Optimizing Open Inventor™ Applications

Release 2.0

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 Tips

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 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`.

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.

2.2 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 3.3: Window Clear Bottleneck
- Section 3.4: Traversal Bottleneck
- Section 3.7: Pixel Fill Bottleneck
- Section 3.8: Transforming Vertices Bottleneck
- Section 3.9: State Change Bottleneck
- Section 3.10: Culling Bottleneck
- Section 3.11: 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.

2.3 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.

2.4 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:

- **Is the application fast enough?** If you have a specific performance number that you must meet (it must render at least 15 frames per second) then this decision is easy. If not, come up with a specific performance number; for example, most users will find a frame rate of 10 frames per second acceptable for an interactive program (more is always better, of course). Try to get your application to run this fast for a typical scene.
- **Are your expectations reasonable?** If the absolute top speed for drawing polygons on your machine is 60,000 unlit, non-depth-buffered triangles per second and you are trying to get 10 frames per second while drawing 6,000 lit, depth-buffered triangles, you will be disappointed. Write short OpenGL benchmark programs, or feed test scene graphs to `ivview`, to help set your expectations.
- **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.**

2.5 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 `ivRender` program (source code in Appendix I; see the comments in Appendix I for ftp/www availability) 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);
```

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 `SoDB::init()` in the `ivRender` source and link their `.o` files into `ivRender`.

`ivRender`'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 `ivRender` so that its camera motion, level of detail switching, and render culling are more appropriate for your application. For example, add something like the following to the beginning of your scene:

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

... and modify `ivRender` so that it either updates the `realTime` global field in its rendering loop (so that the `SoRotor` node rotates) or so that it directly modifies the `SoRotor` node. Do the following to update `realTime`:

```
SoSFTime *realTime =
 (SoSFTime *)SoDB::getGlobalField("realTime");
realTime->setValue(SbTime::getTimeOfDay());
```

3.3 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 800 by 800 pixels big, run `ivRender` like this:

```
ivRender -an -w 800,800 myScene.iv
```

The options given mean:

- -w 800,800 : Make `ivRender` use a window that is 800 pixels wide by 800 pixels high.
- -a : Do not actually apply an `SoGLRenderAction`. The purpose of this experiment is to see how fast the machine can clear the color and depth-buffers only, to give an upper limit on rendering performance.
- -n : Don't bother adding a camera, light, or transform, since we are only trying to measure screen clear speed.

For example, on an Indigo2 Extreme running IRIX 5.2, this experiment gives 48 frames per second for a 1,000 by 1,000 window. Put another way, if the application must run at 30 frames per second, then over 60% of the time you have to render a frame would be 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).

3.4 Traversal Bottleneck

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. Run `ivRender` again on your scene, this time applying the `SoGLRenderAction` and using a reasonable camera:

```
ivRender myScene.iv
```

Run the experiment again, this time with **render caching** turned off:

ivRender -r myScene.iv

- -r : Do not build render caches at `SoSeparators`.

See *The Inventor Mentor*, Chapter 9, for an explanation of render caching. Basically, render caching eliminates Inventor's traversal of the scene. The difference between these two experiments is the amount of traversal overhead in your scene, when Inventor cannot build render caches. By default, Inventor will automatically build render caches at `SoSeparator`s where appropriate. However, 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.

Compare the results of the cached experiment with your application's performance. If your application is about as fast as the fully cached experiment, your application is using caches efficiently and traversal during rendering is not your bottleneck; skip to Section 3.7 to look for other possible bottlenecks. Also skip to Section 3.7 if the cached and uncached times are fairly close to each other; this means that Inventor is able to render your scene efficiently even without the benefit of caching .

The next two sections discuss two ways of reducing traversal overhead: creating more efficient scene graphs and structuring your scenes so that Inventor is able to build render caches.

3.5 Reducing Traversal Overhead

Take a look at the number of nodes and triangles that are in your scene, as reported by `ivRender`. The number of nodes should be small compared to the number of triangles. Get rid of any redundant nodes, get rid of `SoLabel` nodes by using Open Inventor's naming feature (see Chapter 3 of *The Inventor Mentor* for more information). Try to arrange your scene so that property nodes affect a large part of the scene; for example, if all of the objects in your scene are solid, put one

`SoShapeHints` node at the top of the scene instead of inserting it before each of the objects.

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.

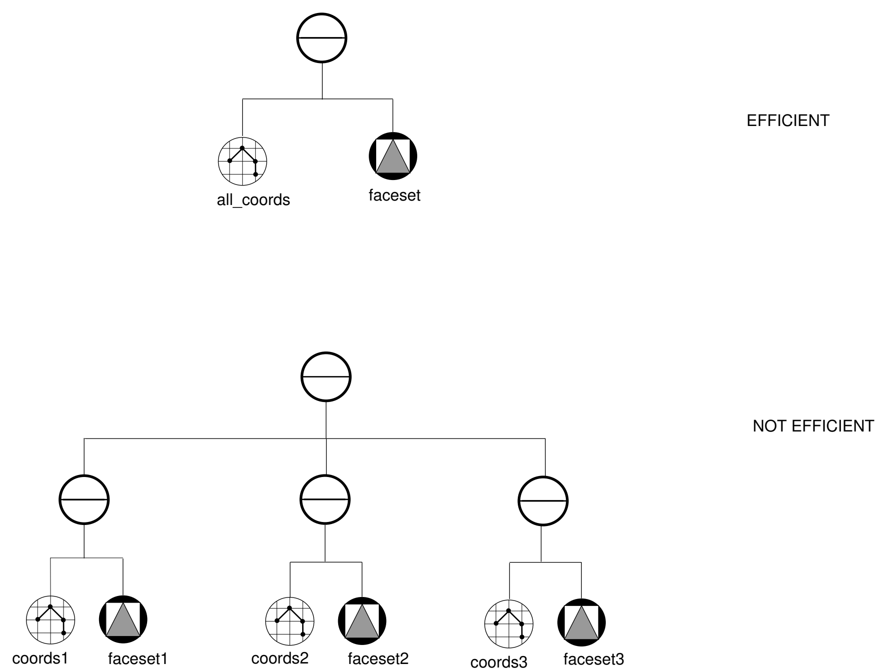

Figure 1 Condensing face sets into fewer nodes

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.

3.6 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 3.11 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 Pixel Fill Bottleneck

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.

To determine whether or not your application is fill-limited, run `ivRender` again, this time instructing it to insert an `SoDrawStyle` node that will override the draw style in the scene so that everything is drawn as points. Do this by creating an "override.iv" file containing this:

```
#Inventor V2.0 ascii
DrawStyle { style POINTS }
```

and then running `ivRender` like this:

```
ivRender myScene.iv override.iv
```

`ivRender` reads the second file given, sets the override flag for all nodes in that file, and inserts them before the nodes in the first file.

The difference between these two experiments is the amount of time being spent filling in the polygons in your scene. If the difference is large, you can speed up pixel fill 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.
- If you are using texturing, see if turning off texturing or decreasing the `SoComplexity::textureQuality` field increases performance. 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. Put a `Complexity` node in the "override.iv" file and rerun `ivRender` to see if this makes a difference on your hardware with your scene.
- Some machines can fill flat-shaded polygons faster than Gouraud-shaded polygons. 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 `ivRender` 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.

3.8 Transforming Vertices Bottleneck

Modify the "override.iv" file created in the previous experiments, changing the `SoDrawStyle` from `POINTS` to `INVISIBLE`, and rerun `ivRender`. The difference between the `POINTS` and `INVISIBLE` experiments is the amount of time spent lighting, fogging, and transforming the vertices of the objects in your scene. If you find that this time 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, and 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 3.11 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 `ivRender` 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 `ivRender` test before spending time modifying your application or scene.
- 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.
- 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`. `ivRender` 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.

3.9 State Change Bottleneck

You might have run the `DrawStyle INVISIBLE` experiment from the last section and been surprised that your scene still doesn't draw as quickly as you expect, even though nothing is being drawn. When `DrawStyle` is `INVISIBLE`, Inventor's shapes do nothing, but Inventor's property nodes will still issue OpenGL calls, since the `DrawStyle` might be reset later during traversal and OpenGL's state must be correct when that happens. So, if rendering isn't as fast as you expect with a cached, `INVISIBLE` scene then executing the GL calls to change properties may be the culprit (other possibilities are level of detail switching and render culling, which the next two sections discuss).

To optimize state changes, first take a look at the structure of your scene. The following little shell command is a good way of getting an overview of your scene's structure:

```
ivcat myScene.iv | egrep '[{}]|USE'
```

Common causes of state change bottlenecks are `SoTransform`, `SoMaterial`, `SoTexture2`, and `SoShapeHints` nodes. If you see that your scene contains a lot of these nodes (or any other property node), create an override.iv file for `ivRender` that overrides that property. Of course, without these nodes, `ivRender` will render your scene incorrectly. Use the following techniques to get both faster and correct rendering if you have a state-change bottleneck:

- If overriding `SoMaterial` nodes increases performance, try the following:
  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).
- If overriding `SoTransform` nodes increases performance, see if you can use `ivquicken` to get rid of them. 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.
- If overriding `SoTexture2` nodes increases performance and your machine supports textures in hardware, you may be running out of texture memory. Unlike Inventor 1, Open Inventor does not automatically use the same texture if two `SoTexture2` nodes happen to have the same filename. Be sure to use instancing (`DEF` and `USE` in the file format) instead of creating several texture nodes with the same filename.

- Try organizing your scene to minimize state changes. For example, if you have several objects that use the same texture, group them together so that the texture is traversed only once. Note that organizing your scene for efficient state changes might interfere with attempts to organize your scene for efficient render culling, where you want objects that are spatially near each other to be grouped together. You will have to find the optimal balance for your application.
- Use render culling to avoid traversing objects with state changes that are outside the view. See Chapter 9 of *The Inventor Mentor* for an explanation of culling.
- Create simpler levels of detail with fewer state changes that can be used for objects that appear small.

3.10 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. 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`.

3.11 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 `ivRender` again (with `DrawStyle` still set to `INVISIBLE` so that the high-complexity objects aren't actually drawn) 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 2 for 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`.

3.12 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 `SoPathSensors` 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:** `SoPaths` 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 `SoTriangleStripSets` 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.

Appendix 1 ivRender Source

This is the source for the `ivRender` tool. It is also available on the internet via anonymous ftp from ftp.sgi.com in the file sgi/inventor/2.0/ivRender.C. If you are using a WWW browser such as Mosaic, use the URL: ftp://ftp.sgi.com/sgi/inventor/2.0/ivRender.C

To compile this code, save it in a file called 'ivRender.C' and then type:

```
CC -O -o ivRender ivRender.C -lInventor -lGLU -lGL -lX11
```

```
//
// ivRender
//
// A simple program for measuring scene graph performance.
//
// To compile:
// CC -o ivRender ivRender.C -lInventor -lGL -lX11
// Run with no arguments for usage message.
//

#include <GL/glx.h>
#include <GL/gl.h>
#include <getopt.h>
#include <stdio.h>
#include <unistd.h>

#include <Inventor/SoDB.h>
#include <Inventor/SoInteraction.h>
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/actions/SoGetBoundingBoxAction.h>
#include <Inventor/actions/SoSearchAction.h>
#include <Inventor/misc/SoChildList.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoShape.h>
#include <Inventor/nodes/SoTransform.h>

static const int default_winsize[2] = { 640, 480 };

// Number of frames between timing calculation
#define NUM_FRAMES_PER_CALC30

// Number of frames to run in batch mode
#define NUM_BATCH_FRAMES60

struct Options {
 SbBooladdNodes;
 SbBoolapply;
 SbBoolbatch;
 SbBoolclear;
 SbBoolcountTris;
```

```
 SbBoolrenderCaching;
 SbBoolrenderCulling;
 SbBoolsingleBuffer;
 int numFrames;
 int winsize[2];
 const char*inputFileName;
 const char*overrideFileName;
};

////////////////////////////////////////////////////////////////
/////////
//
// Description:
// Callback used to count triangles.
//

static void
countTriangleCB(void *userData, SoCallbackAction *,
     const SoPrimitiveVertex *,
     const SoPrimitiveVertex *,
     const SoPrimitiveVertex *)
//
////////////////////////////////////////////////////////////////
/////////
{
 long *curCount = (long *) userData;

 (*curCount)++;
}

////////////////////////////////////////////////////////////////
/////////
//
// Description:
// Counts triangles in given graph using primitive generation,
// returning total.
//

int
countTriangles(SoNode *root)
//
////////////////////////////////////////////////////////////////
/////////
{
 longnumTris = 0;
 SoCallbackActionca;

 ca.addTriangleCallback(SoShape::getClassTypeId(),
      countTriangleCB,
      (void *) &numTris);
 ca.apply(root);

 return numTris;
}

////////////////////////////////////////////////////////////////
/////////
//
// Description:
// Counts number of nodes in given graph, returning total.
// Recursive.
```

```
//

int
countNodes(SoNode *root)
//
////////////////////////////////////////////////////////////////
/////////
{
 // This is a little bit evil-- we use the SoINTERNAL public
 // getChildren() method so we count hidden children (in
nodekits
 // or SoFile nodes) also:
 const SoChildList *kids = root->getChildren();

 if (kids == NULL) // No children
     return 1;

