Optimizing Open Inventor™ Applications

Release 2.0

Copyright © 1993, Silicon Graphics, Inc. All rights reserved.

Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks, and Inventor and Graphics Library are trademarks of Silicon Graphics.

Specifications are subject to change without notice.

Section 1 Introduction

This document describes a process for determining what is limiting the performance of your Open Inventor application, and provides suggestions on how to improve its performance.

The following sections are provided:

Section 2: Benchmarking Tips- Discovering where an application is spending its time is not a matter of random trial and error. This section describes a process for effectively investigating and improving performance.

Section 3: Optimizing Rendering- For each step in the rendering process, this section:

- Briefly describes that step.
- Presents a method for determining how much time your application is spending in that step.
- Describes techniques for reducing that time.

Section 4: Optimizing Everything Else- This section 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 Inventor programming in general and on specific nodes, see *The Inventor Mentor*.

Section 2 Benchmarking T ips

Performance tuning is a necessary chore when developing any application, like fixing bugs or eliminating memory leaks. Proper organization and planning can make this chore much quicker and more pleasant.

The general approach to optimizing your application should be:

1. Set performance goals
2. Devise a method to measure your application's performance
3. Determine where your application is spending its time
4. Modify your application to reduce the bottleneck
5. Finally, measure your application against your goal again and repeat steps 2 and 3 until it meets your goal.

The rest of this section explains these steps in more detail, concentrating on optimizing the rendering performance of Inventor applications.

2.1 Set Performance Goals

Setting a performance goal will help you use your time wisely. Typically, you should decide on a desired frame ratel, 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 will find that frame rate acceptable for most tasks (more is always better, of course).

When setting a goal, you should keep in mind the capabilities of your hardware. If the absolute top speed for drawing polygons on your machine is 60,000 unlit, single-color triangles per second and you are trying to get 10 frames per second while drawing 6,000 lit, color-per-vertex triangles, you will be disappointed. Write short OpenGL benchmark programs, or feed test scene graphs to `ivview` to help set your expectations.

2.2 Measure Performance

It is very important to have an objective way of measuring your application's performance; watching your application run and trying to see if it *seems* faster is likely to cause you to waste your time on insignificant optimizations.

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 the source code for the `ivview` application in `/usr/share/src/Inventor/tools/ivview`, for example.

The `osview` program can also be useful. The *swapbuf* number in the *Graphics* section will also tell 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 will give you (and your boss) a feeling of accomplishment, and will ensure that you are not making things worse.

2.3 Determine 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 machines 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, and then you should devise an experiment that will prove or disprove that theory. Create experiments that isolate one narrow part of your application's performance, and make sure you understand what you are measuring every time you run an experiment.

Section 3 and Section 4 describe the following bottlenecks, show how to determine the amount of time your application is spending in each of them, and give suggestions on improving them:

- Section 3.4: Window Clear Bottleneck
- Section 3.4: Traversal Bottleneck
- Section 3.13: Pixel Fill
- Section 3.12: Vertex transformations
- Section 3.14: Thing ivRender won't measure
- Section 3.16: Culling Bottleneck
- Section 3.17: Level Of Detail Bottleneck
- Section 4.2: Memory Usage
- Section 4.4: Action Construction and Setup
- Section 4.5: Notification
- Section 4.6: Picking and Handling Events

If you have some reason to suspect one of these bottlenecks more than another, you can perform these sections in any order; just make sure you always know what you are measuring and keep good records of your experiments.

2.4 Modify Your Application

Be careful when you apply a performance optimization. Make sure that your modification is actually an improvement: don't assume that all of the suggestions made in this (or any other) document will automatically apply to your application. For example, using Inventor render caching will usually increase performance. However, if the extra memory used by render caching causes your application to run out of memory and start swapping memory to disk, it might actually hurt performance.

Again, keep good records. Record what you did and how much it improved 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 very tempting to read a document like this, make lots of changes, and then see if the application gets faster. Not only does this waste time, but it could be counter productive.

