Chapter 14

System-Specific Tuning

This chapter provides tuning information that's relevant for particular Silicon Graphics systems. Use these techniques as needed if you expect your program to be used primarily on one kind of system, or a group of systems. The chapter discusses:

- "Optimizing Performance on Low-End Graphics Systems" on page 267
- "Optimizing Performance on Mid-Range Systems" on page 274
- "Optimizing Performance on Indigo2 IMPACT Systems" on page 276
- "Optimizing Performance on RealityEngine Systems" on page 282

Some points are also discussed in earlier chapters but repeated here because they result in particularly noticeable performance improvement on certain platforms.

Note: To determine your particular hardware configuration, execute */usr/gfx/gfxinfo*. See the reference page for *gfxinfo* for more information. You can also call **glGetString()** with a GL_RENDERER argument. See the reference page for information about the renderer strings for different systems.

Optimizing Performance on Low-End Graphics Systems

This section discusses how you can get the best results from your application on low-end graphics systems, such as the Indy™, Indigo™ Entry, and Indigo^{2}™ XL systems (but not other Indigo^{2} systems); discussing the following topics:

- "Choosing Features for Optimum Performance"
- "Using the Pipeline Effectively"
- "Using Geometry Operations Effectively"
- "Using Per-Fragment Operations Effectively"

Choosing Features for Optimum Performance

By emphasizing features implemented in hardware, you can significantly influence the performance of your application. As a rule of thumb, consider the following:

- **Hardware-supported features**: Lines, filled rectangles, color shading, alpha blending, alpha function, antialiased lines (color-indexed and RGB), line and polygon patterns, color plane masks, color dithering, logical operations selected with **glLogicOp()**, pixel reads and writes, screen to screen copy, and scissoring.
- **Software-supported features**: All features not in hardware, such as stencil and accumulation buffer, fogging and depth queuing, transforms, lighting, clipping, depth buffering, and texturing. Triangles and polygons are partially software supported.

Using the Pipeline Effectively

The low-end graphics systems' FIFO allows the CPU and the graphics subsystem to work in parallel. For optimum performance, follow these guidelines:

- Make sure the graphics subsystem always has enough in the queue.
- Let the CPU perform preprocessing or non-graphic aspects of the application while the graphics hardware works on the commands in the FIFO.

  For example, a full screen clear takes about 3 ms. Let the application do something else immediately after a clear, the FIFO otherwise fills up and forces a stall.

Note that FIFOs in low-end systems are much smaller than those in high-end systems. Not all graphics processing happens in hardware, and the time spent therefore differs greatly. To detect imbalances between the CPU and the graphics FIFO, execute the *gr_osview* command and observe gfxf in the CPU bar and fiwt and finowt in the gfx bar.

- gfxf: time spent waiting for the graphics FIFO to drain.
- fiwt: FIFO filled up and host went to sleep waiting for it to drain.
- finowt: FIFO filled up but drained fast enough that host continued.

Using Geometry Operations Effectively

If your application seems transform limited, you can improve it by considering the tips in this section. The section starts with some general points, then discusses optimizing line drawing and using triangles and polygons effectively.

To improve performance in the geometry subsystem, follow these guidelines:

- Use single-precision floating point parameters for vertices, normals, and colors.
- Transform paths use single-precision floats—it's fastest to use **glVertex3fv()** and **glVertex2fv()**.
- Use **glOrtho()** and a modelview matrix without rotation for best performance.
- Perspective transforms that require multiplication by 1/W or division by W are much slower.
- To minimize the time used for floating point conversion, use floating point coordinates where possible, except where memory size is critical.
- Don't enable normalizing of normals if the modelview matrix doesn't include scaling and if you have unit-length normals.

Optimizing Line-Drawing

Even on low-end systems, lines can provide real-time interactivity. Consider these guidelines:

- Use line drawing while the scene is changing and solid rendering when the scene becomes static.
- Shaded lines and antialiased lines that are one pixel wide are supported by the hardware. Patterned lines are as fast as solid lines.
- Wide lines are drawn as multiple parallel offset lines.
- Depth-queued lines are about as fast as shaded lines.
- The hardware can usually draw lines faster than the software can produce commands, though long or antialiased lines can cause a backup in the graphics pipeline.
- Avoid depth buffering for lines; incorrect depth-sorting artifacts are usually not noticeable.

