding Elements”
- “Changes in Texture Elements”
- “Changes in Material Elements”
- “Implementation of Node Override”

Changes in Shape Hints Element Methods

The **SoShapeHintsElement** methods **setFilled()**, **setClipped()**, **isFilled()**, **isClipped()**, **getDefaultIsFilled()**, and **getDefaultIsClipped()** are no longer supported.

Changes in Binding Elements

The DEFAULT and NONE values are obsolete for the nodes **SoMaterialBinding** and **SoNormalBinding**. The corresponding elements contain enum values for DEFAULT and NONE that make them the same as OVERALL for materials and PER_VERTEX_INDEXED for normals. If the code is compiled with -DIV_STRICT, these elements do not include values for DEFAULT and NONE, and any reference to them causes a compile-time error.

Changes in Texture Elements

The element **SoGLTextureQualityElement** is obsolete. Use **SoTextureQualityElement** instead.

The following elements (and their OpenGL counterparts) are replaced by **SoTextureImageElement** (and its OpenGL counterpart):

- **SoTextureBlendColorElement**
- **SoTextureModelElement**
- **SoTextureWrapSElement**
- **SoTextureWrapTElement**

Changes in Material Elements

Several elements that were responsible for material properties have been combined into two elements: **SoLazyElement** and its derived OpenGL version **SoGLLazyElement**.

- **SoLazyElement** is responsible for setting and getting material state.
- **SoGLLazyElement** is responsible for tracking the OpenGL state, and issuing send commands, invalidating caches, and so on.

Both elements are required, because some actions (for example callback actions) do not involve OpenGL. The following elements and their OpenGL counterparts are replaced by the lazy elements:

- **SoDiffuseColorElement**
- **SoSpecularColorElement**
- **SoAmbientColorElement**
- **SoEmissiveColorElement**
- **SoTransparencyElement**
- **SoShininessElement**
- **SoLightModelElement**
- **SoGLPolygonStippleElement**
- **SoCurrentGLMaterialElement**
- **SoGLColorIndexElement**

Implementation of Node Override

In Inventor 2.0, the **SoElement** base class automatically implemented override for all elements. Setting override on a node automatically caused an override flag to be set on every element that node overrode.

In Inventor 2.1, a node has to explicitly implement override. A new element, **SoOverrideElement**, has methods for getting and setting a bit for each element that can be overridden. Nodes must cooperate with one another by not doing anything if the bit for a particular element is set and by setting the bit for an element if the node's *Override* flag is set.

Many of the set methods on Inventor's built-in elements no longer take *SoNode* * parameters. In cases where the only reason for the *SoNode* * parameter was to test whether the node had its override set, the *SoNode* * parameter was removed. Elements that are subclasses of **SoReplacedElement** still take an *SoNode* * parameter in their set routines for cache dependencies.

This change improves performance and cache dependencies on override elements. Performance is improved because many nodes don't need override and they no longer have to pay for it. Cache dependencies are improved because the setting of the **SoOverrideElement** sets up the right dependencies.

If you created your own element and want to implement override behavior for it, you can either add and set override methods to the element (this is easier to implement) or implement another element that stores override information (this is better for caching).

Shape Nodes

This section provides information about implementing shape nodes with Inventor 2.1, discussing the following topics:

- "Generating Normals"
- "Managing the Material State"
- "Using the Material Bundle"
- "Getting a Bounding Box"

Generating Normals

The **SoNormalGenerator** class constructors take an argument that determines whether or not the polygons passed are in clockwise or counterclockwise order. Inventor now correctly generates normals for built-in shapes when the **SoShapeHints** node sets the vertex ordering to CLOCKWISE.

You don't have to change extender shapes that use **SoNormalBundle** to generate normals for them. You do have to modify extender shapes that use the **SoNormalGenerator** class directly to get the vertex ordering out of the **ShapeHintsElement** and pass TRUE or FALSE to the **SoNormalGenerator** constructor.

Managing the Material State

Inventor 2.1 has a new, more efficient mechanism for managing the material state during scene traversal. This section first discusses the new **SoLazyElement** and **SoGLLazyElement**, then provides instructions for converting **GLRender()** methods that use the lazy elements.

Using SoLazyElement and SoGLLazyElement

The **SoMaterialBundle** is still supported for backward compatibility, but you should use **SoLazyElement** and **SoGLLazyElement** for improved performance. You also need to use the lazy elements if you are issuing any of the following OpenGL calls:

- **glColor()**
- **glMaterial()**
- **glColorMaterial()**
- **glEnable()** or **glDisable()**, with the following arguments:
  - GL_COLOR_MATERIAL
  - GL_BLEND
  - GL_LIGHTING
  - GL_POLYGON_STIPPLE
- **glPushAttrib()** or **glPopAttrib()**, with the following bits:
  - GL_ENABLE_BIT
  - GL_LIGHTING_BIT
  - GL_POLYGON_BIT
  - GL_POLYGON_STIPPLE_BIT

The lazy element tracks the OpenGL state for all components of the lazy element. If you issue OpenGL calls that change that state, you must call **SoGLLazyElement::reset()** to inform the lazy element that the OpenGL state has changed.

It is important to keep track of **state::push()** and **pop()** calls that occur while rendering, because the identity of the current lazy element will probably change during a push and pop. This can cause problems if **SoMaterialBundle** is used as in the Inventor 2.0 example code */usr/share/src/inventor/examples/Toolmaker/02.Nodes/Pyramid.c++*.

The **SoShape::beginSolidShape()** and **SoShape::endSolidShape()** methods are unnecessary and actually can cause problems in that example, because the destructor for the material bundle is invoked after a **pop()**. This potentially invalidates state that is no longer current. The example code has been revised to work correctly in Inventor 2.1 (see also Appendix A, "Creating a Node").

How to Convert GLRender() Methods

This section provides step-by-step instructions for converting a shape node's **GLRender()** method to use the lazy elements.

The shape nodes are responsible for managing transparency state, managing **glShadeModel**, issuing **glColorMaterial()**, and issuing OpenGL calls to send graphics state.

- In the simplest case (one color per shape), you need only call **SoGLLazyElement::sendAllMaterial()**, once, prior to rendering the shape.
- If there are multiple colors or transparencies in the shape, you may need to do more.

In the most general case, follow these steps:

1. If your node is setting a new value in the lazy element state, call **SoLazyElement::setColorMaterial()**, **setBlending()**, and so on.

   When **ColorMaterial** is TRUE, **glColor()** calls modify the current GL_DIFFUSE color. By default, **ColorMaterial** is FALSE. You should only set it if it needs to be TRUE, and be sure to set it back to FALSE at the end of rendering.

2. After you've set all values, but before any geometry has been sent to OpenGL, issue one of the following calls:
   - **SoGLLazyElement::sendAllMaterial()**
   - **SoGLLazyElement::sendNoMaterial()**
   - **SoGLLazyElement::sendOnlyDiffuseColor()**

The **sendAllMaterial()** call ensures that all of the state managed by the lazy element is current in OpenGL. If the light model is BASE_COLOR, use **SendOnlyDiffuseColor()** instead, to avoid unnecessary OpenGL calls to set ambient colors, specular colors, and so on. If you are providing your own **glMaterial()** or **glColor()** calls, use **sendNoMaterial()**. This ensures that the OpenGL state is current with respect to transparency, light model, color material, and so on.

3. To send additional materials, for example, PER_VERTEX or PER_FACE, to OpenGL, call **SoGLLazyElement::sendDiffuseByIndex()** or issue OpenGL commands.
4. The OpenGL shade model is by default GL_SMOOTH. If you need to have flat shading, invoke **glShadeModel(GL_FLAT)** before rendering, but be sure to set it back to GL_SMOOTH at completion of **GLRender()**.
5. If you are managing transparency, it may be necessary to call **glEnable()** or **glDisable()** with arguments GL_BLENDING or GL_POLYGON_STIPPLE. Be sure to issue **SoGLLazyElement::reset()** on the transparency components of the **SoGLLazyElement**, namely TRANSPARENCY and BLENDING, if you called **glEnable()** or **glDisable()** to affect transparency.
6. If you use stipple transparency, only the first transparency value in the state is used to determine the transparency value in the stipple pattern. If you use other forms (additive or blending transparency), and if you provide as many transparency values as diffuse colors, then transparency varies across the shape, with a transparency being associated with each diffuse color.
7. If you do not want to issue OpenGL calls but need to have multiple colors sent while rendering a shape, use the **sendDiffuseByIndex()** routine for all color send calls after the first. You can use this routine to send a specific diffuse color and/or transparency from the state. When you invoke **lazyElt::sendDiffuseByIndex()**, make sure the lazy element that you are using is current, that is, that there is no **push()** or **pop()** between:

```
SoGLLazyElement* lazyElt =
      (SoGLLazyElement*)SoLazyElement::getInstance(state);
```

and

```
lazyElt->sendDiffuseByIndex();
```

8. If you issued **glColor()** or **glMaterial()** calls, or call **sendDiffuseByIndex()** during rendering, you have to inform the lazy element that the OpenGL state is not the same as the Inventor state. Call **SoGLLazyElement::reset(***state, bitmask***)** at the completion of rendering, with *bitmask* the logical OR of one or more of the following bitmasks, depending on which material you sent:

   - **SoLazyElement::DIFFUSE_MASK**
   - **SoLazyElement::AMBIENT_MASK**
   - **SoLazyElement::EMISSIVE_MASK**
   - **SoLazyElement::SPECULAR_MASK**
   - **SoLazyElement::SHININESS_MASK**

   Make sure that the lazy element to which **reset()** is applied is current. Don't issue a **push()** or **pop()** between obtaining the lazy element and applying **reset()**.

   After rendering, call **SoLazyElement::setColorMaterial(FALSE)** to disable **glColorMaterial()**, if color material was enabled.

Using the Material Bundle

For backward compatibility, the material state of a shape can be managed (as in Inventor 2.0) by use of **SoMaterialBundle**. You have to set up the **GLRender()** method of the shape as follows:

Constructor:	SoMaterialBundle mb(action);
Prior to first rendering:	mb.sendFirst()
For each new material:	mb.send(index)
Destructor:	(implicitly invoked at the end of **GLRender()**)

For correct use of the material bundle, there should not be a **push()** or **pop()** of state between **mb.sendFirst()** and the invocation of the destructor, for example, consider the following code fragment:

```
beginSolidShape(action)
{//Scope material bundle
   SoMaterialBundle mb;
   ...
}
endSolidShape(action);
```

Do not, for example, invoke **SoShape::endSolidShape()** until after the end of the scope of the material bundle because it pops the state, for example:

```
{//Don't do it this way
 SoMaterialBundle mb;
 beginSolidShape(action)
 ...
 endSolidShape(action)
}
```

Getting a Bounding Box

If an Inventor 2.1 shape consists of lines and/or points, you must implement a **getBoundingBox()** method that calls first the parents method and then:

```
SoBoundingBoxCache::setHasLinesOrPoints(action -> getState())
```

See "Getting a Bounding Box" on page 93 for more detailed information.

Material Property Nodes

If you implement a property node to manage material state (colors, light model, or transparency), note that in the **doAction(action)** method (that is, when the node is traversed), material properties are set in the state by **SoLazyElement** methods (see *SoLazyElement.h*):

- **setTransparency**
- **setDiffuse**
- **setAmbient**

- **setSpecular**
- **setShininess**
- **setEmissive**
- **setPacked**
- **setLightModel**
- **setColorIndices**

It is not necessary send materials to OpenGL during traversal of property nodes in Inventor 2.1. For efficiency, the lazy element delays such sends until they are required by the shape node.

If you set either the diffuse color or the transparency, but not both, use **SoColorPacker** to pack the material you are setting into a packed color in the state. If you are setting both diffuse color and transparency, you should pack them both into a packed color and use **SoLazyElement::setPackedColor** to set them in the state. See the *Glow* example (Example A-2 on page 84) for an illustration of how **SoColorPacker** is used.

Before issuing a **set()** on an element, if that element can be overridden, check **SoOverrideElement** for the status of the override on that element. It is now the responsibility of the application to test for override, and to not set the corresponding property in the state if override is set.

Engine Evaluation

If you've written engines that do not use the SO_ENGINE_OUTPUT macro but instead directly write into the fields they are connected to then you have to make the following change: before writing to a connected field, you must check if the field is not read only (**!field::isReadOnly()**) instead of checking if the field is engine modifying (**field::isEngineModifying()**). This change was made as part of the optimization of engine notification and evaluation.

Chapter 6

Optimizing Open Inventor Applications

This chapter explains how to determine what is limiting the performance of your Open Inventor application, and provides suggestions on how to improve its performance.

Note: This chapter was previously available as a small booklet. The information has been updated to Inventor 2.1 as appropriate.

The chapter discusses the following topics:

- "Benchmarking Tips" on page 48—Describes a process for effectively investigating and improving performance—discovering where an application is spending its time is not a matter of random trial and error.
- "Optimizing Rendering" on page 52—Provides for each step in the rendering process:
  - Information about that step.
  - A method for determining how much time your application is spending in that step.
  - Techniques for reducing that time.
- "Optimizing Everything Else" on page 68—Suggests ways of structuring your scene and application for maximum performance when doing tasks other than rendering (for example, picking or modifying the scene).

For more information on Open Inventor programming in general and on specific nodes, see *The Inventor Mentor*.

Benchmarking Tips

Like fixing bugs or eliminating memory leaks, performance tuning is a necessary chore during application development. Proper organization and planning can speed up this chore and make it more pleasant.

This section looks at the steps to take when optimizing your application. discussing these topics:

1. "Setting Performance Goals"
2. "Measuring Performance"
3. "Determining Bottlenecks"
4. "Modifying Your Application to Reduce Bottlenecks"
5. Finally, measuring your application against your goal again and repeating steps 2, 3 and 4 above until performance meets your goal (see "Are You Finished Yet?" on page 51).

Setting Performance Goals

Setting a performance goal helps you use your time wisely. Typically, you should decide on a desired frame rate, such as running at 20 frames per second with a particular scene. A reasonable performance goal for interactive programs is a frame rate of at least 10 frames per second. Most users find that frame rate acceptable for most tasks (more is always better, of course).

When setting a goal, keep in mind the capabilities of your hardware. If the absolute top speed for drawing polygons on your system is 60,000 unlit, single-color triangles per second, don't try to get 10 frames per second while drawing 6,000 lit, color-per-vertex triangles. Write short OpenGL benchmark programs, or feed test scene graphs to *ivview -p* to help set your expectations.

Measuring Performance

It is important to have an objective way of measuring your application's performance. You're likely to waste time on insignificant optimizations if you just watch your application run and try to see if it *seems* faster.

Adding code to your application that measures the number of polygons in your scene and how fast they are being rendered is fairly simple; see, for example, the source code for *ivview* in */usr/share/src/Inventor/tools/ivview*.

The *osview* utility can also be useful. The "swapbuf" number in the Graphics section tells you how many frames per second your application is getting (assuming that your application is double-buffered, and that it is the only double-buffered application running).

Note: Don't confuse *osview* with its graphical counterpart, *gr_osview*, which doesn't have this feature.

Be sure to keep good records of your application's performance before you start optimization. Comparing "before" and "after" performance numbers ensures that you are not making things worse.

Determining Bottlenecks

Most applications spend most of their time executing a small part of the code. Optimizing a procedure that is taking up only 5% of the total time is probably not worthwhile: even if you manage to double the performance of the procedure, the application will speed up by only 2.5%. In fact, on some systems graphical operations can occur in parallel. For example, filling in polygons and transforming polygon vertices might occur at the same time. If the bottleneck is in the vertex transformation stage, increasing the pixel fill time may not increase performance at all! Find the bottlenecks first, and then work on improving them.

Finding bottlenecks is an experimental science. You should first come up with a theory on where the bottleneck might be, then devise an experiment that proves or disproves that theory. Create experiments that isolate one small part of your application's performance, and make sure you understand what you are measuring every time you run an experiment.

"Optimizing Rendering" and "Optimizing Everything Else" describe frequently encountered bottlenecks, show how to determine the amount of time your application is spending in each of them, and give suggestions on improving them. The following topics are discussed:

- "Correcting Window Clear Bottlenecks"
- "Improving Traversal Performance"
- "Optimizing Pixel Fill Operations"
- "Correcting Problems ivperf Does Not Measure"
- "Correcting Culling Bottlenecks"
- "Correcting Level of Detail Bottlenecks"
- "Optimizing Memory Usage"
- "Optimizing Action Construction and Setup"
- "Decreasing Notification Overhead"
- "Picking and Handling Events"

Feel free to start with bottlenecks you suspect are responsible for the most noticeable slowdown. You can look at the sections in any order; just make sure you always know what you are measuring and keep good records of your experiments.