2.5 Are We Done 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 re-examine your goals.
- **Is your application running at 60 or 72 frames per second?** Double-buffered programs will never render faster than the refresh rate of your monitor, and Inventor sets its animation sensors to fire at a maximum of 60 frames per second (for example, the realTime global field is updated 60 times a second by default).
- **Do you need to experiment on different machines?** Different machines will have different bottlenecks; on a machine with very fast graphics, the bottleneck is more likely to be either in the application's code or in Inventor's code. On a machine with slow graphics and a fast CPU, the bottleneck is much more likely to be inside the OpenGL calls. If your application will be used on different types of machines, **make sure performance is acceptable on all of them.**

2.6 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 machines your application supports

Section 3 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!

3.1 Is Rendering the Problem?

The first step is to 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 except for render 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 section Section 4 for information on optimizing the rest of your application.

3.2 Isolate Rendering

Once 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 determine where the bottleneck in rendering is occurring. The easiest way of doing this is to write your scene to a file and then use the `ivPerf` program (see Appendix 1 for information on how to get `ivPerf` via anonymous ftp or the World-Wide-Web) to perform a series of rendering experiments.

The code for writing out your scene will look something like this:

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

3.3 Using ivPerf

`ivPerf` is a utility that reads in a scene graph and analyzes its rendering performance. It provides feedback about 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 will not, in general, be equal to the total time taken 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 will 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 rendering time, how much faster would rendering be?". For example, if `ivPerf` tells you 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 Section 2.12 of the inventor_dev product's release notes for more information on creating DSO's) or call their `initClass()` methods just after the call to `SoInteraction::init()` in the `ivPerf` source and link their `.o` files into `ivPerf`.

`ivPerf`'s camera control is very simplistic: it does a `viewAll()` of 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, you should modify `ivPerf` so that its camera motion is more appropriate for your application. For example, have `ivPerf` use the following little scene:

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

... instead of just a `PerspectiveCamera`.