Optimizing Triangles and Polygons

When rendering triangles and polygons, keep in mind the following:

- Maximize the number of vertices between **glBegin()** and **glEnd().**
- Decompose quads and polygons into triangle strips. The GL_TRIANGLE_STRIP primitive has the fastest path.
- Use connected primitives (triangle, quad, and line strips). Use triangle strips wherever possible and draw as many triangles as possible per **glBegin()/glEnd()** pair.
- When rendering solid triangles, consider the following:
  - Color shading and alpha blending are performed in hardware on Indy and Indigo^2 XL systems. Consult system-specific documentation for information on other low-end systems.
  - Larger triangles have a better overall fill rate than smaller ones because CPU setup per triangle is independent of triangle size.

Using Per-Fragment Operations Effectively

This section looks at some things you can do if your application is fill limited. It provides information about getting the optimum fill rates and about using pixel operations effectively.

Getting the Optimum Fill Rates

To achieve the fastest fill rates possible, consider the following:

- Clear operations and screen-aligned **glRect()** calls that don't use the depth or stencil buffer have a maximum fill rate of 400 MB/sec

  The hardware accelerates drawing rectangles that have their vertical and horizontal edges parallel to those of the window. The OpenGL **glRect()** call—as opposed to IRIS GL **rect()**—specifies rectangles, but depending on the matrix transformations they may not be screen-aligned. If the matrices are such that the rectangle drawn by **glRect()** is screen-aligned, OpenGL detects that and uses the accelerated mode for screen-aligned rectangles.
- Use **glShadeModel()** with GL_FLAT whenever possible, especially for lines.

- Using dithering, shading, patterns, logical operations, writemasks, stencil buffering, and depth buffering (and alpha blending on some systems) slows down an application.
- Use **glEnable()** with GL_CULL_FACE to eliminate backfacing polygons, especially in modes that have slow fill rates, such as depth buffering and texturing (alpha-blending on some systems).
- In any OpenGL matrix mode, low-end systems check for transforms that only scale, and have no rotations or perspective. The system looks at the specified matrices, and if they only scale and have no rotation or perspective, performs optimizations that speed up transformation of vertices to device coordinates. One way to specify this is:

  ```
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0,xsize,0,ysize);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glShadeModel(GL_FLAT);
  ```

  You also have to use a **glVertex2fv()** call to specify 2D vertices.
- Starting with IRIX 6.2, texture mapping speed is increased by about 10 times (compared to previous releases) when texture parameters are specified as follows:

  ```
  glEnable(GL_TEXTURE_2D);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST);
  ```

  In addition, follow these guidelines:

  – For RGB textures, make sure the texture environment mode, set with **glTexEnv(),** is either GL_REPLACE_EXT or GL_DECAL.

  – For RGBA textures, make sure the texture environment mode is GL_REPLACE_EXT.

    Note that the above fast path does not work when stencil is enabled and when depth buffering and alpha testing are both on.

Note: Avoid using depth buffering whenever possible, because the fill rates for depth buffering are much slower. In particular, avoid using depth-buffered lines, as they are especially slow. The depth buffer is as large as the window, so use the smallest possible window to minimize the amount of memory allocated to the depth buffer. The same applies for stencil buffers.

Using Per-Fragment Operations Effectively

Write your OpenGL program to use the combinations of pixel formats and types listed in Table 14-1, for which the hardware can use DMA. The CPU has to reformat pixels in other format and type combinations.

Table 14-1 Pixel Formats and Types Using DMA on Low-End Systems

Format	Type
GL_RGBA	GL_UNSIGNED_BYTE
GL_ABGR_EXT	GL_UNSIGNED_BYTE
GL_COLOR_INDEX	GL_UNSIGNED_BYTE
GL_COLOR_INDEX	GL_UNSIGNED_SHORT

Note that GL_ABGR_EXT provides better performance than GL_RGBA on Indigo Entry systems but not on Indy or Indigo^2 XL where the two perform about the same.

Here are some additional guidelines for optimizing pixel operations:

- **Scrolling.** When scrolling scenes, use **glCopyPixels()** to copy from one scene to the next.

  When you scroll something, such as a teleprompter text scroll or an EKG display, use **glCopyPixels()** to shift the part of the window in the scrolling direction, and draw only the area that's newly exposed.