Modifying Your Application to Reduce Bottlenecks

When you apply a performance optimization, make sure that the modification is really an improvement: don't assume that all the suggestions made in this (or any other) document automatically apply to your application. For example, render culling usually increases performance. However, if you have an application in which all objects are always visible, render culling actually hurts performance.

Again, keep good records. Record what you do and how much it improves performance. Try to minimize the number of things you change at any one time; for example, if you make two “optimizations” and performance goes up by 10%, the speedup might be caused by a 5% improvement for each optimization, or might be caused by a 100% speedup caused by one optimization and a 90% slowdown caused by the other! It is tempting to read a document like this, make lots of changes, then see if the application gets faster. This not only wastes time, it can also be counter-productive.

Are You Finished Yet?

One of the most frustrating things about optimizing an application’s performance is that it can be difficult to determine when you are done. Once you have successfully eliminated one bottleneck, something else becomes the factor limiting performance. Before spending more time on optimization, you should ask yourself:

- Did you meet your performance goal? If you did, go home early. If not, try to find other bottlenecks, consider eliminating features that hurt performance, or reexamine your goals.
- Is your application running at 60 or 72 frames per second? Double-buffered programs never render faster than the refresh rate of your monitor.
- Do you need to experiment on different systems? Different systems have different bottlenecks; on a system with very fast graphics, the bottleneck is more likely to be either in the application’s code or in Inventor’s code. On a system with slow graphics and a fast CPU, the bottleneck is more likely to be inside the OpenGL calls. If your application will be used on different types of systems, make sure performance is acceptable on all of them.

The Five Performance Commandments

1. **Be scientific**.
2. **Keep good records**.
3. **Find bottlenecks**.
4. **Change one thing at a time**.
5. **Test all the types of systems your application supports**.

Optimizing Rendering

The main goal of performance tuning is to make the application look and feel faster. However, just because the goal is to make the application render faster, don't assume that rendering is the bottleneck.

Determining Whether Rendering Is the Problem

To find out whether rendering is the problem, modify your application so that it does everything it normally does except render, and then measure its performance. An easy way of getting your application to do everything but rendering is to insert an **SoSwitch** node with its **whichChild** field set to SO_SWITCH_NONE (the default) above your scene. So, for example, modify your application's code from:

```
myViewer->setSceneGraph(root);
```

To:

```
SoSwitch *renderOff = new SoSwitch;
renderOff->ref();
renderOff->addChild(root);
myViewer->setSceneGraph(renderOff);
```

This experiment gives an upper limit on how much you can improve your application's performance by increasing rendering performance. If your application doesn't run much faster after this change, then rendering is not your bottleneck. See "Optimizing Everything Else" on page 68 for information on optimizing the rest of your application.

Isolating Rendering

If you have determined that your application is spending a significant amount of time rendering the scene, the next step is to isolate rendering from the rest of the things your application does. This makes it easier to find out where the bottleneck in rendering occurs. The easiest way to isolate rendering is to write your scene to a file and then use the *ivperf* program to perform a series of rendering experiments.

The code for writing your scene may look like the following:

```
SoOutput out;
if (!out.openFile(“myScene.iv”)) { ... error ... };
SoWriteAction wa(&out);
wa.apply(root);
```

Using the ivperf Utility to Analyze Rendering Performance

The *ivperf* utility reads in a scene graph and analyzes its rendering performance. It estimates the time spent in each stage of the rendering process while rendering the scene graph.

The process of rendering a single frame can be decomposed into five main stages:

1. Clearing the graphics window
2. Traversing the Inventor scene graph
3. Changing the graphics state (including materials, transformations, and textures)
4. Transforming vertices in the graphics pipeline
5. Filling polygons

The sum of the times spent in these stages does not, in general, equal the total time it takes to render the scene. Depending on the underlying hardware platform and graphics pipeline, some or all of the above can overlap with each other. Thus, completely eliminating one of the stages does not necessarily speed up the application by the time taken by that stage. *ivperf* takes this into account; it answers questions of the type "if I could completely eliminate xxx from my scene, how much faster would rendering be?" For example, if *ivperf* indicates that 50% of your time is spent changing the material graphics state, then making your entire scene a single material would make it render twice as fast. Knowing that materials are taking up a significant part of your rendering time, you can then concentrate on minimizing the number of material changes made by your scene.

If you have created your own node classes, either create DSO's for them (see "DSO Directories and Versions" on page 22) or call their **initClass()** methods just after the call to **SoInteraction:init()** in the *ivperf* source and link their *.o* files into *ivperf*.

The camera control used by *ivperf* is simplistic: it calls **viewAll()** for the scene and just spins the scene around in front of the camera when benchmarking. If you have a sophisticated walk-through or fly-through application that uses level of detail and/or render culling, modify *ivperf* so that its camera motion is more appropriate for your application. For example, have *ivperf* use the following little scene instead of just **SoPerspectiveCamera**:

```
TransformSeparator {
    Rotor { rotation 0 1 0 .1 speed .1 }
    Translation { translation 100 0 0 }
    PerspectiveCamera { nearDistance .1 farDistance 600 }
}
```

ivperf correctly reports the performance of changing scenes, as long as you give it enough information. It automatically deals with scenes containing engines and animation nodes, but if you are using an **SoSensor** to modify the scene, you should mark nodes that your application frequently changes by giving them the special name "NoCache." For example, if your application is frequently changing a transformation in the scene, the transformation should appear in the file given to *ivperf* as:

```
DEF NoCache Transform { }
```

Correcting Window Clear Bottlenecks

The first step in the rendering process is clearing the window. It is easy to forget this step, but depending on the size of your application's window and the type of system you are running on, clearing the window can take a surprisingly long time. If your application's main window is typically 1000 by 1000 pixels, run *ivperf* like this:

```
ivperf -w 1000,1000 myScene.iv
```

ivperf performs many different rendering experiments, and eventually prints information on each rendering stage.

For example, on an Indigo2™ Extreme™ running IRIX 5.3, *ivperf* reports that for rendering a simple cube in a 1000 by 1000 pixel window 46% of the time is spent clearing the window.

Unfortunately, if clearing the window takes too much time, there is not a lot you can do. One possibility is to make the window's default size smaller (while still allowing users to resize the window if necessary).

Improving Traversal Performance

After running *ivperf*, you know how much time your application spends clearing the color and depth buffers. The next experiment is designed to find out how much time Inventor spends traversing your scene. Traversal is the process of walking through the scene graph, deciding which render method needs to be called at each node. Inventor automatically caches the parts of your scene that aren't changing and that are expensive to traverse, building an OpenGL display list and eliminating the traversal overhead.

If most of your scene is changing, or if your scene is not organized for efficient caching, Inventor may not be able to build render caches, and traversal might be the bottleneck in your application. *ivperf* measures the difference between rendering your scene with nothing changing, and rendering with the camera, engines, and nodes named "NoCache" changing.

If traversing the scene is a bottleneck in your program, there are several ways of reducing the traversal overhead:

- Reduce the number of nodes in the scene. For example, eliminate **SoSeparator** and **SoGroup** nodes that have only one child by replacing them with the child.
- Use the **vertexProperty** field of the vertex-based shapes to specify coordinates, normals, texture coordinates, and colors. (See "SoVertexProperty Node" on page 24).
- Beware of features that require multiple traversals of the scene graph for each render update. For example, avoid using accumulation buffer antialiasing and **SoAnnotation** nodes, and use the transparency type **SoGLRenderAction::SCREEN_DOOR**.
- Organize your scene graph for caching. The next section discusses ways of doing this. If you are using **SoLOD** nodes or render culling, also see "Correcting Culling Bottlenecks" on page 66 and "Correcting Level of Detail Bottlenecks" on page 67 for hints on optimizing those features, which *ivperf* also reports as part of caching behavior.

Organizing the Scene for Caching

You may be able to organize your scene so that Inventor can build and use render caches even if part of the scene is changing. Note that the following things inhibit caching:

- Changing fields in the scene destroys caches inside all **SoSeparator** nodes above the node that changed. Even fields that do not affect rendering, such as fields in the **SoLabel** or **SoPickStyle** nodes, destroy caches if they are changed.

  Tip: You can disable notification on these nodes using the **SoNode::enableNotify()** method to keep changes to them from destroying caches.
- The **SoLOD** node (and the older **SoLevelOfDetail** node) breaks caches above it whenever either the camera or any of the matrix nodes affecting it change. Make the children of the **SoLOD** node **SoSeparator** nodes, so that they will be cached. See "Correcting Level of Detail Bottlenecks" for more information on efficient use of the **SoLOD** node.

- Any shape using SCREEN_SPACE complexity breaks caches above it whenever the camera or any of the matrix nodes affecting it change.
- The **SoText2** node breaks caches above it whenever the camera changes (in order to correctly position and justify each line of text, it must perform a calculation based on the camera). Since most applications change the camera frequently, try to separate **SoText2** nodes from the other objects in your scene, to allow the other objects to be cached.

  Tip: In Inventor 2.1, single-line, left-justified **SoText2** nodes do not break render caches.
- Changing the override status of properties at the top of the scene, or changing global properties such as **SoDrawStyle** or **SoComplexity** that affect the rest of the scene, inhibits efficient caching. **SoSeparator** nodes build multiple render caches (by default, a maximum of two) to handle cases in which a small set of global properties are changed back and forth, but you should avoid continuously changing a global property; for example, putting an engine on the value field of an **SoComplexity** node at the top of your scene is bad for caching.

For more information on Inventor's render caching, see Chapter 9 of *The Inventor Mentor*.

Improving Material Change Bottlenecks

If *ivperf* reports that material changes are the rendering bottleneck, try the following:

- Use fewer material nodes. Group objects by material, and use one material node for several objects.

  Tip: When the *ivfix* utility rearranges scene graphs, it groups objects by material.
- Changing between materials with different shininess values is much more expensive than changing any of the other material properties.
- If you are using shapes with a **materialIndex** field, try to sort their parts by material index to minimize material changes. For example, try to change:

```
IndexedFaceSet { materialIndex [ 0,1,0,1,0,1,0,1 ] ... }
```

to:

```
IndexedFaceSet { materialIndex [ 0,0,0,0,1,1,1,1 ] ... }
```

This works only for PER_PART_INDEXED or PER_FACE_INDEXED material bindings.

Optimizing Transformations

For transformation, *ivperf* reports two numbers: the overhead of changing the OpenGL transformation matrix between rendering shapes and the time it takes to transform the vertices in your scene through that matrix. This section helps with the former, giving suggestions on how to make Inventor execute fewer OpenGL matrix operations. See "Optimizing Vertex Transformations" on page 61 for hints on optimizing the transformation of vertices.

- To measure how much time transformations might be taking, *ivperf* temporarily removes all transformations from your scene and then measures how much faster it runs. Beware! This sometimes gives unreliable results; for example, if all your objects become very large or very small without the transformations, then more (or less) time may be spent filling in pixels. If your scene uses render culling, removing the transformations makes more (or fewer) of the objects culled, distorting the results reported by *ivperf*.
- Use **SoRotation**, **SoRotationXYZ**, **SoScale**, or **SoTranslation** nodes instead of the general **SoTransform** node. However, don't bother doing this if you have to replace the **SoTransform** node with more than one of the simpler nodes to get the same transformation.

Performance Tip for Face Sets

For best performance when creating **SoFaceSet** and **SoIndexedFaceSet** shapes, arrange all the triangles first, then quads, and then other faces.

Optimizing Textures

If your scene contains textures, *ivperf* reports two numbers: the time you would save if you could turn off textures completely, and the time you would save if you could make your scene use only one texture. On systems with texturing hardware, the number of textures used can dramatically affect performance; see "Optimizing Texture Management" on page 59 for hints on optimizing texture management. On systems without texture mapping hardware, the bottleneck is probably filling in the textured polygons.

Inventor 2.1 automatically does two things to speed up rendering on systems without texture mapping hardware:

- Inventor's viewers display the scene untextured during interaction by default.
- Inventor uses lower-quality filters for minifying or magnifying textures.

Optimizing Texture Management

If *ivperf* reports a lot of time is spent in texture management, then you are running out of hardware texture memory. Try the following:

- Use smaller textures. Use the *izoom* utility to scale down the images you are using; inadvertently using one big image can easily fill up texture memory on many systems.
- Make textures a power of 2 wide and high. Textures of those dimensions (for example 128 x 64 instead of 129 x 70) make startup faster.
- Reuse nodes. Inventor allows you to modify a texture once it has been read into your application (using the image field of **SoTexture2**), and to change the search path for textures (using methods on **SoInput**). It therefore does not use the same texture memory for two different **SoTexture2** nodes with the same filename field. Be sure to reuse the same **SoTexture2** node instead of creating another node with the same filename.

For example, this scene is inefficient:

```
Separator {
    Texture2 { filename foo.rgb }
    Cube { }
}
Sphere { }
Separator {
    Texture2 { filename foo.rgb }
    Text3 { string “Hello” }
}
```

This scene uses texture-memory efficiently:

```
Separator {
    DEF foo Texture Texture2 { filename foo.rgb }
    Cube { }
}
Sphere { }
Separator {
    USE foo
    Text3 { string “Hello” }
}
```

- Textures are not shared if they are below an **SoSeparator** with **renderCaching** turned on. Textures use less texture memory by building an OpenGL display list or texture object containing the OpenGL texture commands. However, if an OpenGL display list is already created the first time an **SoTexture2** node is traversed, it must add the texture commands to the already open display list. Inventor’s automatic caching algorithm handles this correctly; it is only a problem if you turn caching on explicitly. For example, this scene uses twice as much texture memory with the **renderCaching** field of the **SoSeparator** set to ON:

```
Separator { renderCaching ON # BAD
    DEF TEX Texture2 { filename foo.rgb }
    Cube { }
... more stuff, other textures, etc...
    USE TEX
    Cube { }
}
```

- Use **SoLOD** nodes to create simpler versions of your objects that are not textured or use smaller texture images when the objects are far away.
- Use render culling so the textures for textured objects outside the view volume are not used. For example, imagine a scene that contains 100 textured objects (each with a unique texture), but only 10 of them are in the view volume at any given time. When the scene is rendered, only 10 of the textures need to be in texture memory at any given time, resulting in much better texture management performance.

Using Lights Efficiently

If the scene given to *ivperf* contains light sources, *ivperf* informs you how expensive they are compared to rendering your scene with just a single directional light. If *ivperf* reports that lights are a significant performance bottleneck, try to use fewer light sources, and use simpler lights (a **DirectionalLight** is simpler than a **PointLight**, which is simpler than a **SpotLight**). If possible, put lights inside separators so that they affect only part of the scene, increasing performance for the rest of the scene.

Optimizing Vertex Transformations

If *ivperf* reports that vertex transformations (which include per-vertex lighting calculations) take up a significant portion of the time it takes to render a frame, you can do the following to optimize per-vertex operations:

- Use fewer vertices in your objects. Use **SoComplexity** to turn down complexity for Inventor's primitive objects. If you are using a system with hardware-accelerated texturing, texturing can be used to add visual complexity with very few vertices.

- Create less detailed versions of your objects and use **SoLOD** nodes so that fewer vertices are drawn when objects are small. Use an empty **SoInfo** node as the lowest level of detail so that objects disappear when they get very small. A good rule of thumb for choosing levels of detail is that the switch between levels of detail should be fairly obvious if you are concentrating on the object; for most applications, the user concentrates on objects in the foreground and does not notice background objects "popping" between levels of detail. Beware that **SoLOD** nodes cause smaller caches to be built, which may slow down traversal. See "Correcting Level of Detail Bottlenecks" for more information on efficient use of level of detail.
- Make your vertices simpler. Try to use OVERALL rather than PER_VERTEX material binding. Turn off fog. Note that these suggestions are system-specific; on systems with a lot of hardware for accelerated rendering, fogged vertices may be no slower than plain vertices. Be sure to do a quick *ivperf* test before spending time modifying your application.
- Make sure you are not turning on two-sided lighting unnecessarily; avoid **SoShapeHints** nodes that:
  - set **vertexOrdering** fields to COUNTERCLOCKWISE or CLOCKWISE and
  - set **shapeType** fields to UNKNOWN_SHAPE_TYPE
- If parts of your scene do not require lighting, use an **SoLightModel** node set to model BASE_COLOR to turn off lighting for those parts of the scene. However, be aware that turning lighting on and off can itself become a bottleneck if done too often.
- If you are using **SoFaceSet** or **SoIndexedFaceSet**, try using *ivfix* to convert them into **SoIndexedTriangleStripSet**, which draws more triangles with fewer vertices. Note that *ivfix* cannot create a mesh if your objects have sharp facets or PER_FACE material or normal bindings.
- Watch out for expensive primitives with lots of vertices, like **SoText3** and **SoSphere**. *ivperf* reports the number of triangles in your scene; make sure the number is reasonable for your desired performance.