`ivPerf` will correctly report the performance of changing scenes, as long as you give it enough information . It automatically deals with scenes containing engines, but if you are using `SoSensors` 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 { }
```

3.4 Window Clear Bottleneck

The first step in the rendering process is clearing the window. It is easy to forget about this step, but depending on the size of your application's window and the type of machine 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 big, run `ivPerf` like this:

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

ivPerf will perform many different rendering experiments, and will eventually print out information on each rendering stage.

For example, on an Indigo2 Extreme running IRIX 5.2, ivPerf reports that rendering a simple cube in a 1,000 by 1,000 pixel window 66% of the time is spent clearing the window.

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

3.5 Caching Behavior

After the previous experiment, you know how much time your application is spending clearing the color and depth buffers. The next experiment is designed to find out how much time Inventor is spending **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 will automatically **cache** the parts of your scene that aren't changing, 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` will measure the difference between rendering your scene with nothing changing, and rendering with the camera, engines, and nodes named "NoCache" changing.

The next section discusses ways of organizing your scene so that Inventor will be able to cache most of it. If you are using `SoLevelOfDetail` nodes or render culling, also see Sections 3.16 and 3.17 for hints on optimizing those features, which `ivPerf` also reports as part of caching behavior.

3.6 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 it is changing. You should be aware that the following things inhibit caching:

- Changing fields in the scene will destroy caches inside all `SoSeparator`s above the node that changed. Even fields that do not affect rendering, such as fields in the `SoLabel` or `SoPickStyle` nodes, will destroy caches if they are changed.
- The `SoLevelOfDetail` node will break caches above it whenever either the camera or any of the matrix nodes affecting it change. Make the children of the `SoLevelOfDetail` node `SoSeparator` nodes, so that they will be cached. See Section 3.17 for more information on efficient use of the `SoLevelOfDetail` node.
- Any shape using `SCREEN_SPACE` complexity will break caches above it whenever the camera or any of the matrix nodes affecting it change.
- The `SoText2` node will break 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, you should try to separate `SoText2` nodes from the other objects in your scene, to allow the other objects to be cached.
- 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 will 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 would be bad for caching.
- Generating default normals for vertex-based shapes that cannot be cached is very expensive because it must be done every time the shape is rendered. Use `ivquicken` to generate normals for your objects if they will not be cached.

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

3.7 Material Changes

If ivPerf reports that material changes are your rendering bottleneck, try the following:

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

- If your SoMaterial nodes have multiple values in them, note that having multiple values in just one of the fields is faster than having multiple values in several of the fields. For example, indexing into an SoMaterial node with 10 diffuseColors and 1 ambientColor will be faster than indexing into an SoMaterial node with 10 diffuseColors and 10 ambientColors.
- 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 will only work for PER_PART or PER_FACE material bindings, of course).

3.8 Transformations

ivPerf reports two numbers for transformations; the overhead of changing the GL 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 Section 3.12 for hints on optimizing the transformation of vertices.

- If you are using vertex-based shapes (SoIndexedFaceSet, SoTriangleStripSet, etc.), use ivquicken to get rid of transformations. See the man page for ivquicken for more information.
- Use SoRotation, SoRotationXYZ, SoScale, or SoTranslation nodes instead of the general SoTransform node. Don't bother doing this if you would have to replace the SoTransform node with more than one of the simpler nodes to get the same transformation.

3.9 Textures

ivPerf will report two numbers for textures, if your scene contains textures. It reports 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 machines with texturing hardware, the number of textures used can dramatically affect performance; see Section 3.10 for hints on optimizing texture management. On machines without texture mapping hardware, the bottleneck will probably be actually filling in the textured polygons, in which case you should try:

- Insert an SoComplexity node in your scene with its textureQuality field set lower than its default value of 0.5. A textureQuality of zero will turn off texturing; other values will cause OpenGL to use different filters for texturing, which can affect polygon fill performance. Rerun ivPerf to see if this makes a difference on your hardware with your scene.