  This is much faster than completely redrawing each frame.

- **Minimizing calls.** Make each pixel operation draw as much data as possible. For each call, a certain amount of setup is required; you cut down on that time if you minimize the number of calls.

- **Zooming.** Zoomed pixels can't use DMA. A 2 x 2 zoom is faster than other zoom operations.

- **Depth and scissoring.** Low-end systems use an accelerated algorithm that makes clearing the depth buffer virtually free. However, this has slowed enabling and disabling scissoring and changing the scissor rectangle. The larger the scissor rectangle, the longer the delay. Note that rendering while scissoring is turned on is fast; calling **glEnable()** and **glDisable()** with GL_SCISSOR and calling **glScissor()** is slow. This also includes pushing and popping attributes that may cause a scissor change.

Low-End Specific Extensions

For Indy and Indigo^2 XL systems, an extension has been developed that increases fill rate by drawing pixels as *N* x *N* rectangles (effectively lowering the window resolution). This "framezoom" extension, SGIX_framezoom, is available as a separate patch under both IRIX 5.3 and IRIX 6.2.

Caution: This extension is experimental. The interface and supported systems may change in the future.

When using the extension, consider the following performance tips:

- The extension works best when texturing is enabled. When pixels are zoomed up by *N*, you can expect the fill rate to go up by about $N^2/2$. This number is an estimate; a speed-up of this magnitude occurs only if texturing performance has been optimized as explained in the last bullet of "Getting the Optimum Fill Rates" on page 270.
- When texturing is not enabled, performance, although faster than the texture map case, is relatively slow compared to the non-framezoomed case. Actually, a framezoom value of 2 is slower than if framezoom was not enabled. The reason is that the graphics hardware in low-end systems is optimized for flat or interpolated spans, and not for cases where the color changes from pixel to pixel (as with texturing). When pixels are bigger (as with the framezoom extension), this benefit cannot be used.
- The framezoom factor can be changed on a frame-to-frame basis, so you can render with framezoom set to a larger value when you're moving around a scene, and lower the value, or turn framezoom off, when there are no changes in the scene.

For more detailed information, see the reference page for **glFrameZoomSGIX()** for those systems that have the patch installed.

Optimizing Performance on Mid-Range Systems

This section discusses optimizing performance for two of the Silicon Graphics mid-range systems: Elan™ graphics and Extreme™ graphics. For information on Indigo2 IMPACT systems, see “Optimizing Performance on Indigo2 IMPACT Systems” on page 276.

General Performance Tips

The following general performance tips apply to mid-range graphics systems:

- **Data size.** Mid-range graphics systems are optimized for word-sized and word-aligned data (one word is four bytes). Pixel read and draw operations are fast if the data is word aligned and each row is an integral number of words.
- **Extensions.** The following extensions are hardware accelerated:
  – “The Logical Operation Blending Extension” on page 143
  – “The Polygon Offset Extension” on page 145
  – “The ABGR Extension” on page 125

  Other available extensions are implemented in software.
- **Flushing.** Too many flushes, implicit or explicit, can adversely affect performance:
  – In single buffer mode, you may need to call **glFlush()** after the last of a series of primitives to force the primitives through the pipeline and expedite graphics processing (explicit flushing).
  – In double buffer mode, it is not necessary to call **glFlush()**; the **glXSwapBuffers()** call automatically flushes the pipeline (implicit flushing).

Optimizing Geometry Operations on Mid-Range Systems

Consider the following points when optimizing geometry operations for a mid-range system:

- **Antialiasing.** Mid-range graphics systems support hardware-accelerated lines of width 1.
- **Clipping.** Minimize clipping for better performance.