- Organize your scene graph so that objects that are close to each other spatially are under the same **SoSeparator**, and turn on render culling so that Inventor won't send those objects' vertices when the objects are not in view. See *The Inventor Mentor*, Chapter 9, for more information on render culling.
- See "Making Inventor Produce Efficient OpenGL" on page 64 for hints on making Inventor produce more efficient OpenGL calls.

Optimizing Pixel Fill Operations

A common bottleneck on low-end systems is drawing the pixels in filled polygons. This is especially common for applications that have just a few large polygons, as opposed to applications that have lots of little polygons.

If *ivperf* reports that a large percentage of each frame is spent filling in pixels, try to optimize your scene as follows:

- Render your scene, or parts of your scene, in wireframe or as points when possible. Viewers have "move wireframe" and "move points" modes built in for exactly this case.
- Some systems can fill flat-shaded polygons faster that Gouraud-shaded polygons. Triangle strips and quad meshes set **shademodel(FLAT)** if they have PER_FACE normals and don't have PER_VERTEX materials (and vice versa).
- SCREEN_DOOR transparency (the default) is faster than blended transparency on some systems (it is slower on other systems). Use the **setTransparencyType()** method on either **SoXtRenderArea** or **SoGLRenderAction** to change the transparency type.

Correcting Problems ivperf Does Not Measure

There are several performance problems that *ivperf* doesn't catch. The following sections describe them, and give hints on how to improve them.

Making Inventor Produce Efficient OpenGL

If your application is rendering only 10 frames per second with 1,000 triangles per frame, and you know that your graphics hardware is capable of rendering 100,000 triangles per second (10,000 triangles per frame at 10 frames/second), and *ivperf* reports that your bottleneck is vertex transformations, then your problem might be that Inventor is not making efficient OpenGL calls.

Inventor is much more efficient at rendering multiple triangles if they are all part of one node. For example, you can create a multifaceted polygonal shape using a number of different coordinate and face set nodes, as shown in the lower half of Figure 6-1. A much better technique is to put all the coordinates for the polygonal shape into one **SoCoordinate** or **SoVertexProperty** node, and the description of all the face sets into a second **SoFaceSet** node, as shown in the upper half of Figure 6-1.

Tip: The *ivfix* utility program collapses multiple shapes into single triangle strip sets.

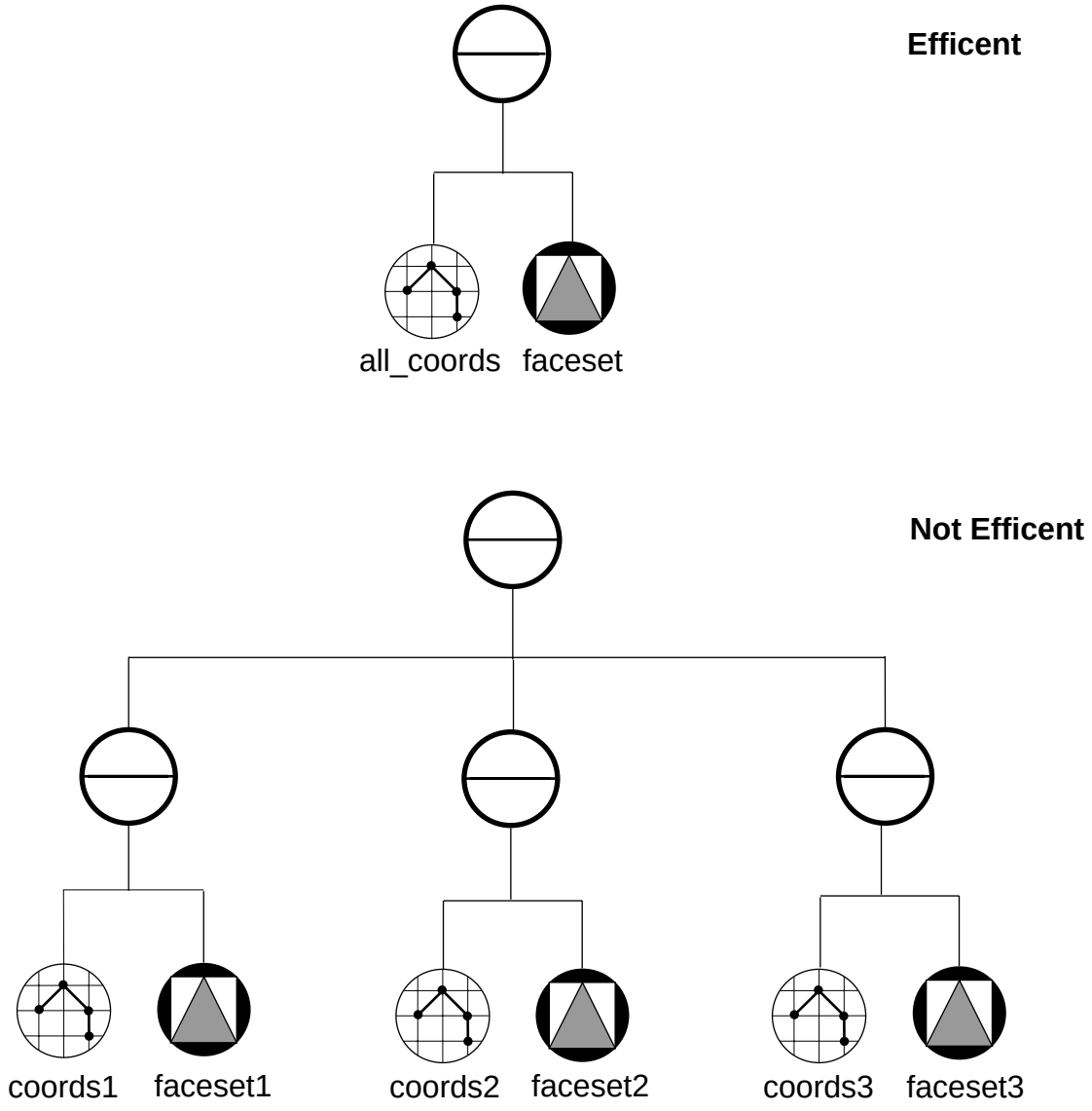

Figure 6-1 Condensing Face Sets Into Fewer Nodes

Using fewer nodes to get the same picture reduces traversal overhead for scenes that cannot be cached. Note also that Inventor optimizes on a node by node basis and generally can't optimize across nodes.

An **SoFaceSet** or **SoIndexedFaceSet** has special code for drawing 3 and 4-vertex polygons. To take advantage of that, you must arrange the polygons so that the 3-vertex polygons (if any) are first in the **coordIndex** array, followed by the 4-vertex polygons, followed by the polygons with more than 4 vertices.

For some applications, consider implementing your own nodes that implement the functionality of a subgraph of your scene. For example, a molecular modeling application might implement a **BallAndStick** node with fields specifying the atoms and bonds in a molecule, instead of using the more general **SoSphere**, **SoCylinder**, **SoMaterial**, **SoTransform**, and **SoGroup** nodes. If the molecular modeling application changes the molecule frequently so Inventor cannot cache the scene, using a specialized node could make traversal orders of magnitude faster (for example, a simple water molecule scene graph with three atoms and two bonds might consist of 20 nodes; replacing this with a single **BallAndStick** node would make traversal 20 times faster). The **BallAndStick** node could also perform application-specific optimizations not done by Inventor, such as not drawing bonds between spheres whose radii were large enough that they intersected, sorting the spheres and cylinders by color, and so on. See *The Inventor Toolmaker* for complete information on implementing your own nodes.

Correcting Culling Bottlenecks

If your application uses render culling, it may be spending most of its time deciding whether or not objects should be culled. *ivperf* lumps this in with bad caching behavior. To find out whether this is the case, use *prof*, *pixie*, or the CaseVision™/WorkShop Performance Analyzer tools to look for a lot of CPU time being spent in the **SoSeparator::cullTest()** or **SoBoundingBoxAction::apply()** routines. See the reference pages for *prof*, *pixie*, or *cvspeed* for information on using these tools.

If a large percentage of the rendering time is spent doing cull tests, try to reorganize your scene so that more triangles are culled for each culling **SoSeparator**. For example, if you have a city scene with thousands of buildings, it may be better to perform one cull test for each city block rather than the thousands of cull tests needed to decide whether or not each individual building is visible. Doing this also allows Inventor to build larger render caches, which may increase traversal speed.

Also, remember that render culling breaks render caches when the camera or transformation matrices change, so double-check to make sure that no **SoSeparator** nodes above an **SoSeparator** doing render culling have their **renderCaching** fields set to ON.

If a lot of time is being spent inside **SoGetBoundingBoxAction::apply()**, something is breaking bounding box caches.

Correcting Level of Detail Bottlenecks

If your application uses **SoLOD** nodes, it might be spending a significant amount of time deciding which level of detail should be drawn. One way of testing to see if this is the case is to temporarily replace all of the **SoLOD** nodes in your scene with **SoSwitch** nodes set to traverse the highest level of detail. Then run *ivperf* again and compare the results. If the **SoSwitch** node scene is much faster, try doing the following:

- Try to group objects so that one level of detail test determines the level of detail for several objects. For example, if you have a group of 10 buildings that are near each other, use one level of detail node instead of 10 level of detail nodes. Doing this also makes it easier for Inventor to build larger render caches, which may increase performance by increasing traversal speed.
- Remember that level of detail nodes break render caches when the camera or transformation matrices change, so make sure that no **SoSeparator** nodes above an **SoLOD** have their **renderCaching** fields set to ON.
- Make sure you use the **SoLOD** node introduced in Inventor 2.1 instead of the **SoLevelOfDetail** node. The **SoLOD** node is more efficient because it uses the distance to a point as the switching criterion. See the reference page for more detail.

Making Your Application Feel Faster

Sometimes it is worthwhile to sacrifice features temporarily to make your application seem faster to the user. Inventor has several features that make this easier:

- Use the **SoGLRenderAction::setAbortCallback()** method to interrupt rendering before the entire scene has been drawn. For this to be most effective, you must organize your scene so that the most important objects are drawn first, and you should abort only when it is important that rendering happen quickly, even if the rendering is not complete, such as when the user is interactively manipulating the scene.
- Use one of the "Move ..." draw styles if you are using a viewer, so that a simpler version of the scene is drawn when the user is interacting with the viewer.
- Use the start and finish callbacks of manipulators and components to temporarily modify the scene to make it simpler while the user is interacting with it.

Optimizing Everything Else

If you have determined that rendering is not your bottleneck, or if you have already optimized rendering as much as possible and a significant amount of time is still being spent doing something other than rendering, it's time to look for other bottlenecks. This section helps you find other bottlenecks, and suggests Inventor-specific things to look for by discussing the following:

- "Useful Tools"
- "Optimizing Memory Usage"
- "Looking at CPU Usage"
- "Optimizing Action Construction and Setup"
- "Decreasing Notification Overhead"
- "Picking and Handling Events"

Useful Tools

The standard performance analysis tools (*prof*, *pixie*, or the CaseVision/WorkShop Performance Analyzer) make performance analysis of the non-graphics part of your application easy. See the reference pages for *prof*, *pixie*, or *cvspeed* for information on using these tools.

Optimizing Memory Usage

First, make sure your application isn't running out of physical memory by running *gr_osview* with the *-a* flag and looking for "swap" in the "CPU Wait" usage bar. If your application is swapping, try to reduce its memory usage as follows:

- Turn off render caching. Call **SoSeparator::setNumRenderCaches(0)** just after initializing Inventor to globally turn off automatic render caching. You can also turn off render caching for parts of your scene using the **renderCaching** field of **SoSeparator**.

  The automatic caching algorithm in Inventor 2.1 avoids caching notes that contain a large number of polygons.

- If you are using caching, avoid using PER_FACE or PER_FACE_INDEXED materials or normal bindings for **SoTriangleStripSet**, **SoIndexedTriangleStripSet**, and **SoQuadMesh** nodes. FACE bindings force Inventor to break each triangle or quad into an individual triangle or quad, more than doubling the space the node takes in the render cache.

- If you have **SoBaseColor** or **SoMaterial** nodes containing just diffuse colors, change them to **SoPackedColor** nodes, which use less memory.

- Use instancing instead of duplicating geometry or properties wherever possible. Instancing makes your scene graph take up less memory and enables Inventor to build OpenGL display lists that are used more than once. This is especially important for **SoTexture2** nodes.

Looking at CPU Usage

If memory is not the problem, start by looking at "inclusive" CPU times for your procedures (inclusive times include time spent in that procedure and all procedures it calls; exclusive times are just the time spent in that procedure). Ignore the very highest level routines like **main()** or **SoXt::mainLoop()**; look for Inventor **beginTraversal()** methods or for application routines that are taking a significant percentage of time. If a lot of time is spent in **SoGLRenderAction::beginTraversal()**, see "Optimizing Rendering" on page 52 for information on improving rendering performance.

If your application is spending a lot of time in code written by you, you are on your own! The rest of this section describes Inventor routines that often show up on profile traces, describes what these routines do, and suggests ways of using them more efficiently.

Optimizing Action Construction and Setup

Inventor actions perform a lot of work the first time they are applied to a scene (subsequent traversals are very fast). Therefore, if your performance traces show a lot of time being spent inside an action's constructor or the **SoAction::setUpState()** method, try to create an action once and reapply it instead of constructing a new action. For example, if you often compute the bounding boxes of some objects in the scene, keep an instance of an **SoBoundingBoxAction** around that is reused:

```
static SoGetBoundingBoxAction *bbAction = NULL;
if (bbAction == NULL) bbAction = new SoGetBoundingBoxAction;
bbAction->apply(myScene);
```

instead of the much less efficient:

```
SoGetBoundingBoxAction bbAction;
                                // inefficient if called a lot!
bbAction.apply(myScene);
```

Decreasing Notification Overhead

Every time you change a field in the scene, Inventor performs a process called notification. A notification message travels up the scene graph to the node's parents, scheduling sensors, causing caches to be destroyed, and marking any connections to engines or other fields as needing evaluation.

If your performance traces show a lot of time is being spent in a **startNotify()** method, try the following to decrease notification overhead:

- If you are modifying several values in a multiple-valued field, use the **setValues()** methods or the **startEditing()**/**finishEditing()** methods instead of repeatedly calling the **set1Value()** method.
- Build scenes from the bottom up. Set leaf nodes' fields first, then add them to their parents, then add the parents to their parents, and so on. For example, do this:

```
SoCube *myCube = new SoCube;
c->width = 10.0;
SoCylinder *myCylinder = new SoCylinder;
myCylinder->radius = 4.0;
SoSwitch *mySwitch = new SoSwitch;
mySwitch->whichChild = 0;
mySwitch->addChild(cube);
mySwitch->addChild(cylinder);
SoSeparator *root = new SoSeparator;
root->ref();
root->addChild(mySwitch);
```

instead of the less efficient:

```
SoSeparator *root = new SoSeparator;
root->ref();
SoSwitch *mySwitch = new SoSwitch;
root->addChild(mySwitch);
mySwitch->whichChild = 1;
SoCube *myCube = new SoCube;
mySwitch->addChild(myCube);
myCube->width = 4.0;
SoCylinder *myCylinder = new SoCylinder;
mySwitch->addChild(myCylinder);
myCylinder->radius = 4.0;
```

- Using many path sensors can cause notification to become slow, since an **SoPathSensor** is notified whenever any change happens underneath the head node of the **SoPath** monitored by the **SoPathSensor**.

  Note: **SoPath** itself does not have this problem in Inventor 2.X (but did in Inventor 1.X).

- Notification can be enabled or disabled on a per-node or per-engine basis. Beware that because caching, sensors, and connections rely on notification for proper operation, you must be very careful when using this feature. See the **SoFieldContainer** reference page for information on the **enableNotify()** method.