- Use the viewer's "Move No Texture" mode so that interaction is fast, saving the high-quality textured image for when the user is not interacting with the view.

3.10 Texture Management

If `ivPerf` reports a lot of time being spent in texture management, then you are running out of hardware texture memory. Try:

- Use smaller textures. Use the `izoom` utility to scale down the images you are using; accidently using one big image can easily fill up texture memory on many machines.
- Because Open Inventor allows you to modify a texture once it has been read into your application (using the `image` field of `SoTexture2`), and because Open Inventor allows you to change the search path for textures (using methods on `SoInput`), it does not use the same texture memory for two different `SoTexture2` nodes with the same `filename` field. Be sure to re-use 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 is much more texture-memory efficient:

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

- Textures may not be shared if they are below an `SoSeparator` with `renderCaching` turned `ON`. Textures use less texture memory by building an OpenGL display list containing the OpenGL texture commands. However, if there is already an OpenGL display list being 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 `Separator`'s `renderCaching` field set to `ON`:

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

- Use `SoLevelOfDetail` nodes to create simpler versions of your objects that are not textured 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 containing 100 textured objects (each with a unique texture), but only 10 of them are in the view volume at any given time. When rendered, only 10 of the textures will need to be in texture memory at any given time, resulting in much better texture management performance.

3.11 Lights

If the scene given to ivPerf contains light sources, ivPerf will tell 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.

3.12 Vertex transformations

If `ivPerf` reports that vertex transformations is 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 machine 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 `SoLevelOfDetail` 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 will not notice background objects "popping" between levels of detail. Beware that `SoLevelOfDetail` nodes will cause smaller caches to be built, which may slow down traversal. See Section 3.17 for more information on efficient use of level of detail.
- Make your vertices simpler. Try to use `OVERALL` rather than `PER_VERTEX` material binding (putting a `MaterialBinding { value OVERALL }` node in the "override.iv" file and re-running `ivPerf` is a quick way of seeing if this might help). Turn off fog. Note that these suggestions are machine-specific; on machines 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.
- If you are using `SoFaceSets` or `SoIndexedFaceSets`, try using `ivquicken` to convert them into `SoIndexedTriangleStripSets`, which draw more triangles with fewer vertices. Note that `ivquicken` won't be able to 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 `renderCulling` 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 Section 3.15 for hints on making Inventor produce more efficient OpenGL calls.

3.13 Pixel Fill

A common bottleneck on low-end machines 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 a large percentage of each frame is spent filling in pixels, try to optimize your scene by:

- Rendering 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 machines can fill flat-shaded polygons faster than Gouraud-shaded polygons. When in RGB mode, Inventor never changes the shade model from OpenGL's default, which is Gouraud shading. Try adding a call to `glShadeModel(GL_FLAT)` after the call to `glXMakeCurrent` in `ivPerf` and re-running your scene. If there is a significant performance increase, you should determine which parts of your scene look good flat-shaded and insert `SoCallback` nodes to turn flat-shading on and off for those parts of your scene. Note that an `SoLightModel` node set to `BASE_COLOR` lighting does not turn on flat shading, since unlit primitives may still have different colors at each vertex and be Gouraud shaded.
- `SCREEN_DOOR` transparency (the default) is faster than the blended transparency on some machines (it is slower on some machines, too). Use the `setTransparencyType()` method on either `SoXtRenderArea` or `SoGLRenderAction` to change the transparency type.

3.14 Thing ivPerf won't measure

There are several performance problems that `ivPerf` won't catch. The following sections describe them, and give hints on how to improve them:

3.15 Making Inventor produce efficient OpenGL

If your application is rendering only 10 frames/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 could create a multifaceted polygonal shape using a number of different coordinate and face set nodes, as shown in the lower half of Figure 1. A much better technique is to put all the coordinates for the polygonal shape into one `SoCoordinate` node, and the description of all the face sets into a second `SoFaceSet` node, as shown in the upper half of Figure 1.