Optimizing Per-Fragment Operations on Mid-Range Systems

Consider the following issues when optimizing per-fragment operations for a mid-range system:

- **Alpha Blending.** Mid-range graphics systems support alpha blending in hardware. All primitives can be blended, with the exception of antialiased lines and points, which use the blending hardware to determine pixel coverage. The alpha value is ignored for these primitives. Pixel blends work best in 24-bit, single-buffered RGB mode. In double-buffered RGB mode, the blend quality degrades.
- **Dithering.** Dithering is used to expand the range of colors that can be created from a group of color components and to provide smooth color transitions. Disabling dither can improve the performance of **glClear()**. Dithering is enabled by default. To change that, call

  ```
  glDisable(GL_DITHER)
  ```
- **Fog.** Mid-range graphics systems do not accelerate per-fragment fog modes. To select a hardware-accelerated fog mode, call

  ```
  glHint (GL_FOG_HINT, GL_FASTEST)
  ```
- **Lighting.** Mid-range graphics systems accelerate all lighting features.
- **Pixel formats.** The GL_ABGR_EXT pixel format is much faster than the GL_RGBA pixel format. For details, see "The ABGR Extension" on page 125.

  The combinations of types and formats shown in Table 14-2 are the fastest.

Table 14-2 Pixel Formats and Types That Are Fast on Mid-Range Systems

Format	Type
GL_RGBA	GL_UNSIGNED_BYTE
GL_ABGR_EXT	GL_UNSIGNED_BYTE
GL_COLOR_INDEX	GL_UNSIGNED_SHORT
GL_COLOR_INDEX	GL_UNSIGNED_BYTE

- **Texture Mapping.** All texture mapping is performed in software. As a result, textured primitives run with reduced performance.

- Elan Graphics™ accelerates depth buffer operations on systems that have depth buffer hardware installed (default on Elan, optional on XS and XS24, not available on Entry systems).
- **Fast Clear Operations.** The hardware performs combined color and depth clear under the following conditions:
  - depth buffer is cleared to 1 and the depth test is GL_LEQUAL
  - depth buffer is cleared to 0 and the depth test is GL_GEQUAL

Optimizing Performance on Indigo2 IMPACT Systems

This section provides performance tips for Indigo2 IMPACT graphics systems. All information applies to both Indigo2 High IMPACT and Indigo2 Maximum IMPACT systems, unless specifically stated otherwise. You learn about:

- "General Tips for Performance Improvement"
- "Achieving Peak Geometry Performance"
- "Using Textures"
- "Using Images"
- "Accelerating Color Space Conversion"
- "Using Display Lists Effectively"
- "Offscreen Rendering Capabilities"

General Tips for Performance Improvement

This section provides some general tips for improving overall rendering performance. It also lists some features that are much faster than on previous systems and may now be used by applications that could not consider them before.

- **Fill-rate limited applications.** Because per-primitive operations (transformations, lighting, and so on) are very efficient on Indigo2 IMPACT systems, applications may find that they are fill-rate limited when drawing large polygons (more than 50 pixels per triangle). In that case, you can actually increase the complexity of per-primitive operations at no cost to overall performance. For example, additional lights or two-sided lighting may come for free.

For general advice on improving performance for fill-rate limited applications, see “Tuning the Raster Subsystem” on page 243. Specific for Indigo2 IMPACT systems is that texturing is optimized.

- **Geometry-limited applications.** For applications that draw many small polygons, consider a different approach: Use textures to avoid drawing so many triangles. See “Using Textures” on page 278.
- **Clipping.** For optimum performance, avoid clipping. Special hardware supports clipping within a small range outside of the viewport. By keeping geometry within this range, you may be able to significantly reduce clipping overhead.
- **GLU NURBS.** If you use GLU NURBS, store the tessellation result in display lists to take full advantage of evaluator performance. Don’t for example, recompute tessellations.
- **Antialiasing.** Antialiased lines on Indigo2 IMPACT systems are high quality and fast. Applications that did not use antialiased lines before because of the performance penalty may now be able to take advantage of them. All antialiased lines are rendered with the same high quality, regardless of the settings of GL_LINE_SMOOTH_HINT. Although available, wide antialiased lines are not supported in hardware and should be avoided. Wide antialiased points are supported in hardware with good performance.

  Multisampling is not supported. Antialiasing of polygons is not supported in hardware. You can, however, draw antialiased line loops around polygons to get antialiasing.

Achieving Peak Geometry Performance

Because of the Indigo2 IMPACT hardware setup, rendering of primitives is especially fast if you follow these recommendations:

- **Triangles.** Work with triangle strips consisting of six triangles (or multiples of six). Render individual triangles in groups of four (or multiples or four).

  Note that the hardware allows mixing of different lengths of triangle strips. Grouping like primitives is highly recommended.
- **Quads.** Work with quad strips consisting of three quads (or multiples of three). Render individual quads in sets of three (or multiples of three).