Picking and Handling Events

If your application profiles show a lot of time is spent inside the methods **SoHandleEventAction::beginTraversal()** or **SoPickAction::beginTraversal()**, try the following to improve picking and/or event handling performance:

- Insert **SoPickStyle::UNPICKABLE** nodes in your scene to turn off picking for objects that should never be picked (for example "dead" background graphics).
- Insert **SoPickStyle::BOUNDING_BOX** nodes in your scene if you do not need detailed picking information. This helps most for complicated objects like **SoText3** or **SoTriangleStripSets** with many triangles.
- If you have objects with a lot (thousands) of polygons in them, break them up into several objects under different separators, grouping polygons that are close to each other. This allows **SoSeparator** pick culling to quickly reject many of the triangles.
- To speed up event handling, try to put active objects that respond to events toward the left and top of the scene graph. An **SoHandleEventAction** ends traversal as soon as a node reports that it has handled the event.
- If you write your own event callback node, or implement a node that responds to events, be sure to use the **grabEvents()** method when appropriate. Because grabbing short-circuits traversal of the scene, it is a useful way to speed up event distribution.

Appendix A

Creating a Node

This appendix explains how you can create new subclasses of **SoNode**. It discusses enabling elements in the state, constructing a node, and implementing actions for it. Chapter 1 in *The Inventor Toolmaker* provides important background material on these concepts, and this appendix assumes you are familiar with the material presented there.

Note: This appendix provides an update of Chapter 2, "Creating a Node," of *The Inventor Toolmaker.* Its main purpose is to help developers who are new to Inventor work with Inventor 2.1 without being hindered by outdated information. Developers familiar with Inventor may prefer to look at Chapter 5, "Incompatible Extender API Changes" and at the revised example programs instead of reading this complete appendix.

The first part of this appendix offers an overview of the steps required to create a new node. When necessary, additional sections explain key concepts in further detail and list the relevant macros. Next, the appendix examples show how to create three new node classes:

- "Creating a Property Node" on page 82
- "Creating a Shape Node" on page 88
- "Creating a Group Node" on page 108

Sections at the end of the chapter discuss the following:

- "Using New Node Classes" on page 116
- "Creating an Abstract Node Class" on page 120
- "The copyContents() Method" on page 120
- "The affectsState() Method" on page 121
- "Uncacheable Nodes" on page 121
- "Generating Default Normals" on page 122
- "Creating an Alternate Representation" on page 122

Overview

The file *SoSubNode.h* contains the macros for defining new node classes. The SO_NODE_HEADER() macro declares type identifier and naming variables and methods that all node classes must support. The SO_NODE_SOURCE() macro defines the static variables and methods declared in the SO_NODE_HEADER() macro. Other macros useful in creating new node classes are mentioned in the following sections.

Creating a new node requires these steps:

1. Select a name for the new node class and determine what class it is derived from.
2. Define and name each field in the node.
3. Define an **initClass()** method to initialize the type information and to ensure that the required elements are enabled in the state (see "Initializing the Node Class" on page 75).
4. Define a constructor (see "Defining the Constructor" on page 76).
5. Implement the actions supported by the node (see "Implementing Actions" on page 78).
   - For a property node, you usually need to implement the **GLRender()** and **callback()** methods (see "Creating a Property Node" on page 82). You may also need to implement **getBoundingBox()**, **getMatrix()**, and other methods.
   - For a shape node, you need to implement the **generatePrimitives()** method for the **SoCallbackAction** as well as the **getBoundingBox()** method. You may want to implement a specific **GLRender()** or **rayPick()** method as well (see "Creating a Shape Node" on page 88). For vertex-based shapes, you may need to implement a **generateDefaultNormals()** method.
   - For a group node, you need to implement all actions to ensure that the children are traversed (see "Creating a Group Node" on page 108).
6. Implement a **copyContents()** method if the node contains any non-field instance data (see "The copyContents() Method" on page 120).
7. Implement an **affectsState()** method if it cannot be inherited from the parent class (see "The affectsState() Method" on page 121).

Initializing the Node Class

As discussed in Chapter 1 of *The Inventor Toolmaker*, the **initClass()** method sets up the type identifier and file format name information for the class. The initialization macro for nodes, SO_NODE_INIT_CLASS(), does most of the work for you. One additional task for you as the node writer is to enable each of the elements in the state for each action the node supports. The following subsections provide additional information about enabling elements in the state.

Enabling Elements in the State

In the **initClass()** method, use the SO_ENABLE() macro (defined in *SoAction.h*) to enable the elements required by your node in the state. To use a simple example, **SoDrawStyle** enables these elements in the **SoGLRenderAction**:

```
SO_ENABLE(SoGLRenderAction, SoGLDrawStyleElement);
SO_ENABLE(SoGLRenderAction, SoGLLinePatternElement);
SO_ENABLE(SoGLRenderAction, SoGLLineWidthElement);
SO_ENABLE(SoGLRenderAction, SoGLPointSizeElement);
```

SoDrawStyle also implements the **SoCallbackAction**. It enables these elements in the **SoCallbackAction**:

```
SO_ENABLE(SoCallbackAction, SoDrawStyleElement);
SO_ENABLE(SoCallbackAction, SoLinePatternElement);
SO_ENABLE(SoCallbackAction, SoLineWidthElement);
SO_ENABLE(SoCallbackAction, SoPointSizeElement);
```

Tip: If you know that the element is already enabled by another node or action, you can skip this step.

Now that these elements have been enabled, their values can be set and queried. (The debugging version of Inventor generates an error if you try to access or set an element that has not been enabled.)

Inheritance Within the Element Stack

The example using **SoDrawStyle** elements in the previous section brings up another feature of the element stack: Some elements have corresponding GL versions that are derived from them. The **SoGL** version of an element typically sends its value to OpenGL when it is set. **SoGLDrawStyleElement** is derived from **SoDrawStyleElement**, and **SoGLLinePatternElement** is derived from **SoLinePatternElement**. The parent element class and its derived class *share* the same stack index.

If you try to enable two classes that share a stack index (for example, **SoGLDrawStyleElement** and **SoDrawStyleElement**), only the more derived class is actually enabled (in this case, **SoGLDrawStyleElement**). However, you can always use the base class static method to set or get the value for either the parent or the derived class. (You cannot, however, enable only the parent version and then try to treat it as the derived GL version.)

Defining the Constructor

The constructor defines the fields for the node and sets up their default values. If the fields contain enumerated values, their names and values are defined in the constructor as well. Use the SO_NODE_CONSTRUCTOR() macro to perform the basic work.

The SO_NODE_IS_FIRST_INSTANCE() macro returns a Boolean value that can be tested in constructors. If your class requires considerable overhead when it is initialized, you may want to perform this work only once when the first instance of the class is created. For example, the **SoCube** class sets up the coordinates and normals of the cube faces during construction of its first instance. (You could put this code in the **initClass()** method, but putting it in the constructor guarantees that someone is actually using your node class first.)

Setting Up the Node's Fields

The SO_NODE_ADD_FIELD() macro defines the fields in the node and sets up their default values. The first parameter is the name of the field. The second parameter is the default field value, in parentheses. Using **SoDrawStyle** as an example:

```
SO_NODE_ADD_FIELD(style, (SoDrawStyleElement::getDefault()));
SO_NODE_ADD_FIELD(lineWidth,
                  (SoLineWidthElement::getDefault()));
SO_NODE_ADD_FIELD(linePattern,
                  (SoLinePatternElement::getDefault()));
SO_NODE_ADD_FIELD(pointSize,
                  (SoPointSizeElement::getDefault()));
```

To add a field with a vector value, the syntax is as follows:

```
SO_NODE_ADD_FIELD(translation, (0.0, 0.0, 0.0));
```

Defining Enumerated Values for a Field

In the preceding example, the **style** field contains an enumerated value: FILLED, LINES, POINTS, or INVISIBLE. Use the SO_NODE_DEFINE_-ENUM_VALUE() macro to define the enumerated values. The first parameter is the type of the enumerated value. The second parameter is its value, as shown here:

```
SO_NODE_DEFINE_ENUM_VALUE(Style, FILLED);
SO_NODE_DEFINE_ENUM_VALUE(Style, LINES);
SO_NODE_DEFINE_ENUM_VALUE(Style, POINTS);
SO_NODE_DEFINE_ENUM_VALUE(Style, INVISIBLE);
```

Then, to specify that these enumerated values can be used in the **style** field of the **SoDrawStyle** node, use the SO_NODE_SET_SF_ENUM_TYPE() macro:

```
O_NODE_SET_SF_ENUM_TYPE(style, Style);
```

Implementing Actions

Your next task is to implement each of the actions your new node supports. The **SoDrawStyle** node, as you have already seen, supports two actions, the **SoGLRenderAction** and the **SoCallbackAction**, in addition to the **SoSearchAction** and the **SoWriteAction**, which it inherits from **SoNode**.

Tip: Do not apply a new action within another action (because caching will not function properly). Also, if you are creating a new node, do not modify the node (for example, call **setValue()** on a field) within an action method.

The doAction() Method

For the GL render action, the **SoDrawStyle** node changes the values of 4 elements in the state based on the value of the corresponding fields. For example, if its **style** field has a value of INVISIBLE, it changes the value of the **SoGLDrawStyleElement** in the state to INVISIBLE. First it must check if another node has overridden the draw style. The code to set the element's value is:

```
if (! style.isIgnored() &&
    ! SoOverrideElement::getDrawStyleOverride(state)) {
    if (isOverride()) {
        SoOverrideElement::setDrawStyleOverride(state, this,
                                                TRUE);
    }
    SoDrawStyleElement::set(state,
          (SoDrawStyleElement::Style) style.getValue());
}
```

For the callback action, **SoDrawStyle** does the same thing: It sets the value of the element based on the value of the corresponding field in the node. Since the two actions perform exactly the same tasks, the common code is put into a separate method that both the GL render and the callback actions can call. By convention, this shared method used by property nodes is called **doAction()**, which is a virtual method on **SoNode**. The code for the draw-style node's callback action followed by the node's GL render action:

```
void
SoDrawStyle::callback(SoCallbackAction *action)
(
```

```
   doAction(action);
}

void
SoDrawStyle::GLRender(SoGLRenderAction *action)
(
   doAction(action);
}
```

This is the draw-style node's **doAction()** method:

```
SoDrawStyle::doAction(SoAction *action)
{
    SoState     *state = action->getState();

    if (! style.isIgnored() &&
        ! SoOverrideElement::getDrawStyleOverride(state)) {
        if (isOverride()) {
            SoOverrideElement::setDrawStyleOverride(
                                        state, this, TRUE);}
        SoDrawStyleElement::set(state,
                                (SoDrawStyleElement::Style)
                                 style.getValue());
    }

    if (! pointSize.isIgnored() &&
        ! SoOverrideElement::getPointSizeOverride(state)) {
        if (isOverride()) {
            SoOverrideElement::setPointSizeOverride(
                                        state, this, TRUE);}
        SoPointSizeElement::set(state, pointSize.getValue());
    }

    if (! lineWidth.isIgnored() &&
       ! SoOverrideElement::getLineWidthOverride(state)) {
        if (isOverride()) {
            SoOverrideElement::setLineWidthOverride(state,
                                        state, this, TRUE);}
        SoLineWidthElement::set(state, lineWidth.getValue());
    }
```

```
    if (! linePattern.isIgnored() &&
       !SoOverrideElement::getLinePatternOverride(state)) {
        if (isOverride()) {
            SoOverrideElement::setLinePatternOverride(
                                          state, this, TRUE);}
        SoLinePatternElement::set(state,
                                    linePattern.getValue());
    }
}
```

The advantage of this scheme becomes apparent when you consider extending the set of actions (see Chapter 4 of *The Inventor Toolmaker*). You can define a new action class and implement a static method for **SoNode** that calls **doAction()**. Then all properties that implement **doAction()** will perform the appropriate operation without needing any static methods for them.

Changing and Examining State Elements

As discussed in Chapter 1 of *The Inventor Toolmaker*, each element class provides methods for setting and querying its value. The static **set()** method usually has two parameters, as shown in the previous section:

- The state from which to retrieve the element (which the element obtains from the given action)
- The new value for the element

Most element classes also define a static **get()** method that returns the current value stored in an element instance. For example, to obtain the current draw style, use:

```
style = SoDrawStyleElement::get(action->getState());
```

Elements that have multiple values may define a different sequence of **get()** methods. For example, the material color elements and coordinate element can contain many values. In these cases, the element class defines three methods:

getInstance() returns the top instance of the element in the state as a **const** pointer

getNum()	returns the number of values in the element
get(*n*)	returns the *n*th value in the element

Element Bundles

Elements are designed to be small and specific, for two reasons. The first is that it should be possible for a node to change one aspect of the state without having to change any of the rest, including related elements. For example, the **SoMaterialBinding** node changes only the **SoMaterialBindingElement** without affecting any other material elements. The second reason has to do with caching. It is easy to determine when any element's value has changed, since (typically) the whole element changes at once. Therefore, determining which nodes affect a cache is a straightforward task.

However, some elements are related to each other, and it's good to deal with them together for convenience and efficiency. Classes called *bundles* provide simple interfaces to collections of related elements.

Supported Inventor bundle classes are:

- **SoMaterialBundle**
- **SoNormalBundle**
- **SoTextureCoordinateBundle**

The **SoMaterialBundle** class accesses all elements having to do with surface materials. In Inventor 2.1, these functions are more efficiently performed by using the **SoLazyElement** (see the next section for more information). The following examples use the **SoLazyElement** instead of material bundles.

Methods on the **SoMaterialBundle** allow shapes to step easily through sequential materials and to send the current material to OpenGL. **SoNormalBundle** lets you step easily through sequential normals and provides functions for generating default normals. **SoTextureCoordinateBundle** lets you step through texture coordinates and provides methods for generating texture coordinates if the shape is using **SoTextureCoordinatePlane** or **SoTextureCoordinateEnvironment**.

The SoLazyElement

Several elements that were responsible for material properties in Inventor 2.0 have been combined into two elements: **SoLazyElement** and **SoGLLazyElement**, its derived OpenGL version.

- **SoLazyElement** is responsible for setting and getting material state.
- **SoGLLazyElement** is responsible for tracking the OpenGL state, invalidating caches, and so on.

See "Changes in Material Elements" on page 38 for more information.

Creating a Property Node

The easiest way to learn how to add a node class is by example. The first example creates a new property class called **Glow**, which modifies the emissive color of the current material to make objects appear to glow. It has a field called **color**, which is the color of the glow, and a float field called **brightness**, ranging from 0 to 1, indicating how much the object should glow and a **transparency** field, indicating the transparency of the glowing material. Its value is between 0 (opaque) and 1 (invisible).

In Inventor 2.1, the transparency and diffuse color are combined in the state into a packed color. To facilitate dealing with diffuse colors and transparency independently, a class **SoColorPacker** is provided. In the example that follows, the **SoColorPacker** is constructed in the **Glow** node constructor, and is used whenever the transparency is set in the state.

The **doAction()** method of the **Glow** node must check whether the emissive color or transparency is already being overridden by another material node. The **SoOverrideElement** determines what is overridden.

For this class, the program needs to implement actions that deal with materials: **GLRender()** and **callback()**. It uses **doAction()** (see "The doAction() Method" on page 78) since it performs the same operation for both actions. The **doAction()** method for the **Glow** class updates the lazy color element based on the values of the **color**, **transparency**, and **brightness** fields of the node.

The class header for the new node is shown in Example A-1.

Example A-1 Glow.h

```
#include <Inventor/SbColor.h>
#include <Inventor/fields/SoSFColor.h>
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/nodes/SoSubNode.h>

// Need SoColorPacker to put the transparency into the state:
class SoColorPacker;

class Glow : public SoNode {

   SO_NODE_HEADER(Glow);

 public:
   // Fields:
   SoSFColor  color;            // Color of glow
   SoSFFloat  brightness;       // Amount of glow (0-1)
   SoSFFloat  transparency; // transparency of glowing object

   // Initializes this class for use in scene graphs. This
   // should be called after database initialization and
   // before any instance of this node is constructed.
   static void    initClass();

   // Constructor
   Glow();

 protected:
   // These implement supported actions. The only actions
   // dealing with materials are the callback and GL render
   // actions. We will inherit all other action methods from
   // SoNode.
   virtual void   GLRender(SoGLRenderAction *action);
   virtual void   callback(SoCallbackAction *action);

   // This implements generic traversal of Glow node, used in
   // both of the above methods.
   virtual void   doAction(SoAction *action);

 private:
   // Destructor. Private to keep people from trying to
   // delete nodes instead of using reference count
```

```
   // mechanism.
   virtual ~Glow();

   // A pointer to the color packer is required in nodes
   // that set diffuse or transparency.  This will
   // be initialized in constructor, deleted in destructor:
   SoColorPacker *colorPacker;

   // Keep a copy of the transparency that this node puts
   // into the state:
   float transpValue;
```

The **Glow** node is representative of most property nodes in that it is concerned solely with editing the current traversal state, regardless of the action being performed.