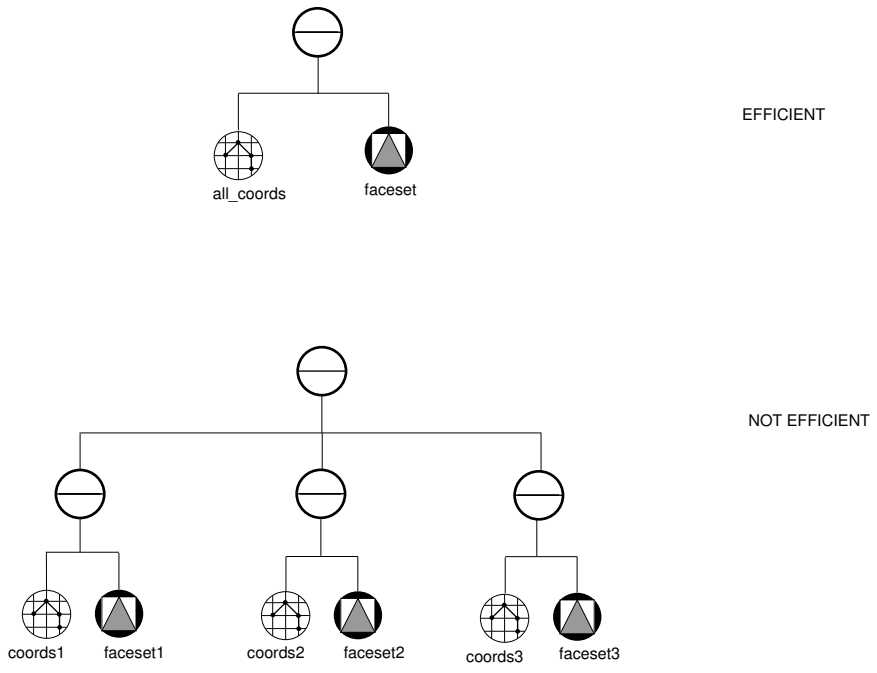

Figure 1 Condensing face sets into fewer nodes

Using fewer nodes to get the same picture reduces traversal overhead for scenes that cannot be cached. Arranging your scene so that each node makes more OpenGL calls also gives Inventor a better opportunity to produce efficient OpenGL calls.

For some applications, you should 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/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 (e.g. 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, etc. See *The Inventor Toolmaker* for complete information on implementing your own nodes.

3.16 Culling Bottleneck

If your application is using render culling, it might be spending most of its time deciding whether or not objects should be culled. `ivPerf` will lump 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 `SoBoundingBox::transform()` routines. See the man pages for `pixie`, `prof`, or `cvspeed` for information on using these tools.

If a large percentage of the rendering time is being spent doing cull tests, try to re-organize 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 will also allow Inventor to build larger render caches, which will 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 `SoSeparators` above an `SoSeparator` doing render culling have their `renderCaching` fields set to `ON`.

3.17 Level Of Detail Bottleneck

Like render culling, if your application is using `SoLevelOfDetail` 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 `SoLevelOfDetail` 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:

- Use a level of detail node with a simpler, faster level of detail test. See Appendix 1 for information on obtaining the source to `LODD`, which has a very simple, very fast test.
  Note: There is a bug in Inventor 2.0 which causes `SoLevelOfDetail` nodes to be slow when used with render culling. This bug will be fixed in Inventor 2.0.1; use the `LODD` node as a workaround until then.

- Try to group objects together 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 will also make it easier for Inventor to build larger render caches, which will 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 `SoSeparators` above an `SoLevelOfDetail` have their `renderCaching` fields set to `ON`.

3.18 Making it 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 only abort 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.

Section 4 Optimizing Everything Else

Once 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, this section suggests ways of finding other bottlenecks, and suggests Inventor-specific things to look for.

4.1 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 man pages for `pixie`, `prof`, or `cvspeed` for information on using these tools.

4.2 Memory Usage

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

- 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`.
- If you are using caching, avoid using `PER_FACE` or `PER_FACE_INDEXED` material 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 wherever possible instead of duplicating geometry or properties. Instancing will make your scene graph take up less memory and will also enable Inventor build OpenGL display lists that are used more than once. This is especially important for `SoTexture2` nodes.

4.3 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 that are taking a significant percentage of time, or application routines that take a significant percentage of time. If a lot of time is being spent in `SoGLRenderAction::beginTraversal()`, see Section 3 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.

4.4 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, try to create an action once and reapply it instead of constructing a new action if your performance traces show a lot of time being spent inside an action's constructor or the `SoAction::setUpState()` method.
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; // BAD if done a lot!
bbAction.apply(myScene);
```

4.5 Notification

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 being spent in a `startNotify()` method, then 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, etc. 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 lots of `SoPathSensor`s 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`s themselves do not have this problem in Inventor 2 (but they did in Inventor 1).
- Notification can be enabled or disable on a per-node or per-engine basis, if absolutely necessary. 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` man page for information on the `enableNotify()` method.

4.6 Picking and Handling Events

If your application profiles show a lot of time being spent inside the `SoPickAction::beginTraversal()` or `SoHandleEventAction::beginTraversal()` methods, 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 (e.g. “dead” background graphics).
- Insert `SoPickStyle::BOUNDING_BOX` nodes in your scene if you do not need detailed picking information. This will help most for complicated objects like `SoText3` or `SoTriangleStripSet`s with lots of triangles.
- If you are using `SoPickStyle::SHAPE` (the default), put explicit `SoNormal` nodes in your scene. Otherwise, Inventor will have to spend time generating normals for you every time an object is picked.

- 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.