- Use **glLoadIdentity()** to put identity matrixes on the stack. The system can optimize the pipeline if the identity matrix is used, but does not check whether a matrix loaded by **glLoadMatrix()** is the identity matrix.

Using Textures

Indigo2 IMPACT systems render textures quickly: Texturing is only slightly slower than non-textured fill rates and significantly improves image quality for your application. To get the most benefit from textures, use the extensions to OpenGL for texture management as follows:

- Use the texture object extension (EXT_texture_object) to keep as many textures resident in texture memory as possible. The extension lets you bind a texture to a name, then use it as needed (similar to the way you define and call a display list). The extension also allows you to specify a set of textures and prioritize which textures should be resident in texture memory. For more information, see "The Texture Object Extension" on page 93.
- Use the texture-LOD extension to minimize the number of mipmap levels you need to have resident while at the same time having the advantage of several levels of detail. For more information, see "The Texture LOD Extension" on page 121.
- Use the subtexture extension to make texture definition more efficient. For example, assume an application uses several large textures, all of the same size and component type. Instead of declaring multiple textures, declare one, then use **glTexSubImage2DEXT()** to redefine the image as needed. For more information, see "The Subtexture Extension" on page 96.
- Use the GL_RGBA4 internal format to improve performance and conserve memory (see Table 6-1, "Some Internal Formats Supported by the Texture Extension," on page 87).

  This format is especially important if you have a large number of textures. The quality is reduced, but you can fit more textures into memory because they need less space.
- Use the GL_RGBA4 internal format and the packed pixels extension to minimize disk space and improve download rate (see "The Packed Pixels Extension" on page 125).
- Use the 3D texture extension for volume rendering. Note, however, that due to the large amount of data, you typically have to tile the texture. You can set up the

texture as a volume and slice through it as needed. For more information, see "The 3D Texture Extension" on page 100.

- If you use GL_LUMINANCE and GL_LUMINANCE_ALPHA textures, you can speed up loading by using the texture select extension.
- For Indigo2 IMPACT graphics, data coherence enhances performance. If you're drawing texture-mapped points, short lines, or very small triangles, try to draw together the ones that are close together in the image. Don't bounce around, or drawing may slow down.

Using Images

This section provides some tips for using images on Indigo2 IMPACT systems.

On many systems, a program encounters a noticeable performance cliff when a certain specific feature (for example depth-buffering) is turned on, or when the number of modes or components exceeds a certain limit.

On Indigo2 IMPACT systems, performance scales with the number of components. For example, on some systems, a switch from RGBA to RGB may not result in a change in performance, while on Indigo2 IMPACT systems, you should expect a performance improvement of 25%. (Note that while this applies to loading textures, it does not apply to using loaded textures.)

Here are some additional hints for optimizing image processing:

- Instead of **glPixelMaps()**, use the Silicon Graphics color table extension, discussed in "The Color Table Extension" on page 136, especially when working with GL_LUMINANCE or GL_LUMINANCE_ALPHA images.

  OpenGL requires expansion of pixels using formats other than GL_RGBA to GL_RGBA. Conceptually, this expansion takes place before any pixel operation is applied. Indigo2 IMPACT systems attempt to postpone expansion as long as possible: this improves performance (operations must be performed on all components present in an image—a non-expanded image has fewer components and therefore requires less computation). Because pixel maps are inherently four components, GL_LUMINANCE and GL_LUMINANCE_ALPHA images must be expanded (a different lookup table is applied to the red, green, and blue components derived from the luminance value). However, if the internal format of an image matches the internal format of the color table, Indigo2 IMPACT hardware postpones the expansion, which speeds up processing.

- The convolution extension, discussed in “The Convolution Extension” on page 129 has been optimized. If possible, use the extension with separable convolution filters.

  Indigo2 IMPACT systems are tuned for 3 x 3, 5 x 5, and 7 x 7 convolution kernels. If you choose a kernel size not in that set, performance is comparable to that of the closest member of the set. For example, if you specify 2 x 7, performance is similar to using 7 x 7.
- Use texture-based zooming instead of **glPixelZoom()**.