The source code for the **Glow** class is shown in Example A-2.

Example A-2 Glow.c++

```
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/elements/SoLazyElement.h>
#include <Inventor/elements/SoOverrideElement.h>

#include "Glow.h"

SO_NODE_SOURCE(Glow);

//
// Initializes the Glow class. This is a one-time thing that
// is done after database initialization and before any
// instance of this class is constructed.
//

void
Glow::initClass()
{
   // Initialize type id variables. Arguments to the macro
   // are: the name of the node class, the class this is
   // derived from, and the name registered with the type of
   // the parent class.
   SO_NODE_INIT_CLASS(Glow, SoNode, "Node");
}
```

```
//
// Constructor
//

Glow::Glow()
{
   // Do standard constructor stuff:
   SO_NODE_CONSTRUCTOR(Glow);

   // Add "color" field to the field data. The default value
   // for this field is R=G=B=1, which is white.
   SO_NODE_ADD_FIELD(color, (1.0, 1.0, 1.0));

   // Add "brightness" field to the field data. The default
   // value for this field is 0.
   SO_NODE_ADD_FIELD(brightness, (0.0));

   // Add "transparency" field to the field data.  Default
   // value is 0 (opaque).
   SO_NODE_ADD_FIELD(transparency,  (0.0));

   // Initialize the color Packer (required of any property
   // node that uses an SoColorPacker to set diffuse color
   // or transparency):
   colorPacker = new SoColorPacker;

}

//
// Destructor
//

Glow::~Glow()
{
    //  Delete the color Packer:
    delete colorPacker;
}

//
// Implements GL render action.
//

void
Glow::GLRender(SoGLRenderAction *action)
```

```
{
   // Set the elements in the state correctly. Note that we
   // prefix the call to doAction() with the class name. This
   // avoids problems if someone derives a new class from the
   // Glow node and inherits the GLRender() method; Glow's
   // doAction() will still be called in that case.
   Glow::doAction(action);

   //  Note that in Inventor 2.1 the GLRender method only
   //  sets the color in the lazy element; it does not send
   //  it to GL.  This is for efficiency, since there is no
   //  need to send it until it is really used.

}

//
// Implements callback action.
//

void
Glow::callback(SoCallbackAction *action)
{
   // Set the elements in the state correctly.
   Glow::doAction(action);
}

//
// Typical action implementation; it sets the correct element
// in the action's traversal state. We assume that the lazy
// element has been enabled.
//

void
Glow::doAction(SoAction *action)
{
   // Make sure the "brightness" field is not ignored. If it
   // is, then we don't need to change anything in the state.

   // Also check that the emissive color is not being
   // overridden; if it is, this node should not set it.

   if (! brightness.isIgnored() &&
        ! SoOverrideElement::getEmissiveColorOverride
                                        (action->getState())) {
```

```
        SbColor emissiveColor = color.getValue() *
                                        brightness.getValue();

        //Use the Lazy element to set emissive color.
        //Note that this won't actually send the color to GL.

        SoLazyElement::setEmissive(action->getState(),
                                              &emissiveColor);

    }

    // To send transparency we again check ignore flag and
    // override element.

    if (! transparency.isIgnored() &&
         ! SoOverrideElement::getTransparencyOverride
                                       (action->getState())) {

         // Keep a copy of the transparency that we are
         // putting in the state:
         transpValue = transparency.getValue();

         // The color packer must be provided when
         // transparency is set, so that the transparency
         // will be merged with current diffuse color
         // in the state:
         SoLazyElement::setTransparency(action->getState(),
                        this, 1, &transpValue,  colorPacker);
     }
}
```

Creating a Shape Node

This next example is more complicated than the property-node example, because shape nodes need to access more of the state and implement different behaviors for different actions. For example, a shape needs to draw geometry during rendering, return intersection information during picking, and compute its extent when getting a bounding box.

All shapes need to define at least two methods: **generatePrimitives()** and **getBoundingBox()**. When you define the **generatePrimitives()** method for your new class, you can inherit the **GLRender()** and **rayPick()** methods from the base class, **SoShape**, because they use the generated primitives. This feature saves time at the prototyping stage, since you need to implement only the **generatePrimitives()** method, and rendering and picking are provided at no extra cost. When you are ready for fine-tuning, you can redefine these two methods to improve performance.

Generating Primitives

When a shape node is traversed to generate primitives for the **SoCallbackAction**, it generates triangles, line segments, or points. The information for each vertex of the triangle, line segment, or point is stored in an instance of **SoPrimitiveVertex**. The shape fills in the information for each vertex. Then, for each primitive generated (that is, triangle, line segment, or point), an appropriate callback function is invoked by a method on **SoShape**. For example, if the shape generates triangles, the triangle callback function is invoked for every triangle generated. Filled shapes, such as **SoCone** and **SoQuadMesh**, generate triangles (regardless of draw style), line shapes (such as **SoLineSet** and **SoIndexedLineSet**) generate line segments, and point shapes (such as **SoPointSet**) generate points.

SoPrimitiveVertex

The **SoPrimitiveVertex** contains all information for that vertex:

- Point (coordinates, in object space)
- Normal
- Texture coordinates
- Material index
- A pointer to an instance of an **SoDetail** subclass (may be NULL)

The shape's **generatePrimitives()** method sets each of these values.

The appropriate callback function can be invoked either automatically or explicitly. If you want explicit control over when the callback function is invoked, you can use the following methods provided by the **SoShape** class:

- **invokeTriangleCallbacks()**
- **invokeLineSegmentCallbacks()**
- **invokePointCallbacks()**

To take advantage of the automatic mechanism, use these three methods, provided by the **SoShape** base class as a convenience:

- **beginShape**(*action*, *shapeType*)
- **shapeVertex(***&vertex*)
- **endShape()**

The *shapeType* parameter is TRIANGLE_FAN, TRIANGLE_STRIP, TRIANGLES, or POLYGON. For example, if you choose TRIANGLE_FAN, this method performs the necessary triangulation and invokes the appropriate callbacks for each successive triangle of the shape. This mechanism is similar to OpenGL's geometry calls.

Creating Details

You may want your shape to store additional information in an **SoDetail**—for example, what part of the shape each vertex belongs to. In this case, you can use an existing subclass of **SoDetail** (see *The Inventor Mentor,* Chapter 9), or you can create a new **SoDetail** subclass to hold the appropriate information. By default, the pointer to the detail in **SoPrimitiveVertex** is NULL.

If you decide to store information in an **SoDetail**, you create an instance of the subclass and store a pointer to it in the **SoPrimitiveVertex** by calling **setDetail()**.

Rendering

For rendering, you may be able to inherit the **GLRender()** method from the **SoShape** class. In this case, you define a **generatePrimitives()** method as described in the previous sections. Each primitive is generated and then rendered separately.

In other cases, you may want to write your own render method for the new shape class, especially if it would be more efficient to send the vertex information to OpenGL in some other form, such as triangle strips. The Pyramid node created later in this appendix implements its own **GLRender()** method. Before rendering, the shape should test whether it needs to be rendered. You can use the **SoShape::shouldGLRender()** method, which checks for INVISIBLE draw style, BOUNDING_BOX complexity, delayed transparency, and render abort.

Inventor takes care of sending the draw-style value to OpenGL (where it is handled by **glPolygonMode()**). This means that filled shapes are drawn automatically as lines or points if the draw style indicates such. Note that if your object is composed of lines, but the draw style is POINTS, you need to handle that case explicitly. You need to check whether the draw-style element in the state is points or lines and render the shape accordingly.

The lazy elements are used in the **GLRender()** method of the **Pyramid** to send the colors to OpenGL. This has to be set up as follows:

- **SoLazyElement::getLightModel()** is used to tell if lighting is on. If so, it is necessary to send normals to OpenGL.
- **SoLazyGLElement::sendAllMaterial()** is needed prior to rendering the shape. This ensures that the OpenGL material state is in agreement with the current Inventor state.
- **SoGLLazyElement::sendDiffuseByIndex()** is required if more than one diffuse color or transparency is sent. This issues the appropriate OpenGL commands to send the specified color and transparency to OpenGL.
- **SoGLLazyElement::reset()** is required at the end of rendering, if colors other than the first color were sent to OpenGL. This informs the lazy element that the current color in the state is not the last one sent to GL.

Picking

For picking, you may also be able to inherit the **rayPick()** method from the **SoShape** class. In this case, you define a **generatePrimitives()** method, and the parent class **rayPick()** method tests the picking ray against each primitive that has been generated. If the ray intersects the primitive, it creates an **SoPickedPoint**. **SoShape** provides three virtual methods for creating details:

- **createTriangleDetail**()
- **createLineDetail()**
- **createPointDetail()**

The default methods return NULL, but your shape can override this to set up and return a detail instance.

The **Pyramid** node created later in this appendix inherits the **rayPick()** method from **SoShape** in this manner.

For some shapes, such as spheres and cylinders, it is more efficient to check whether the picking ray intersects the object without tessellating the object into primitives. In such cases, you can implement your own **rayPick()** method and use the **SoShape::shouldRayPick()** method, which first checks to see if the object is pickable.

The following excerpt from the **SoSphere** class shows how to implement your own **rayPick()** method:

```
void
SoSphere::rayPick(SoRayPickAction *action)
{
   SbVec3f             enterPoint, exitPoint, normal;
   SbVec4f             texCoord(0.0, 0.0, 0.0, 1.0);
   SoPickedPoint       *pp;

   // First see if the object is pickable.
   if (! shouldRayPick(action))
       return;

   // Compute the picking ray in our current object space.
   computeObjectSpaceRay(action);

   // Create SbSphere with correct radius, centered at zero.
   float       rad = (radius.isIgnored() ? 1.0 :
                      radius.getValue());
   SbSphere    sph(SbVec3f(0., 0., 0.), rad);

   // Intersect with pick ray. If found, set up picked
   // point(s).
   if (sph.intersect(action->getLine(), enterPoint,
                                       exitPoint)) {
      if (action->isBetweenPlanes(enterPoint) &&
          (pp = action->addIntersection(enterPoint)) !=
                                                 NULL) {

         normal = enterPoint;
         normal.normalize();
         pp->setObjectNormal(normal);
         // This macro computes the s and t texture
         // coordinates for the shape.
         COMPUTE_S_T(enterPoint, texCoord[0], texCoord[1]);
         pp->setObjectTextureCoords(texCoord);
      }

      if (action->isBetweenPlanes(exitPoint) &&
          (pp = action->addIntersection(exitPoint)) != NULL){

         normal = exitPoint;
         normal.normalize();
         pp->setObjectNormal(normal);
```

```
            COMPUTE_S_T(exitPoint, texCoord[0], texCoord[1]);
            texCoord[2] = texCoord[3] = 0.0;
            pp->setObjectTextureCoords(texCoord);
        }
    }
}
```

Getting a Bounding Box

SoShape provides a **getBoundingBox()** method that your new shape class can inherit. This method calls a virtual **computeBBox()** method, which you need to define. (The **computeBBox()** method is also used during rendering when bounding-box complexity is specified.)

If you are deriving a class from **SoNonIndexedShape**, you can use the **computeCoordBBox()** method within your **computeBBox()** routine. This method computes the bounding box by looking at the specified number of vertices, starting at **startIndex**. It uses the minimum and maximum coordinate values to form the diagonal for the bounding box and uses the average of the vertices as the center of the object.

If you are deriving a class from **SoIndexedShape**, you can inherit **computeBBox()** from the base **SoIndexedShape** class. This method uses all nonnegative indices in the coordinates list to find the minimum and maximum coordinate values. It uses the average of the coordinate values as the center of the object.

For improved picking performance, the **rayPick()** action assumes that the bounding boxes include only surfaces; not lines or points. If your shape object is composed of lines and/or points, you must implement a **getBoundingBox()** method similar to the following method from the **SoLineSet** implementation:

```
void
SoLineSet::getBoundingBox(SoGetBoundingBoxAction *action)
{
    // Let our parent class do the real work
    SoNonIndexedShape::getBoundingBox(action);
```

```
    // If there are any open bounding box caches, tell them
    // that they contain lines
    SoBoundingBoxCache::setHasLinesOrPoints
                                        (action->getState());
}
```

Pyramid Node

This example creates a **Pyramid** node, which has a square base at $y = -1$ and its apex at (0.0, 1.0, 0.0). The code presented here is similar to that used for other primitive (nonvertex-based) shapes, such as cones and cylinders. The pyramid behaves like an **SoCone**, except that it always has four sides. And, instead of a **bottomRadius** field, the **Pyramid** class has **baseWidth** and **baseDepth** fields in addition to the **parts** and **height** fields.

Some of the work for all shapes can be done by methods on the base shape class, **SoShape**. For example, **SoShape::shouldGLRender()** checks for INVISIBLE draw style when rendering. **SoShape::shouldRayPick()** checks for UNPICKABLE pick style when picking. This means that shape subclasses can concentrate on their specific behaviors.

To define a vertex-based shape subclass, you probably want to derive your class from either **SoNonIndexedShape** or **SoIndexedShape**. These classes define some methods and macros that can make your job easier.

You may notice in this example that there are macros (defined in *SoSFEnum.h*) that make it easy to deal with fields containing enumerated types, such as the parts field of our node. Similar macros are found in *SoMFEnum.h* and in the header files for the **bitmask** fields.

The class header for the **Pyramid** node is shown in Example A-3.

Example A-3 Pyramid.h

```
#include <Inventor/SbLinear.h>
#include <Inventor/fields/SoSFBitMask.h>
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/nodes/SoShape.h>
// SoShape.h includes SoSubNode.h; no need to include it here

// Pyramid texture coordinates are defined on the sides so
```

```
// that the seam is along the left rear edge, wrapping
// counterclockwise around the sides. The texture
// coordinates on the base are set up so the texture is
// right side up when the pyramid is tilted back.

class Pyramid : public SoShape {

   SO_NODE_HEADER(Pyramid);

 public:

   enum Part {                  // Pyramid parts:
      SIDES = 0x01,             // The 4 side faces
      BASE  = 0x02,             // The bottom square face
      ALL   = 0x03,             // All parts
   };

   // Fields
   SoSFBitMask   parts;         // Visible parts
   SoSFFloat     baseWidth;     // Width of base
   SoSFFloat     baseDepth;     // Depth of base
   SoSFFloat     height;        // Height, base to apex

   // Initializes this class
   static void   initClass();

   // Constructor
   Pyramid();

   // Turns on/off a part of the pyramid. (Convenience)
   void          addPart(Part part);
   void          removePart(Part part);

   // Returns whether a part is on or off. (Convenience)
   SbBool        hasPart(Part part) const;

 protected:
   // This implements the GL rendering action. We inherit
   // all other action behavior, including rayPick(), which
   // is defined by SoShape to pick against all of the
   // triangles created by generatePrimitives.
   virtual void  GLRender(SoGLRenderAction *action);

   // Generates triangles representing a pyramid
   virtual void  generatePrimitives(SoAction *action);
```

```
   // This computes the bounding box and center of a pyramid.
   // It is used by SoShape for the SoGetBoundingBoxAction;
   // also to compute the correct box to render or pick when
   // complexity is BOUNDING_BOX. Note that we do not have to
   // define a getBoundingBox() method, since SoShape already
   // takes care of that (using this method).
   virtual void   computeBBox(SoAction *action,
                              SbBox3f &box, SbVec3f &center);
 private:
   // Face normals. These are static because they are
   // computed once and are shared by all instances
   static SbVec3f frontNormal, rearNormal;
   static SbVec3f leftNormal,  rightNormal;
   static SbVec3f baseNormal;

   // Destructor
   virtual ~Pyramid();

   // Computes and returns half-width, half-height, and
   // half-depth based on current field values
   void           getSize(float &halfWidth,
                          float &halfHeight,
                          float &halfDepth) const;
};
```

The source code for the **Pyramid** node is shown in Example A-4.