 // Total starts at 1 to count myself:
 int total = 1;

 for (int i = 0; i < kids->getLength(); i++)
     total += countNodes((*kids)[i]);

 return total;
}

///////////////////////////////////////////////////////////////
//
// Description:
// Parses command line arguments, setting options.
//

static SbBool
parseArgs(int argc, char *argv[], Options &options)
//
///////////////////////////////////////////////////////////////
{
 SbBoolok = TRUE;
 int c, curArg;

 // Initialize options
 options.addNodes= TRUE;
 options.apply= TRUE;
 options.batch= TRUE;
 options.clear= TRUE;
 options.countTris= TRUE;
 options.renderCaching= TRUE;
 options.renderCulling= TRUE;
 options.singleBuffer= TRUE;
 options.numFrames= -1;
 options.inputFileName= NULL;
 options.overrideFileName= NULL;
 options.winsize[0]= default_winsize[0];
 options.winsize[1]= default_winsize[1];

 while ((c = getopt(argc, argv, "abcf:knrtw:12")) != -1) {
     switch (c) {
      case 'a':
      options.apply = FALSE;
      break;
      case 'b':
```

```
     options.batch = FALSE;
     break;
     case 'c':
     options.clear = FALSE;
     break;
     case 'f':
     options.numFrames = atoi(optarg);
     break;
     case 'k':
     options.renderCulling = FALSE;
     break;
     case 'n':
     options.addNodes = FALSE;
     break;
     case 'r':
     options.renderCaching = FALSE;
     break;
     case 't':
     options.countTris = FALSE;
     break;
     case 'w':
     sscanf(optarg, "%d,%d",
     &options.winsize[0], &options.winsize[1]);
     break;
     case '1':
     options.singleBuffer = TRUE;
     break;
     case '2':
     options.singleBuffer = FALSE;
     break;
     default:
     ok = FALSE;
     break;
    }
}
if (options.batch && options.numFrames < 0)
    options.numFrames = NUM_BATCH_FRAMES;

curArg = optind;

// Check for input filename at end
if (curArg < argc)
    options.inputFileName = argv[curArg++];

// Check for override filename at end
if (curArg < argc)
    options.overrideFileName = argv[curArg++];

// Not enough or extra arguments? Error!
if (options.inputFileName == NULL || curArg < argc)
    ok = FALSE;

// Report options and file names
if (ok) {
    printf("Reading graph from %s\n",
     options.inputFileName);
    if (options.overrideFileName != NULL)
     printf("Reading override graph from %s\n",
     options.overrideFileName);
    else
     printf("No override graph\n");
```

```
    if (options.batch)
     printf("Batch mode is ON (%d frames)\n",
     options.numFrames);
    else
     printf("Batch mode is OFF\n");
    printf("Clear mode is %s\n",
     options.clear ? "ON" : "OFF");
    printf("Render caching is %s\n",
     options.renderCaching ? "ON" : "OFF");
    printf("Render culling is %s\n",
     options.renderCulling ? "ON" : "OFF");
    printf("Using %s\n", options.singleBuffer ?
     "1 buffer" : "2 buffers");
    printf("\n");
 }

 return ok;
}

//////////////////////////////////////////////////////////////
//
// Description:
// Callback used by openWindow().
//

static Bool
waitForNotify(Display *, XEvent *e, char *arg)
//
//////////////////////////////////////////////////////////////
{
 return (e->type == MapNotify) &&
     (e->xmap.window == (Window) arg);
}

//////////////////////////////////////////////////////////////
//
// Description:
// Creates and initializes GL/X window.
//

static void
openWindow(Display *&display, Window &window,
     SbBool singleBuffer, const int winsize[2])
//
//////////////////////////////////////////////////////////////
{
 XVisualInfo*vi;
 Colormapcmap;
 XSetWindowAttributesswa;
 GLXContextcx;
 XEventevent;
 static intattributeList[] = {
    GLX_RGBA,
    GLX_RED_SIZE, 1,
    GLX_GREEN_SIZE, 1,
    GLX_BLUE_SIZE, 1,
    GLX_DEPTH_SIZE, 1,
    None,// May be replaced w/GLX_DOUBLEBUFFER
    None,
 };
```