Example A-4 Pyramid.c++

```
/*-------------------------------------------------------------
 *  This is an example from the Inventor Toolmaker,
 *  chapter 2, example 4.
 *
 *  Source file for "Pyramid" shape node.
*-------------------------------------------------------------*/

#include <GL/gl.h>
#include <Inventor/SbBox.h>
#include <Inventor/SoPickedPoint.h>
#include <Inventor/SoPrimitiveVertex.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/actions/SoRayPickAction.h>
#include <Inventor/elements/SoGLLazyElement.h>
#include <Inventor/elements/SoGLTextureCoordinateElement.h>
#include <Inventor/elements/SoGLTextureEnabledElement.h>
```

```
#include <Inventor/elements/SoMaterialBindingElement.h>
#include <Inventor/elements/SoModelMatrixElement.h>
#include <Inventor/misc/SoState.h>
#include "Pyramid.h"

// Shorthand macro for testing whether the current parts
// field value (parts) includes a given part (part)
#define HAS_PART(parts, part) (((parts) & (part)) != 0)

SO_NODE_SOURCE(Pyramid);

// Normals to four side faces and to base
SbVec3f Pyramid::frontNormal, Pyramid::rearNormal;
SbVec3f Pyramid::leftNormal,  Pyramid::rightNormal;
SbVec3f Pyramid::baseNormal;

//
// This initializes the Pyramid class.
//

void
Pyramid::initClass()
{
   // Initialize type id variables
   SO_NODE_INIT_CLASS(Pyramid, SoShape, "Shape");
}

//
// Constructor
//

Pyramid::Pyramid()
{
   SO_NODE_CONSTRUCTOR(Pyramid);
   SO_NODE_ADD_FIELD(parts,     (ALL));
   SO_NODE_ADD_FIELD(baseWidth, (2.0));
   SO_NODE_ADD_FIELD(baseDepth, (2.0));
   SO_NODE_ADD_FIELD(height,    (2.0));

   // Set up static values and strings for the "parts"
   // enumerated type field. This allows the SoSFEnum class
   // to read values for this field. For example, the first
   // line below says that the first value (index 0) has the
   // value SIDES (defined in the header file) and is
   // represented in the file format by the string "SIDES".
```

```
   SO_NODE_DEFINE_ENUM_VALUE(Part, SIDES);
   SO_NODE_DEFINE_ENUM_VALUE(Part, BASE);
   SO_NODE_DEFINE_ENUM_VALUE(Part, ALL);

   // Copy static information for "parts" enumerated type
   // field into this instance.
   SO_NODE_SET_SF_ENUM_TYPE(parts, Part);

   // If this is the first time the constructor is called,
   // set up the static normals.
   if (SO_NODE_IS_FIRST_INSTANCE()) {
      float invRoot5      = 1.0 / sqrt(5.0);
      float invRoot5Twice = 2.0 * invRoot5;

      frontNormal.setValue(0.0, invRoot5,  invRoot5Twice);
      rearNormal.setValue( 0.0, invRoot5, -invRoot5Twice);
      leftNormal.setValue( -invRoot5Twice, invRoot5, 0.0);
      rightNormal.setValue( invRoot5Twice, invRoot5, 0.0);
      baseNormal.setValue(0.0, -1.0, 0.0);
   }
}

//
// Destructor
//

Pyramid::~Pyramid()
{
}

//
// Turns on a part of the pyramid. (Convenience function.)
//

void
Pyramid::addPart(Part part)
{
   parts.setValue(parts.getValue() | part);
}

//
// Turns off a part of the pyramid. (Convenience function.)
//

void
```

```
Pyramid::removePart(Part part)
{
   parts.setValue(parts.getValue() & ~part);
}

//
// Returns whether a given part is on or off. (Convenience
// function.)
//

SbBool
Pyramid::hasPart(Part part) const
{
   return HAS_PART(parts.getValue(), part);
}

//
// Implements the SoGLRenderAction for the Pyramid node.
//

void
Pyramid::GLRender(SoGLRenderAction *action)
{

   // Access the state from the action.
   SoState  *state = action->getState();

   // See which parts are enabled.
   int curParts = (parts.isIgnored() ? ALL :
                                          parts.getValue());

   // First see if the object is visible and should be
   // rendered now. This is a method on SoShape that checks
   // for INVISIBLE draw style, BOUNDING_BOX complexity, and
   // delayed transparency.
   if (! shouldGLRender(action))
      return;

   // Declare a pointer to a GLLazyElement.  This will be
   // used if we send multiple colors.
   SoGLLazyElement* lazyElt;

   // In Inventor 2.1 we do not use beginSolidShape and
   // endSolidShape.  Instead, this information should be
   // provided in shape hints.
```

```
// Change the current GL matrix to draw the pyramid with
// the correct size. This is easier than modifying all of
// the coordinates and normals of the pyramid. (For extra
// efficiency, you can check if the field values are all
// set to default values - if so, then you can skip this
// step.) Scale world if necessary.
float         halfWidth, halfHeight, halfDepth;
getSize(halfWidth, halfHeight, halfDepth);
glPushMatrix();
glScalef(halfWidth, halfHeight, halfDepth);

// See if texturing is enabled. If so, we will have to
// send explicit texture coordinates. The "doTextures"
// flag will indicate if we care about textures at all.

// Note this has changed slightly in Inventor version 2.1.
// The texture coordinate type now is either FUNCTION or
// DEFAULT. Texture coordinates are needed only for
// DEFAULT textures.

SbBool doTextures =
   (SoGLTextureEnabledElement::get(state) &&
    SoTextureCoordinateElement::getType(state) !=
    SoTextureCoordinateElement::FUNCTION);

// Determine if we need to send normals. Normals are
// necessary if we are not doing BASE_COLOR lighting.

// Use the lazy element to get the light model.
SbBool sendNormals = (SoLazyElement::getLightModel(state)
         != SoLazyElement::BASE_COLOR);

// Determine if there's a material bound per part.
SoMaterialBindingElement::Binding binding =
   SoMaterialBindingElement::get(state);
SbBool materialPerPart =
   (binding == SoMaterialBindingElement::PER_PART ||
    binding ==
              SoMaterialBindingElement::PER_PART_INDEXED);

// Issue a lazy element send.
// The send ensures all material state in GL is current.

SoGLLazyElement::sendAllMaterial(state);
```

```
   // Render the parts of the pyramid. We don't have to worry
   // about whether to render filled regions, lines, or
   // points, since that is already taken care of. We are
   // also ignoring complexity, which we could use to render
   // a more finely tesselated version of the pyramid.

   // We'll use this macro to make the code easier. It uses
   // the "point" variable to store the vertex point to send.
   SbVec3f  point;

#define SEND_VERTEX(x, y, z, s, t)   \
      point.setValue(x, y, z);       \
      if (doTextures)                \
         glTexCoord2f(s, t);         \
      glVertex3fv(point.getValue())

   if (HAS_PART(curParts, SIDES)) {

      // Draw each side separately, so that normals are
      // correct. If sendNormals is TRUE, send face normals
      // with the polygons. Make sure the vertex order obeys
      // the right-hand rule.

      glBegin(GL_TRIANGLES);

      // Front face: left front, right front, apex
      if (sendNormals)
         glNormal3fv(frontNormal.getValue());
      SEND_VERTEX(-1.0, -1.0,  1.0, .25,  0.0);
      SEND_VERTEX( 1.0, -1.0,  1.0, .50,  0.0);
      SEND_VERTEX( 0.0,  1.0,  0.0, .325, 1.0);

      // Right face: right front, right rear, apex
      if (sendNormals)
         glNormal3fv(rightNormal.getValue());
      SEND_VERTEX( 1.0, -1.0,  1.0, .50,  0.0);
      SEND_VERTEX( 1.0, -1.0, -1.0, .75,  0.0);
      SEND_VERTEX( 0.0,  1.0,  0.0, .625, 1.0);

      // Rear face: right rear, left rear, apex
      if (sendNormals)
         glNormal3fv(rearNormal.getValue());
      SEND_VERTEX( 1.0, -1.0, -1.0, .75,  0.0);
      SEND_VERTEX(-1.0, -1.0, -1.0, 1.0,  0.0);
      SEND_VERTEX( 0.0,  1.0,  0.0, .875, 1.0);
```

```
   // Left face: left rear, left front, apex
   if (sendNormals)
      glNormal3fv(leftNormal.getValue());
   SEND_VERTEX(-1.0, -1.0, -1.0, 0.0,  0.0);
   SEND_VERTEX(-1.0, -1.0,  1.0, .25,  0.0);
   SEND_VERTEX( 0.0,  1.0,  0.0, .125, 1.0);

   glEnd();
}

if (HAS_PART(curParts, BASE)) {

   // Send the next material if it varies per part.
   // Use SoGLLazyElement::sendDiffuseByIndex().
   // This also sends transparency, so that if
   // transparency type is not SCREEN_DOOR, there can be
   // a change of transparency across the shape.

   if (materialPerPart){
       //Obtain a current copy of the SoGLLazyElement;
       //use this for the send.
       lazyElt =
      (SoGLLazyElement*)SoLazyElement::getInstance(state);
       lazyElt->sendDiffuseByIndex(1);
   }

   if (sendNormals)
      glNormal3fv(baseNormal.getValue());

   // Base: left rear, right rear, right front, left front
   glBegin(GL_QUADS);
   SEND_VERTEX(-1.0, -1.0, -1.0, 0.0,  0.0);
   SEND_VERTEX( 1.0, -1.0, -1.0, 1.0,  0.0);
   SEND_VERTEX( 1.0, -1.0,  1.0, 1.0,  1.0);
   SEND_VERTEX(-1.0, -1.0,  1.0, 0.0,  1.0);
   glEnd();
}

// Restore the GL matrix.
glPopMatrix();

// Reset the diffuse color, if we sent it twice.
if (materialPerPart)
   lazyElt->reset(state, SoLazyElement::DIFFUSE_MASK);
```

```
}

//
// Generates triangles representing a pyramid.
//

void
Pyramid::generatePrimitives(SoAction *action)
{
   // The pyramid will generate 6 triangles: 1 for each side
   // and 2 for the base. (Again, ignore complexity.)
   // This variable is used to store each vertex.
   SoPrimitiveVertex   pv;

   // Access the state from the action.
   SoState  *state = action->getState();

   // See which parts are enabled.
   int curParts = (parts.isIgnored() ? ALL :
                                       parts.getValue());

   // We need the size to adjust the coordinates.
   float halfWidth, halfHeight, halfDepth;
   getSize(halfWidth, halfHeight, halfDepth);

   // See if we have to use a texture coordinate function,
   // rather than generating explicit texture coordinates.
   SbBool useTexFunc =
      (SoTextureCoordinateElement::getType(state) ==
       SoTextureCoordinateElement::FUNCTION);

   // If we need to generate texture coordinates with a
   // function, we'll need an SoGLTextureCoordinateElement.
   // Otherwise, we'll set up the coordinates directly.
   const SoTextureCoordinateElement *tce;
   SbVec4f texCoord;
   if (useTexFunc)
      tce = SoTextureCoordinateElement::getInstance(state);
   else {
      texCoord[2] = 0.0;
      texCoord[3] = 1.0;
   }

   // Determine if there's a material bound per part.
   SoMaterialBindingElement::Binding binding =
```

```
      SoMaterialBindingElement::get(state);
   SbBool materialPerPart =
      (binding == SoMaterialBindingElement::PER_PART ||
       binding ==
                 SoMaterialBindingElement::PER_PART_INDEXED);

   // We use this macro to make the code easier. It uses the
   // "point" variable to store the primitive vertex's point.
   SbVec3f  point;

#define GEN_VERTEX(pv, x, y, z, s, t, normal)   \
      point.setValue(halfWidth  * x,            \
                     halfHeight * y,            \
                     halfDepth  * z);           \
      if (useTexFunc)                           \
         texCoord = tce->get(point, normal);    \
      else {                                    \
         texCoord[0] = s;                       \
         texCoord[1] = t;                       \
      }                                         \
      pv.setPoint(point);                       \
      pv.setNormal(normal);                     \
      pv.setTextureCoords(texCoord);            \
      shapeVertex(&pv)

   if (HAS_PART(curParts, SIDES)) {

      // We will generate 4 triangles for the sides of the
      // pyramid. We can use the beginShape(), shapeVertex(),
      // andendShape() convenience functions on SoShape to
      // make the triangle generation easier and clearer.
      // The shapeVertex() call is built into the macro.

      // Note that there is no detail information for the
      // Pyramid. If there was, we would create an instance
      // of the correct subclass of SoDetail (such as
      // PyramidDetail) and call pv.setDetail(&detail) once.

      beginShape(action, TRIANGLES);

      // Front face: left front, right front, apex
      GEN_VERTEX(pv, -1.0, -1.0,  1.0, .25,  0.0,
                                       frontNormal);
      GEN_VERTEX(pv,  1.0, -1.0,  1.0, .50,  0.0,
                                       frontNormal);
```

```
    GEN_VERTEX(pv,  0.0,  1.0,  0.0, .325, 1.0,
                                     frontNormal);

    // Right face: right front, right rear, apex
    GEN_VERTEX(pv,  1.0, -1.0,  1.0, .50,  0.0,
                                     rightNormal);
    GEN_VERTEX(pv,  1.0, -1.0, -1.0, .75,  0.0,
                                     rightNormal);
    GEN_VERTEX(pv,  0.0,  1.0,  0.0, .625, 1.0,
                                     rightNormal);

    // Rear face: right rear, left rear, apex
    GEN_VERTEX(pv,  1.0, -1.0, -1.0, .75,  0.0,
                                     rearNormal);
    GEN_VERTEX(pv, -1.0, -1.0, -1.0, 1.0,  0.0,
                                     rearNormal);
    GEN_VERTEX(pv,  0.0,  1.0,  0.0, .875, 1.0,
                                     rearNormal);

    // Left face: left rear, left front, apex
    GEN_VERTEX(pv, -1.0, -1.0, -1.0, 0.0,  0.0,
                                      leftNormal);
    GEN_VERTEX(pv, -1.0, -1.0,  1.0, .25,  0.0,
                                      leftNormal);
    GEN_VERTEX(pv,  0.0,  1.0,  0.0, .125, 1.0,
                                      leftNormal);

    endShape();
}

if (HAS_PART(curParts, BASE)) {

    // Increment the material index in the vertex if
    // necessary. (The index is set to 0 by default.)
    if (materialPerPart)
       pv.setMaterialIndex(1);

    // We will generate two triangles for the base, as a
    // triangle strip.
    beginShape(action, TRIANGLE_STRIP);
```

```
      // Base: left front, left rear, right front, right rear
      GEN_VERTEX(pv, -1.0, -1.0,  1.0, 0.0,  1.0,
                                        baseNormal);
      GEN_VERTEX(pv, -1.0, -1.0, -1.0, 0.0,  0.0,
                                        baseNormal);
      GEN_VERTEX(pv,  1.0, -1.0,  1.0, 1.0,  1.0,
                                         baseNormal);
      GEN_VERTEX(pv,  1.0, -1.0, -1.0, 1.0,  0.0,
                                         baseNormal);

      endShape();
   }
}

//
// Computes the bounding box and center of a pyramid.
//

void
Pyramid::computeBBox(SoAction *, SbBox3f &box, SbVec3f
                                                  &center)
{
   // Figure out what parts are active.
   int curParts = (parts.isIgnored() ? ALL :
                                          parts.getValue());

   // If no part is active, set the bounding box to be tiny.
   if (curParts == 0)
      box.setBounds(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

   else {
      // These points define min and max extents of the box.
      SbVec3f min, max;

      // Compute the half-width, half-height, and half-depth
      // of the pyramid. We'll use this info to set the min
      // and max points.
      float   halfWidth, halfHeight, halfDepth;
      getSize(halfWidth, halfHeight, halfDepth);

      min.setValue(-halfWidth, -halfHeight, -halfDepth);
```

```
      // The maximum point depends on whether the SIDES are
      // active. If not, only the base is present.
      if (HAS_PART(curParts, SIDES))
         max.setValue(halfWidth, halfHeight, halfDepth);
      else
         max.setValue(halfWidth, -halfHeight, halfDepth);

      // Set the box to bound the two extreme points
      box.setBounds(min, max);
   }

   // This defines the "natural center" of the pyramid. We
   // could define it to be the center of the base, if we
   // want, but let's just make it the center of the
   // bounding box.
   center.setValue(0.0, 0.0, 0.0);
}

//
// Computes and returns half-width, half-height, and
// half-depth based on current field values.
//

void
Pyramid::getSize(float &halfWidth,
                 float &halfHeight,
                 float &halfDepth) const
{
   halfWidth  = (baseWidth.isIgnored() ? 1.0 :
                 baseWidth.getValue()  / 2.0);
   halfHeight = (height.isIgnored()    ? 1.0 :
                 height.getValue()     / 2.0);
   halfDepth  = (baseDepth.isIgnored() ? 1.0 :
                 baseDepth.getValue()  / 2.0);
}
```

Tip: The easiest way to make sure your **generatePrimitives()** method is working is to use it for rendering, by temporarily commenting out your shape's **GLRender()** method (if it has one).