```
 if (! singleBuffer)
     attributeList[9] = GLX_DOUBLEBUFFER;

 display = XOpenDisplay(0);
 vi = glXChooseVisual(display,
      DefaultScreen(display),
      attributeList);
 cx = glXCreateContext(display, vi, 0, GL_TRUE);
 cmap = XCreateColormap(display,
      RootWindow(display, vi->screen),
      vi->visual, AllocNone);
 swa.colormap= cmap;
 swa.border_pixel= 0;
 swa.event_mask= StructureNotifyMask;
 window = XCreateWindow(display,
      RootWindow(display, vi->screen),
      10, 10, winsize[0], winsize[1],
      0, vi->depth, InputOutput, vi->visual,
      (CWBorderPixel | CWColormap | CWEventMask), &swa);

 // Make the window appear in the lower left corner
 XSizeHints *xsh = XAllocSizeHints();
 xsh->flags = USPosition;
 XSetWMNormalHints(display, window, xsh);
 XFree(xsh);

 XMapWindow(display, window);
 XIfEvent(display, &event, waitForNotify, (char *) window);
 glXMakeCurrent(display, window, cx);
}

//////////////////////////////////////////////////////////////
//
// Description:
// Turns on override flag for all non-group nodes
// under given node.
//

static void
turnOnOverride(SoNode *root)
//
//////////////////////////////////////////////////////////////
{
 if (root->isOfType(SoGroup::getClassTypeId())) {
     SoGroup*group = (SoGroup *) root;
     inti;

     for (i = 0; i < group->getNumChildren(); i++)
      turnOnOverride(group->getChild(i));
 }

 else
     root->setOverride(TRUE);
}

//////////////////////////////////////////////////////////////
//
// Description:
// Creates and returns scene graph containing given
// scene. Adds a perspective camera, a directional
// light, and a transform, which is returned in
```

```
// "sceneTransform". If the overrideInput is not
// NULL, it adds that graph with all override flags
// turned on (on non-groups) right after the
// transform. Then it adds the graph from
// sceneInput. Returns NULL on error.
//

static SoSeparator *
setUpGraph(Options &options,
      const SbViewportRegion &vpReg,
      SoInput *sceneInput,
      SoInput *overrideInput,
      SoTransform *&sceneTransform)
//
//////////////////////////////////////////////////////////////
{
 SoSeparator*root, *inputRoot, *overrideRoot;
 SoPerspectiveCamera*camera;
 SoDirectionalLight*light;
 int i;

 // Create a root separator to hold everything. Turn
 // caching off, since the transformation will blow
 // it anyway.
 root = new SoSeparator;
 root->ref();
 root->renderCaching = SoSeparator::OFF;

 if (options.addNodes) {
     // Add a camera and directional light
     camera = new SoPerspectiveCamera;
     light = new SoDirectionalLight;
     root->addChild(camera);
     root->addChild(light);

     // Add a transform node to spin the scene
     sceneTransform = new SoTransform;
     root->addChild(sceneTransform);
 }

 // Read and add override scene graph if requested
 if (overrideInput != NULL) {
     overrideRoot = SoDB::readAll(overrideInput);
     if (overrideRoot == NULL) {
      fprintf(stderr, "Problem reading override data\n");
      root->unref();
      return NULL;
     }
     turnOnOverride(overrideRoot);

     // Add all children of override separator,
     // since we don't want them separated
     for (i = 0; i < overrideRoot->getNumChildren(); i++)
      root->addChild(overrideRoot->getChild(i));

     overrideRoot->unref();
 }

 // Read and add input scene graph
 inputRoot = SoDB::readAll(sceneInput);
 if (inputRoot == NULL) {
```

```
    fprintf(stderr, "Problem reading data\n");
    root->unref();
    return NULL;
}
root->addChild(inputRoot);

if (options.addNodes) {
    camera->viewAll(root, vpReg);

    // Make the center of rotation the center of
    // the scene
    SoGetBoundingBoxActionbba(vpReg);
    bba.apply(root);
    sceneTransform->center =
     bba.getBoundingBox().getCenter();
}

return root;
}

///////////////////////////////////////////////////////////////
//
// Description:
// Turns off culling on all separators in given
// graph. Recursive.
//

static void
turnOffCulling(SoNode *root)
//
///////////////////////////////////////////////////////////////
{
if (root->isOfType(SoGroup::getClassTypeId())) {
    if (root->isOfType(SoSeparator::getClassTypeId()))
     ((SoSeparator *) root)->renderCulling =
    SoSeparator::OFF;

    SoGroup*group = (SoGroup *) root;
    inti;

    for (i = 0; i < group->getNumChildren(); i++)
     turnOffCulling(group->getChild(i));
}
}

///////////////////////////////////////////////////////////////
//
// Description:
// Prints usage message.
//

static void
printUsage(const char *progName)
//
///////////////////////////////////////////////////////////////
{
fprintf(stderr,
     "Usage: %s [-abcknrt12] scenefile.iv [overridefile.iv]\n",
     progName);
fprintf(stderr,
    "\t-a do not apply render action\n"
```

```
    "\t-b do not run in batch mode\n"
    "\t-c do not clear between frames\n"
    "\t-f N render N frames if in batch mode (default 60)\n"
    "\t-k turn all GLRender kulling off\n"
    "\t-n do not add camera, light, and transform nodes\n"
    "\t-r turn all GLRender caching off\n"
    "\t-t do not count triangles in input data\n"
    "\t-w W,H use a window W by H pixels big (default
640,480)\n"
    "\t-2 run in doublebuffer mode (default singlebuffer)\n");
}

//////////////////////////////////////////////////////////////
//
// Description:
// Mainline
//

main(int argc, char **argv)
//
//////////////////////////////////////////////////////////////
{
 Optionsoptions;
 SoSeparator*root;
 SoTransform*sceneTransform;
 Display*display;
 Windowwindow;
 int frameIndex, numNodes, numTris;
 SbTimetimeDiff, startTime;
 floatframes;

 // Init Inventor
 SoInteraction::init();

 // Parse arguments
 if (! parseArgs(argc, argv, options)) {
    printUsage(argv[0]);
    return 1;
 }

 // Open scene graphs
 SoInputsceneInput, overrideInput;
 if (! sceneInput.openFile(options.inputFileName)) {
    fprintf(stderr,
    "Cannot open %s\n", options.inputFileName);
    return 1;
 }
 if (options.overrideFileName != NULL &&
    ! overrideInput.openFile(options.overrideFileName)) {
    fprintf(stderr,
    "Cannot open %s\n", options.overrideFileName);
    return 1;
 }

 if (! options.renderCaching)
    SoSeparator::setNumRenderCaches(0);

 SbViewportRegion vpr(options.winsize[0], options.winsize[1]);

 root = setUpGraph(options, vpr,
     &sceneInput,
```

```
        (options.overrideFileName == NULL ? NULL :
        &overrideInput),
        sceneTransform);

    if (! options.renderCulling)
        turnOffCulling(root);

    // Count nodes and triangles
    numNodes = countNodes(root);
    if (options.countTris)
        numTris = countTriangles(root);
    else
        numTris = 0;
    printf("%d nodes, approximately %d triangles\n",
        numNodes, numTris);

    // Create and initialize window
    openWindow(display, window, options.singleBuffer,
        options.winsize);

    // Render
    SoGLRenderAction ra(vpr);

    glEnable(GL_DEPTH_TEST);
    glClearColor(.5,.5,.5,1);

    startTime = SbTime::getTimeOfDay();

    int numFramesRendered = 0;

    for (frameIndex = 0; ; frameIndex++) {

        // Rotate the world
        if (options.addNodes)
         sceneTransform->rotation.setValue(
        SbVec3f(1, .5, 0), frameIndex * M_PI / 60);

        if (options.clear || frameIndex == 0)
         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        if (options.apply)
         ra.apply(root);

        if (! options.singleBuffer)
         glXSwapBuffers(display, window);

        ++numFramesRendered;

        if (! options.batch &&
         (numFramesRendered % NUM_FRAMES_PER_CALC == 0)) {

         // Don't really need to do a glFinish() here; since
         // we're measuring performance over and over, any
         // pipeline latency missed in one timing will be
         // accounted for in the next timing, and everything
         // will average out OK.
         glFlush();

         timeDiff = SbTime::getTimeOfDay() - startTime;
         frames = numFramesRendered / timeDiff.getValue();
         printf("\t%10.5f seconds/frame "
```

```
        "(%10.5f frames/sec, %10.5f tris/sec, "
        "%10.5f nodes/sec)\n",
        1.0/frames, frames, frames * numTris,
        frames * numNodes);

        // Start clock over...
        startTime = SbTime::getTimeOfDay();
        numFramesRendered = 0;
       }
       else if (options.batch &&
        (numFramesRendered == options.numFrames)) {

        // Must do a glFinish() here to wait for pipeline to
        // empty.
        glFinish();

        timeDiff = SbTime::getTimeOfDay() - startTime;
        frames = numFramesRendered / timeDiff.getValue();
        printf("\t%10.5f seconds/frame "
        "(%10.5f frames/sec, %10.5f tris/sec, "
        "%10.5f nodes/sec)\n",
        1.0/frames, frames, frames * numTris,
        frames * numNodes);
        break;
       }
   }

   root->unref();

   return 0;
  }
```

Appendix 2 LODD Source

This is the header file and source code for a low-cost level of detail node which you can use in your programs instead of the standard `SoLevelOfDetail` node. This code is also available via anonymous ftp from ftp.sgi.com in the file `sgi/inventor/LODD.C`.