Creating a Group Node

The example discussed in this section illustrates how to create a group node subclass. (It is unlikely, however, that you'll need to create a new group class.) Our example class, **Alternate**, traverses every other child (that is, child 0, then child 2, and so on). Since, like the base **SoGroup** class, it has no fields, this example also illustrates how to create a node with no fields.

If you do create a new group class, you probably need to define a new traversal behavior for it. You may be able to inherit some of the traversal behavior from the parent class. Most groups define a protected **traverseChildren()** method that implements their "typical" traversal behavior. Your new group can probably inherit the **read()** and **write()** methods from **SoGroup**.

Child List

SoGroup, and all classes derived from it, store their children in an instance of **SoChildList**. This extender class provides useful methods for group classes, including the **traverse()** method, which has three forms:

traverse(*action*) traverses all children in the child list

traverse(*action, firstChild, lastChild*)
traverses the children from first child to last child, inclusive

traverse(*action, childIndex*)
traverses one child with the specified index

Hidden Children

If you want your new node to have children, but you don't want to grant public access to the child list, you can implement the node to have *hidden children*. Node kits are an example of groups within the Inventor library that have hidden children. Because node kits have a specific internal structure, access to the children needs to be restricted. If you want the node to have hidden children, it should not be derived from **SoGroup**, which has public children only.

SoNode provides a virtual **getChildren()** method that returns NULL by default. To implement a new node with hidden children, you need to do the following:

1. Maintain an **SoChildList** for the node. This list can be a hierarchy of nodes.
2. Implement a **getChildren()** method that returns a pointer to the child list. **(SoPath** uses **getChildren()** to maintain paths.)

Using the Path Code

Recall that an action can be applied to a node, a single path, or a path list. Before a group can traverse its children, it needs to know what the action has been applied to. The **getPathCode()** method of **SoAction** returns an enumerated value that indicates whether the action is being applied to a path and, if so, where this group node is in relation to the path or paths. The values returned by **getPathCode()** are as follows:

NO_PATH	The action is not applied to a path (that is, the action is applied to a node).
BELOW_PATH	This node is at or below the last node in the path chain.
OFF_PATH	This node is not on the path chain (the node is to the left of the path; it needs to be traversed if it affects the nodes in the path).
IN_PATH	The node is in the chain of the path (but is not the last node).

For **SoGroup**, if the group's path code is NO_PATH, BELOW_PATH, or OFF_PATH, it traverses all of its children. (Even if a node is OFF_PATH, you need to traverse it because it affects the nodes in the path to its right. Note, though, that if an **SoSeparator** is OFF_PATH, you do not need to traverse it because it does not have any effect on the path.) If a node is IN_PATH, you may not need to traverse all children in the group, since children to the right of the action path do not affect the nodes in the path. In this case, **getPathCode()** returns the indices of the children that need to be traversed. The **traverseChildren()** method for **SoGroup** looks like this:

```
void
SoGroup::traverseChildren(SoAction *action)
{
   int           numIndices;
   const   int   *indices;

   if (action->getPathCode(numIndices, indices)
       == SoAction::IN_PATH)
       children.traverse(action, 0, indices[numIndices - 1]);
           // Traverse all children up to and including the
           // last child to traverse.

   else
      children.traverse(action);   // traverse all children
}
```

The GL render, callback, handle event, pick, and search methods for **SoGroup** all use **traverseChildren()**. The write method for **SoGroup**, which can be inherited by most subclasses, tests each node in the group before writing it out. The get matrix method does not use **traverseChildren()** because it doesn't need to traverse as much. If the path code for a group is NO_PATH or BELOW_PATH, it does not traverse the children. Here is the code for **SoGroup::getMatrix()**:

```
void
SoGroup::getMatrix(SoGetMatrixAction *action)
{
   int         numIndices;
   const int   *indices;

   switch (action->getPathCode(numIndices, indices)) {
      case SoAction::NO_PATH:
      case SoAction::BELOW_PATH:
         break;
      case SoAction::IN_PATH:
         children.traverse(action, 0,
                                   indices[numIndices - 1]);
         break;
      case SoAction::OFF_PATH:
         children.traverse(action);
         break;
      }
}
```

If a node is IN_PATH, the **getMatrix()** method traverses all the children in the group up to and including the last child in the action path (but not the children to the right of the path). If a node is OFF_PATH, the **getMatrix()** method traverses all the children in the group, since they can affect what is in the path.

What Happens If an Action Is Terminated?

Some actions, such as the GL render, handle event, and search actions, can terminate prematurely—for example, when the node to search for has been found. The **SoAction** class has a flag that indicates whether the action has terminated. The **SoChildList** class checks this flag automatically, so this termination is built into the **SoChildList::traverse()** methods, and the group traversal methods do not need to check the flag.

The new **Alternate** class can inherit the read and write methods from **SoGroup**. We just have to define the traversal behavior for the other actions. Most of the other actions can be handled by the **traverseChildren()** method.

If your group subclass is derived from **SoSeparator**, you must write special rendering methods that correspond to the different path nodes. See "The GLRender() Method" on page 34.

Alternate Node

The class header for the **Alternate** node is shown in Example A-5.

Example A-5 Alternate.h

```
#include <Inventor/nodes/SoGroup.h>
// SoGroup.h includes SoSubNode.h; no need to include it.

class Alternate : public SoGroup {

   SO_NODE_HEADER(Alternate);

 public:
   // Initializes this class.
   static void   initClass();
```

```
   // Default constructor
   Alternate();

   // Constructor that takes approximate number of children
   // as a hint
   Alternate::Alternate(int numChildren);

 protected:
   // Generic traversal of children for any action.
   virtual void  doAction(SoAction *action);

   // These implement supported actions.
   virtual void  getBoundingBox(SoGetBoundingBoxAction
                                          *action);
   virtual void  GLRender(SoGLRenderAction *action);
   virtual void  handleEvent(SoHandleEventAction *action);
   virtual void  pick(SoPickAction *action);
   virtual void  getMatrix(SoGetMatrixAction *action);
   virtual void  search(SoSearchAction *action);

 private:
   // Destructor
   virtual ~Alternate();
};
```

The **Alternate** class source code is shown in Example A-6.

Example A-6 Alternate.c++

```
#include <Inventor/misc/SoChildList.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/actions/SoGetBoundingBoxAction.h>
#include <Inventor/actions/SoGetMatrixAction.h>
#include <Inventor/actions/SoHandleEventAction.h>
#include <Inventor/actions/SoPickAction.h>
#include <Inventor/actions/SoSearchAction.h>
#include "Alternate.h"

SO_NODE_SOURCE(Alternate);

// This initializes the Alternate class.

void
Alternate::initClass()
{
```

```
   // Initialize type id variables
   SO_NODE_INIT_CLASS(Alternate, SoGroup, "Group");
}

// Constructor
Alternate::Alternate()
{
   SO_NODE_CONSTRUCTOR(Alternate);
}
// Constructor that takes approximate number of children.

Alternate::Alternate(int numChildren) : SoGroup(numChildren)
{
   SO_NODE_CONSTRUCTOR(Alternate);
}

// Destructor

Alternate::~Alternate()
{
}

// Each of these implements an action by calling the standard
// traversal method. Note that (as in the Glow node source)
// we prefix the call to doAction() with the name of the
// class to avoid problems with derived classes.

void
Alternate::getBoundingBox(SoGetBoundingBoxAction *action)
{
   Alternate::doAction(action);
}

void
Alternate::GLRender(SoGLRenderAction *action)
{
   Alternate::doAction(action);
}

void
Alternate::handleEvent(SoHandleEventAction *action)
{
   Alternate::doAction(action);
}
```

```
void
Alternate::pick(SoPickAction *action)
{
   Alternate::doAction(action);
}

// This implements the traversal for the SoGetMatrixAction,
// which is handled a little differently: it does not
// traverse below the root node or tail of the path it is
// applied to. Therefore, we need to compute the matrix only
// if this group is in the middle of the current path chain
// or is off the path chain (since the only way this could
// be true is if the group is under a group that affects the
// chain).

void
Alternate::getMatrix(SoGetMatrixAction *action)
{
   int         numIndices;
   const int   *indices;

   // Use SoAction::getPathCode() to determine where this
   // group is in relation to the path being applied to (if
   // any).
   switch (action->getPathCode(numIndices, indices)) {

    case SoAction::NO_PATH:
    case SoAction::BELOW_PATH:
     // If there's no path, or we're off the end, do nothing.
     break;

    case SoAction::OFF_PATH:
    case SoAction::IN_PATH:
     // If we are in the path chain or we affect nodes in the
     // path chain, traverse the children.
     Alternate::doAction(action);
     break;
   }
}

// This implements the traversal for the SoSearchAction,
// which is also a little different. The search action is
// set to either traverse all nodes in the graph or just
// those that would be traversed during normal traversal. We
// need to check that flag before traversing our children.
```

```
void
Alternate::search(SoSearchAction *action)
{
   // If the action is searching everything, then traverse
   // all of our children as SoGroup would.
   if (action->isSearchingAll())
     SoGroup::search(action);

   else {
     // First, make sure this node is found if we are
     // searching for Alternate (or group) nodes.
     SoNode::search(action);

     // Traverse the children in our usual way.
     Alternate::doAction(action);
   }
}

// This implements typical action traversal for an Alternate
// node, skipping every other child.

void
Alternate::doAction(SoAction *action)
{
   int         numIndices;
   const int   *indices;

   // This will be set to the index of the last (rightmost)
   // child to traverse.
   int         lastChildIndex;

   // If this node is in a path, see which of our children
   // are in paths, and traverse up to and including the
   // rightmost of these nodes (the last one in the
   // "indices" array).
   if (action->getPathCode(numIndices, indices) ==
      SoAction::IN_PATH)
     lastChildIndex = indices[numIndices - 1];

   // Otherwise, consider all of the children.
   else
     lastChildIndex = getNumChildren() - 1;

   // Now we are ready to traverse the children, skipping
   // every other one. For the SoGetBoundingBoxAction, we
```

```
    // have to do some extra work in between each pair of
    // children - we have to make sure the center points get
    // averaged correctly.
    if (action->isOfType(
          SoGetBoundingBoxAction::getClassTypeId())) {
      SoGetBoundingBoxAction *bba =
        (SoGetBoundingBoxAction *) action;
      SbVec3f  totalCenter(0.0, 0.0, 0.0);
      int      numCenters = 0;

      for (int i = 0; i <= lastChildIndex; i += 2) {
        children->traverse(bba, i);

        // If the traversal set a center point in the action,
        // add it to the total and reset for the next child.
        if (bba->isCenterSet()) {
          totalCenter += bba->getCenter();
          numCenters++;
          bba->resetCenter();
        }
      }
      // Now, set the center to be the average. Since the
      // centers were already transformed, there's no need to
      // transform the average.
      if (numCenters != 0)
        bba->setCenter(totalCenter / numCenters, FALSE);
    }

    // For all other actions, just traverse every other child.
    else
      for (int i = 0; i <= lastChildIndex; i += 2)
        children->traverse(action, i);
}
```

Using New Node Classes

Node classes you have created must be initialized in every application that uses them. Example A-7 shows how this is done, using the **Glow**, **Pyramid**, and **Alternate** node classes defined in the previous examples. The program reads a file (*newNodes.iv,* shown in Example A-8) that has a scene graph containing instances of these nodes. It writes the scene graph to standard output and then opens an examiner viewer to display the graph.

You can see from this example that extender node classes should be initialized after standard classes, which are initialized by **SoDB::init()**. In this program, **SoDB::init()** is called by **SoXt::init()**. Also, base classes must be initialized before any classes derived from them, since the initialization macros for a node class refer to the parent class.

Notice in Example A-8 that the **Pyramid** and **Glow** nodes, because they are not built into the Inventor library, write out their field names and types. (The **Alternate** class has no fields.) See the discussion of the file format for new (unknown) nodes in *The Inventor Mentor*, Chapter 11.

The **isBuiltIn** flag is a protected variable in **SoFieldContainer**, from which **SoNode** is derived. If this flag is FALSE, field types are written out along with the field values. By default, this flag is FALSE, but all Inventor classes set it to TRUE. If you are building a toolkit that uses Inventor and want your new classes to appear the same as Inventor classes, be sure to set this flag to TRUE.

Example A-7 NewNodes.c++

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/actions/SoWriteAction.h>
#include <Inventor/nodes/SoSeparator.h>

// Header files for new node classes
#include "Glow.h"
#include "Pyramid.h"
#include "Alternate.h"

main(int, char **argv)
{
   SoInput      myInput;
   SoSeparator  *root;

   // Initialize Inventor and Xt.
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   // Initialize the new node classes.
   Glow::initClass();
   Pyramid::initClass();
```

```
   Alternate::initClass();

   if (! myInput.openFile("newNodes.iv")) {
      fprintf(stderr, "Can't open \"newNodes.iv\"\n");
      return 1;
   }

   root = SoDB::readAll(&myInput);
   if (root == NULL) {
      printf("File \"newNodes.iv\" contains bad data\n");
      return 2;
   }

   root->ref();

   // Write the graph to stdout.
   SoWriteAction wa;
   wa.apply(root);

   // Render it.
   SoXtExaminerViewer *myViewer =
      new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);

   //The following results in high-quality transparency and
   //will permit PER_PART material binding to vary
   //transparency.  Delete it and the pyramid will be
   //rendered with the same SCREEN_DOOR transparency
   //overall.
   myViewer->setTransparencyType
                  (SoGLRenderAction::SORTED_OBJECT_BLEND);
   myViewer->setTitle("NewNodes");
   myViewer->show();
   myViewer->viewAll();

   SoXt::show(myWindow);
   SoXt::mainLoop();

   return 0;
}
```

Example A-8 newNodes.iv

```
#Inventor V2.0 ascii

#
# Input file for "newNodes" example program
#

Separator {
   MaterialBinding { value PER_PART }
   Material {
      diffuseColor   [.3 .6 .3, .9 .3 .2]
      transparency   [ 0.8 , 0.0]
      shininess      .5
   }

   # Skip every other child.
   Alternate {
      fields         []

      Pyramid {
         fields         []
      }

      Cube {}           # This child is skipped

      Separator {
         MaterialBinding { value OVERALL }
         Glow {
            fields [ SFColor color , SFFloat brightness ,
                                     SFFloat transparency]
            brightness  .6
            color       .8 .3 .3
            transparency .9
         }
         Transform {
            translation 3 .6 0
         }
         Pyramid {
            fields      [SFFloat   height ]
            height      3.2
         }
      }
```

```
      Sphere {}         # This child is skipped.
   }
}
```

Creating an Abstract Node Class

Creating an abstract node class is slightly different from creating a nonabstract one. Examples of abstract node classes are **SoCamera**, **SoLight**, and **SoShape**.

First, abstract classes should use the ABSTRACT versions of the macros described in *SoSubNode.h*. For example, the **SoLight** class makes this call in its **initClass()** method:

```
SO_NODE_INIT_ABSTRACT_CLASS(SoLight, "Light", SoNode);
```

Second, the constructor for an abstract class should be protected, meaning that it is impossible to create an instance of it.

The copyContents() Method

The **copy()** method defined for **SoNode** creates a copy of an instance of a node. It invokes the virtual **copyContents()** method. If your node has no data other than fields and public children, then the **copyContents()** methods defined for **SoNode** and **SoGroup** should suffice.

However, if you have extra instance data in your node that needs to be copied, you have to override the **copyContents()** method. For example, if the **Pyramid** node class defined earlier contained a private integer member variable called **count** (for some private reason), the **copyContents()** method would look like this:

```
void
Pyramid::copyContents(const SoFieldContainer *fromFC,
                      SbBool copyConnections)
{
    // Copy the usual stuff by calling the base class method.
    SoShape::copyContents(fromFC, copyConnections);
```

```
    // Copy the "count" field explicitly.
    count = ((Pyramid *)fromFC)->count;
}
```

The affectsState() Method

The **affectsState()** method on **SoNode** indicates whether a node has a net effect on the state. (For example, **SoSeparator** changes the state, but it restores the state, so there's no net effect.) The default value for this method is TRUE, but some node classes such as **SoSeparator**, **SoShape**, **SoArray**, and **SoMultipleCopy** define it to be FALSE. When you define a new node class, you may need to redefine its **affectsState()** method if it differs from that of the parent class.

Uncacheable Nodes

You may create a new node whose effects should not be cached during rendering or bounding-box computation. For example, the **SoCallback** node allows a user to change the effect of the callback function, such as drawing a cube instead of a sphere, without ever making an Inventor call.

Uncacheable nodes such as **SoCallback** should call:

```
SoCacheElement::invalidate(state)
```

which aborts construction of the current cache. This call can be made during the render or bounding box action (the two actions that support caching). The **invalidate()** method also turns off auto-caching on any **SoSeparator** nodes over the uncacheable node.

Creating an Alternate Representation

When you create a new node, you probably also want to create an alternate representation for it that can be written to a file. For example, a good alternate representation for the **Glow** node would be an **SoMaterial** node with all fields ignored except for **emissiveColor** and **transparency**. The alternate representation is in the form of a field called **alternateRep**, of type **SoSFNode**. If your node is later read into an Inventor application that is not linked with this new node, Inventor can render the node using this alternate representation even though the node has not been initialized with the database. (See Chapter 11 in *The Inventor Mentor* on reading in extender nodes and engines.)

Within your program, when a change is made to the original node, you may want the alternate representation to change as well. In this case, override the **write()** method on **SoNode** to update the alternate representation, and then have it call the **write()** method of the base class.

Generating Default Normals

If you define your own vertex-based shape class and the parent class does not generate default normals, you need to generate default normals for rendering with the Phong lighting model and for generating primitives. **SoVertexShape** provides the **generateDefaultNormals()** method. Although the specifics depend on the shape itself, **SoNormalBundle** provides methods to facilitate this process.

Tip: If you define a node class that creates a node sensor attached to itself or a field sensor attached to one of its fields, you need to redefine **readInstance()** so that the sensor doesn't fire when the node is read from a file. Your **readInstance()** method needs to detach the sensor, call the **readInstance()** method of the parent class, and then reattach the sensor. Node kits provide the **setUpConnections()** method to make and break these connections (see Chapter 7 in *The Inventor Toolmaker*).

Index

Numbers

3-vertex polygons, 66
4-vertex polygons, 66

A

abstract node classes, 120
actions
  applying, 109
  implementing, 78
  optimizing, 70
  See Also doAction()
  terminating, 111
affectsState(), 121
Alternate group class, 111
Alternate group node, 108, 116
alternate representation, 122
AMBIENT_MASK, 44
analyzing rendering performance, 53
applications
  optimizing, 47-72
  porting, 1
applying actions, 109
automatic normal generation, 12
automatic texture coordinate generation, 12

B

beginShape(), 89
benchmark programs, 48
binding elements, 37
bindings
  DEFAULT, 12
  NONE, 12
bitmasks, 44
blended transparency, 63
blending, 43
bottlenecks, 49
  culling, 66
  level of detail, 67
bounding box, 93

C

caching, 81
  dependencies, 39
  invalidation, 121
  normal bindings, 69
  optimizing, 55
  PER_FACE materials, 69
  See also uncacheable nodes
  turning off render caching, 69

callbacks
  finish, 68
  generating primitives, 88
  start, 68
camera control, 54
CaseVision/Workshop Performance Analyzer, 66
child list, 108
children
  hidden, 108
  traversing, 109-111
classes
  problems caused by changes, 5
clearing windows, 55
clipped objects, 8
CLOCKWISE vertex ordering, 40
colors
  changes, 10
  diffuse, 46, 69
  packed, 46
complexity, 9
  changes, 9
  SCREEN_SPACE, 57
computeBBox(), 93
computeCoordBBox(), 93
constructor
  abstract classes, 120
constructors
  for actions, 70
  nodes, 76
copy(), 36, 120
copyContents(), 36
C pre-processor symbols, 22
creaseAngle default, 8
createLineDetail(), 91
createPointDetail(), 91
createTriangleDetail(), 91
creating details, 90
creating group classes, 108-116
creating nodes, 73-107
creating shape classes, 88-??
culling, 61, 63
  bottlenecks, 66
current element, 43
custom classes, 33
  for performance improvement, 66
  optimizing, 54
  porting, 3
cvspeed, 66

D

debugging library, 75
DEFAULT binding, 12, 13
default normal, 12
default texture coordinate, 12
details
  creating, 90
DIFFUSE_MASK, 44
diffuse colors, 46, 69
DirectionalLight, 61
disabling notification, 56
-DIV_STRICT, 2, 12
doAction(), 78, 82
double-buffering, 51
downgrading files, 30
DSO, 22, 54

E

element bundles, 81
elements
  accessing, 80
  enabling, 75
  setting, 80
EMISSIVE_MASK, 44
enableNotify(), 72
enabling elements, 75
endShape(), 89
engines, 46
enumerated values, 77
event handling performance, 72
examples
  group class, 111-116
  property class, 83-87
  shape class, 94-107
extender API changes, 33
extensions
  texture object, 29

F

face set performance, 58
fields, 77
field types
  writing, 117
file format
  unknown nodes, 117
file header methods, 23
files
  downgrading, 30
  reading, 23
  writing, 23
file version, 21
finish callback, 68
flags
  isBuiltIn, 117
  set_version, 22
flat-shaded polygons, 63
floating point precision, 23
fog, 62
frame rate, 51
  performance goal, 48

G

generatePrimitives(), 88, 89, 91
generating normals, 12, 40
generating primitives, 88
get(), 80, 81
getBoundingBox(), 88, 93
getChildren(), 109
getInstance(), 80
getNum(), 81
getPathCode(), 109
GL_BLENDING, 43
GL_POLYGON_STIPPLE, 43
glColor(), 44
glColorMaterial(), 42
glMaterial, 44
Glow property class, 82, 83
Glow property node, 116
GLRender(), 34, 88, 90
GLRenderBelowPath(), 34
GLRenderInPath(), 34
GLRenderOffPath(), 34
glShadeModel(GL_FLAT), 43
Gouraud-shaded polygons, 63
gr_osview, 49
grabEvents(), 72

group classes
  creating, 108-116
  example, 111-116
groups
  hidden children, 108
  path code, 109-111

H

hardware-dependent performance, 51
hardware texture mapping, 9
hidden children, 108

I

implementing actions, 78
inheritance
  element stack, 76
initClass(), 75, 76
initClass() method
  nodes, 120
initialization
  node classes, 116
initializing nodes, 75
Internet, 26
invalidate(), 121
invokeLineSegmentCallbacks(), 89
invokePointCallbacks(), 89
invokeTriangleCallbacks(), 89
isBuiltIn flag, 117
isEngineModifying(), 46
isolating rendering, 53
ivAddVP, 30
ivdowngrade, 30
ivfix, 31
ivperf, 31, 53
  and culling, 66
  and OpenGL, 64
  camera control, 54
ivview -p, 48

K

keeping records, 51

L

LD_LIBRARY_PATH, 1
level of detail bottlenecks, 67
lights
  optimizing, 62
line segment callback function, 89
line segments, 88
LOD node, 26
longToInt32, 18

M

macros
  abstract versions, 120
  nodes, 74
manipulators, 29
masks, 44
material binding, 62
materialIndex field, 57
materials
  changes, 10
material state elements, 38
measuring performance, 49

memory
  optimizing, 69
multiple colors, 43
multiple materials, 10

N

node classes
  abstract, 120
  extender
    using, 116-120
  initializing, 116, 117
node kits
  hidden children, 108
node override, 38, 39
nodes
  constructor, 76
  creating, 73-107
  initializing, 75
  macros, 74
  uncacheable, 121
NONE binding, 12
normals
  changes, 12
  default, 12
  generating, 40
  requirement for SoNormalBinding, 12
notification
  disabling, 56
notification overhead, 71
numVertices field, 13

O

OpenGL, 76, 90
  performance, 64
  texture object extension, 29
OpenGL benchmark programs, 48
optimizing actions, 70
optimizing applications, 47
optimizing caching, 55
optimizing custom classes, 54
optimizing event handling, 72
optimizing face sets, 58
optimizing level of detail, 67
optimizing lights, 61
optimizing memory usage, 69
optimizing picking, 72
optimizing pixel fill, 63
optimizing rendering, 52
optimizing scene graphs, 31
optimizing textures, 59
optimizing transformations, 58
optimizing vertex transformations, 61
optimizing window clearing, 55
osview
  swapbuf number, 49
Override flag, 16
override status
  and caching, 57
overriding nodes, 38, 39

P

packed color, 46
path code, 109-111
PER_FACE materials, 69
performance, 47
  hardware dependency, 51
  hardware limitations, 48
  interrupting rendering, 68
  keeping records, 51
  lights, 61

measuring, 49
rendering test, 52
SoTransform, 58
performance goal, 48
picked point, 91
picking, 91
performance, 72
pixel filling
performance, 63
pixie, 66
point callback function, 89
PointLight, 61
points, 88
pop(), 43
porting, 1
custom classes, 3
pre-processor symbols, 22
primitives
generating, 88
performance, 62
primitive shapes, 8
prof, 66
property classes
creating, 82-87
example, 83-84
prototyping, 88
push(), 43
Pyramid shape node, 94, 116

R

rayPick(), 88, 91
read(), 108
reading files, 23
readInstance(), 35
reading children, 35
record-keeping, 51
render caching, 69
renderCaching ON, 60
render culling, 61, 63, 66
rendering, 90
analyzing performance, 53
experiments, 53
interrupt for performance, 68
isolating rendering, 53
optimizing rendering, 52
rendering methods, 34
rendering performance
analyzing with ivperf, 31
reset(), 44
revision symbols, 22

S

scene graphs
optimizing, 31
scene traversal, 55
SCREEN_DOOR transparency, 63
SCREEN_SPACE complexity, 57
sendAllMaterial(), 42
sendDiffuseByIndex(), 43
sending multiple colors, 43
sendNoMaterial(), 42
sendOnlyDiffuseColor(), 42
set(), 80
set_version flag, 22
setDetail(), 90
setDrawStyle(), 14
setFloatPrecision(), 23
setTransparencyType(), 63
shade model, 43
shademodel(FLAT), 63

shape classes, 40
  creating, 88-107
  example, 94-107
  SoMaterialBundle, 44
shape hints
  changes, 8
  element methods, 37
shapes
  sending multiple colors, 43
  vertex-based, 2
shapeVertex(), 89
shininess, 57
SHININESS_MASK, 44
shouldGLRender()
  SoShape method, 90
shouldRayPick()
  SoShape method, 91
SO_ENABLE(), 75
SO_ENGINE_OUTPUT, 46
SO_NODE_ADD_FIELD(), 77
SO_NODE_CONSTRUCTOR(), 76
SO_NODE_DEFINE_ENUM_VALUE(), 77
SO_NODE_HEADER(), 74
SO_NODE_INIT_CLASS(), 34, 75
SO_NODE_IS_FIRST_INSTANCE(), 76
SO_NODE_SET_SF_ENUM_TYPE(), 77
SO_NODE_SOURCE(), 74
SO_SWITCH_NODE, 52
SO_VERSION, 22
SO_VERSION_REVISION, 22
SoAction.h, 75
SoAsciiText, 26
SoBoundingBoxAction::apply(), 66
SoCallback, 121
SoCallbackAction, 88
SoChildList, 108
SoClipPlane
  clipped objects, 8
  wireframe objects, 8
SoColorPacker, 46
SoComplexity, 61
  and caching, 57
  complexity, 9
  no hardware texture mapping, 9
  textureQuality field, 9
SoCone
  primitive shapes, 8
SoCoordinate, 64
SoCube
  primitive shapes, 8
SoCylinder
  primitive shapes, 8
SoDB
  file header methods, 23
SoDetail, 89
SoDrawStyle
  clipped objects, 8
  wireframe objects, 8
SoElement, 39
  override, 38
SoFaceSet, 64
  3-vertex polygons, 66
  4-vertex polygons, 66
  optimizing, 62
  performance, 58
SoFieldContainer, 117
SoFieldContainer enableNotify(), 72
SoFontStyle, 26
SoGLLazyElement, 38
SoGLLazyElement::reset(), 41
SoGLLazyElement::sendAllMaterial(), 42
SoGLLazyElement::sendDiffuseByIndex, 43
SoGLRenderAction::addMethod(), 34

SoGLRenderAction::setAbortCallback(), 68
SoGLTextureQualityElement, 37
SoGroup, 108
  implementation of copying, 36
SoIndexedFaceSet
  3-vertex polygons, 66
  4-vertex polygons, 66
  performance, 58
SoIndexedShape, 93
SoLabel
  caching, 56
SoLazyElement, 38
  masks, 44
SoLevelOfDetail, 26
  caching, 56
SoLightModel
  BASE_COLOR, 62
SoLocateHighlight, 26
SoLOD, 26, 61, 62
  caching, 56
  optimizing, 67
SoMaterial
  multiple materials, 10
  override flag, 16
SoMaterialBinding
  DEFAULT, 37
  NONE, 37
  override flag, 16
SoMaterialBundle, 81
  new shape nodes, 44
SoNode::copy(), 36
SoNode::enableNotify(), 56
SoNode parameter, 39
SoNonIndexedShape, 13, 93
  USE_REST_OF_VERTICES, 13
SoNormal
  automatic normal generation, 12
  default normal, 12
  normal requirement, 12
SoNormalBinding
  automatic normal generation, 12
  DEFAULT binding, 12
  NONE binding, 12
  normal requirement, 12
SoNormalBundle, 81
SoNormalGenerator class constructor, 40
SoOutput
  file header, 23
  isASCIIHeader(), 16
  isBinaryHeader(), 16
  setFloatPrecision(), 23
SoOverrideElement, 39, 46
SoPickedPoint, 91
SoPickStyle
  caching, 56
SoPrimitiveVertex, 89
SoReplacedElement
  SoNode parameter, 39
SoSeparator
  caching, 56
  renderCaching ON, 60
SoSeparator::cullTest(), 66
SoShape, 88
SoShape::beginSolidShape(), 42
SoShape::endSolidShape(), 42
SoShapeHints
  clipped objects, 8
  creaseAngle default, 8
  primitive shapes, 8
  shapeType, 62
  vertexOrdering, 62
  wireframe objects, 8
SoShapeHintsElement
  method changes, 37

SoSphere
  primitive shapes, 8
SoSubNode.h, 74
SoSwitch
  whichChild field, 52
SoText2
  and caching, 57
SoTexture2
  no hardware texture mapping, 9
  textureQuality field, 9
SoTextureBlendColorElement, 38
SoTextureCoordinate2
  automatic coordinate generations, 12
  default texture coordinate, 12
  texture coordinate requirement, 13
SoTextureCoordinateBinding
  DEFAULT binding, 13
SoTextureCoordinateBundle, 81
SoTextureModelElement, 38
SoTextureQualityElement, 37
SoTextureWrapSElement, 38
SoTextureWrapTElement, 38
SoTransform
  performance, 58
SoTransformerManip, 29
SoTriangleStripSet
  numVertices field, 13
SoType::overrideType(), 34
SoVertexProperty, 24
SoVertexShape
  complexity, 9
SoWWWAnchor, 26
SoWWWInline, 26
SoXtViewer
  texture mapping, 14
  viewers, 14
SPECULAR_MASK, 44
sphere
  rayPick(), 92-93
SpotLight, 61
start callback, 68
startNotify(), 71
state elements, 81
stipple transparency, 43
swapbuf in osview, 49

T

terminating actions, 111
texture coordinates
  automatic generation, 12
  changes, 12
  default, 12
  requirement, 13
texture mapping
  and viewers, 14
texture object extension, 29
texture quality element, 37
textureQuality field, 9
textures
  optimizing, 59
The Inventor Mentor, 90, 122
transformations
  optimizing, 58
transparency, 43
  and diffuse colors, 46
  blended, 63
  SCREEN_DOOR, 63
traversal, 55
traversal state
  enabling elements, 75
traverse(), 108
traverseChildren(), 108

triangle callback function, 88
triangle fans, 89
triangles, 88
triangle strips, 89

U

uncacheable nodes, 121
unknown nodes
  file format, 117
USE_REST_OF_VERTICES, 13
user-defined file headers, 23

V

version number, 22
vertex-based shapes
  changes, 13
  vertex property field, 2
vertex ordering
  CLOCKWISE, 40
vertex properties
  converting files, 30
vertexProperty field, 2, 30
vertex property node, 24
vertex transformations
  performance, 61
viewAll(), 54
viewers
  and texture mapping, 14
Virtual Reality Modeling Language, 26
VRML
  nodes, 26

W

wireframe, 63
wireframe objects, 8
write(), 108
writing files, 23

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3078-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389