```
/*
 LODD : faster, dumber level of detail. Based on code for
 SoLevelOfDetail. This version just switches based on how
 near/far the center of the LODD's children's bounding box is
 from the eyepoint.

 The easiest way to use this code is to save it into a file
 called LODD.C, compile that file into a DSO, set the
 LD_LIBRARY_PATH environment variable so that Inventor can find
 the class, and then just read in files containing LODD nodes.
 The commands to do all this look like this:

 CC -O -shared -o LODD.so LODD.C
 setenv LD_LIBRARY_PATH .
 ivview -q LODD.iv

 Here's a test LODD.iv you can use to make sure things are
 working:

#Inventor V2.0 ascii
Separator {
 LODD {
     distance [ 5, 8, 12 ]
 Cube { } # See a cube when close to eye...
 Sphere { } # Sphere when a little farther...
 Cone { } # Sphere when a little farther...
 Cylinder { } # Cylinder when farthest
 }
}

 Author(s): Dave Immel, Gavin Bell
 */

#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/actions/SoGetBoundingBoxAction.h>
#include <Inventor/actions/SoGetMatrixAction.h>
#include <Inventor/actions/SoRayPickAction.h>
#include <Inventor/actions/SoSearchAction.h>
#include <Inventor/errors/SoDebugError.h>
#include <Inventor/elements/SoModelMatrixElement.h>
#include <Inventor/elements/SoViewingMatrixElement.h>
#include <Inventor/elements/SoViewportRegionElement.h>
#include <Inventor/elements/SoViewVolumeElement.h>
#include <Inventor/misc/SoChildList.h>
#include <Inventor/misc/SoState.h>

// LODD.h file; inserted here to make life easier...
```

```
#include <Inventor/fields/SoMFFloat.h>
#include <Inventor/nodes/SoGroup.h>
class SoState;

//////////////////////////////////////////////////////////////
//
//
// Class: LODD
//
// LevelOfDetailDistance. This is a faster, simpler version of
// the SoLevelOfDetail node that transforms (0,0,0) in object
// space into world space and figures out how far away that
// point is away from the eye.
//
// If there are N children, this node's distance field should
// contain N-1 distances, with the closest distance first.
//
//////////////////////////////////////////////////////////////
//

class LODD : public SoGroup {

 SO_NODE_HEADER(LODD);

 public:
 // Fields
 SoMFFloatdistance; // Distances to use for comparison
 // Default constructor
 LODD();

 SoEXTENDER public:
 // Implement actions:
 virtual voiddoAction(SoAction *action);
 virtual voidcallback(SoCallbackAction *action);
 virtual voidgetBoundingBox(SoGetBoundingBoxAction
      *action);
 virtual voidgetMatrix(SoGetMatrixAction *action);
 virtual voidGLRender(SoGLRenderAction *action);
 virtual voidrayPick(SoRayPickAction *action);
 virtual voidsearch(SoSearchAction *action);

 SoINTERNAL public:
 static voidinitClass();

 protected:
 // Destructor
 virtual ~LODD();

 // Called by doAction
 int whichToTraverse(SoState *state);

 SbVec3f objOrigin;

 private:
 SbBool firstTime;
 int earlyRenderCount;
};

#ifndef DEBUG
#define NDEBUG
```

```
#endif
#include <assert.h>

SO_NODE_SOURCE(LODD);

////////////////////////////////////////////////////////
//
// Description:
// Init LODD class
//

void
LODD::initClass()

//
////////////////////////////////////////////////////////
{
 SO_NODE_INIT_CLASS(LODD, SoGroup, "Group");
}

////////////////////////////////////////////////////////
//
// Description:
// Constructor
//
// Use: public

LODD::LODD()
//
////////////////////////////////////////////////////////
{
 SO_NODE_CONSTRUCTOR(LODD);
 SO_NODE_ADD_FIELD(distance, (0));
 firstTime = TRUE;
}

////////////////////////////////////////////////////////
//
// Description:
// Destructor
//
// Use: private

LODD::~LODD()
//
////////////////////////////////////////////////////////
{
}

////////////////////////////////////////////////////////
//
// Description:
// Determine which child to traverse based on distance from
camera
//
int
LODD::whichToTraverse(SoState *state)
//
////////////////////////////////////////////////////////
{
 int numKids = getNumChildren();
```

```
    int numDistances = distance.getNum();

    // If no children or 1 child, decision is easy
    if (numKids == 0)
        return -1;

    if (numKids == 1 || numDistances == 0)
        return 0;

    // We cannot assume the origin of our children is (0,0,0).
    // So, we take the bounding box center. This code assumes
    // that the bounding box isn't changing (or if it is, that
    // the initial center will be OK), so it computes the bbox
    // just once for better performance:
    if (firstTime) {
        firstTime = FALSE;

        SoGetBoundingBoxAction
         bba(SoViewportRegionElement::get(state));
        bba.apply(this);
        objOrigin = bba.getCenter();
    }

    // Transformed the object origin into world space.
    const SbMatrix &modelMtx = SoModelMatrixElement::get(state);
    SbVec3f worldPt;
    modelMtx.multVecMatrix(objOrigin, worldPt);

    // And find out where the eye is in world space:
    SbVec3f eyePt =
        SoViewVolumeElement::get(state).getProjectionPoint();

    // Figure out distance:
    // Eliminate the need for sqrt - compare the squared values
    //float d = (worldPt - eyePt).length();
    SbVec3f dvec = worldPt - eyePt;
    float d = dvec.dot(dvec);

    // Figure out how close we are...
    for (int i = 0; i < numDistances; i++) {
        // Eliminate need for sqrt - compare squared values
        if (d < distance[i]*distance[i])
         break;
    }

    // Make sure we didn't go off the deep end
    if (i >= numKids)
        i = numKids - 1;

    return i;
}

////////////////////////////////////////////////////////////////
/
//
// Description:
// Implements typical traversal, determining child to traverse
// based on distance from camera
//
// Use: extender
```

```
void
LODD::doAction(SoAction *action)
//
/////////////////////////////////////////////////////////////////
{
 int childToTraverse;
 SoState *state = action->getState();

 childToTraverse = whichToTraverse(state);

 // Traverse just the one kid
 if (childToTraverse >= 0) {
     children->traverse(action, childToTraverse,
      childToTraverse);
 }
}

////////////////////////////////////////////////////////////////
//
// Description:
// Implements callback action for LODD nodes.
//
// Use: extender

void
LODD::callback(SoCallbackAction *action)
//
////////////////////////////////////////////////////////////////
{
 doAction(action);
}

////////////////////////////////////////////////////////////////
//
// Description:
// Traversal for computing bounding box. Computes bbox of ALL
// kids.
//
// Use: extender

void
LODD::getBoundingBox(SoGetBoundingBoxAction *action)
//
/////////////////////////////////////////////////////////////////
//
{
 SoGroup::getBoundingBox(action);
}

/////////////////////////////////////////////////////////////////
//
//
// Description:
// Implements getMatrix action.
//
// Use: extender

void
LODD::getMatrix(SoGetMatrixAction *action)
//
```

```
////////////////////////////////////////////////////////////////
//
{
 int numIndices;
 const int*indices;

 // Only need to compute matrix if this node is in the middle
 // of the current path chain. We don't need to push or pop
 // the state, since this shouldn't have any effect on other
 // nodes being traversed.

 if (action->getPathCode(numIndices, indices) ==
     SoAction::IN_PATH)
     children->traverse(action, 0, indices[numIndices - 1]);
}

////////////////////////////////////////////////////////////////
//
// Description:
// Traversal for rendering. This uses the screen distance
// comparison.
//
// Use: extender

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

////////////////////////////////////////////////////////////////
//
// Description:
// Implements ray picking.
//
// Use: extender

void
LODD::rayPick(SoRayPickAction *action)
//
////////////////////////////////////////////////////////////////
{
 doAction(action);
}

////////////////////////////////////////////////////////////////
//
// Description:
// Implements search action for LODD nodes. This determines if
// the LODD should be searched. If so, this calls the search
// method for SoGroup to do the work.
//
// Use: extender

void
LODD::search(SoSearchAction *action)
//
////////////////////////////////////////////////////////////////
//
```

```
{
 SbBooldoSearch = TRUE;

 // See if we're supposed to search only if the stuff under
 // the LODD is relevant to the search path (stolen from
 // SoSeparator)

 if (! action->isSearchingAll()) {
     intnumIndices;
     const int*indices;

     // Search through this LODD node only if not searching
     // along a path or this node is on the path
     if (action->getPathCode(numIndices, indices) ==
      SoAction::OFF_PATH)
      doSearch = FALSE;
 }

 if (doSearch) {
     SoGroup::search(action);
 }
}
```