  Texture loading and interpolation is fast on Indigo2 IMPACT, and texture-based zooming therefore results in a speed increase and higher-quality, more controllable results.
- Where possible, minimize color table and histogram sizes and the number of color tables activated. If you don’t, you may experience performance loss because the two compete for limited resources with other OpenGL applications.

Accelerating Color Space Conversion

Indigo2 IMPACT systems provide accelerated color space conversions and device-specific color matching.

- **Linear color space conversion.** Use the color matrix extension to handle linear color space conversion, such as CMY to RGB, in hardware. This extension is also useful for reassigning or duplicating components. See “The Color Matrix Extension” on page 128 for more information.
- **Non-linear color space conversions.** Use the 3D and 4D texture extension for color conversion (for example, RGBA to CMYK). Using the **glPixelTexGenSGIX()** command, you can direct pixels into the lookup table and get other pixels out. Performance has been optimized.

Using Display Lists Effectively

If you work on a CAD application, or other application that uses relatively static data, and therefore find it useful to use display lists instead of immediate mode, you can benefit from the display list implementation on Indigo2 IMPACT systems:

- When the display list is compiled, most OpenGL functions are stored in a format that the hardware can use directly. At execution time, these display list segments are simply copied to the graphics hardware with little CPU overhead.

- A further optimization is that a DMA mechanism can be used for a subset of display lists. By default, the CPU feeds the called list to the graphics hardware. Using DMA display lists, the host gives up control of the bus and Indigo2 IMPACT uses DMA to feed the contents to the graphics pipeline. The speed improvement at the bus is fourfold; however, a setup cost makes this improvement irrelevant for very short lists. The break-even point varies depending on the list you're working with, whether it's embedded in other lists, and other factors.

Display List Compilation on Indigo2 IMPACT Hardware

The functions that are direct (use hardware formats) will change over time. The following items are currently NOT compiled to direct form:

- **glCallLists()** and **glListBase()**
- all imaging functions
- all texture functions, with the exception of **glBindTextureEXT()**
- **glHint()**, **glClear()**, and **glScissor()**
- **glEnable()** and **glDisable()**
- **glPushAttrib()** and **glPopAttrib()**
- all evaluator functions
- most OpenGL extensions

DMA Display Lists on Indigo2 IMPACT Systems

If a display list meets certain criteria, Indigo2 IMPACT systems use DMA to transfer data from the CPU to the graphics pipeline. This is useful if an application is bus limited. It can also be an advantage in a multi-threaded application, because the CPU can do some other work while the graphics subsystem pulls the display list over.

The DMA method is used under the following conditions:

- Only functions that are compiled down to direct form are used.
- There is no hierarchy in the display list that is more than eight levels deep.
- If the display list hierarchy uses texture objects, all textures that are referenced have to fit into hardware texture memory (TRAM) at the same time.

Note that the system tests recursively whether the DMA model is appropriate: If an embedded display list meets the criteria, it can be used in DMA mode even if the higher-level list is processed by the CPU.

Offscreen Rendering Capabilities

Offscreen rendering can be accelerated using the pixel buffer extension discussed in "The Pixel Buffer Extension" on page 179.

Optimizing Performance on RealityEngine Systems

This section provides information on optimizing applications for RealityEngine and RealityEngine2. It discusses:

- "Optimizing Geometry Performance"
- "Optimizing Rasterization"
- "Optimizing Use of the Vertex Array Extension"
- "Optimizing Multisampling and Transparency"
- "Optimizing the Imaging Pipeline"

Optimizing Geometry Performance

Here are some tips for improving RealityEngine geometry performance:

- **Primitive length.** Most systems have a characteristic primitive length that the system is optimized for. On RealityEngine systems, multiples of 3 vertices are preferred, and 12 vertices (for example a triangle strip that consists of 10 triangles) result in the best performance.
- **Fast mode changes.** Changes involving logic op, depth func, alpha func, shade model, cullface, or matrix mode are fast.
- **Slow mode changes**. Changes involving texture binding, lighting and material changes, line width and point size changes, scissor, or viewport are slow.
- **Texture coordinates.** Automatic texture coordinate generation with **glTexGen()** results in a relatively small performance degradation.

- **Quads and polygons.** When rendering quads, use GL_POLYGON instead of GL_QUADS. The GL_QUADS primitive checks for self-intersecting quads and is therefore slower.

Optimizing Rasterization

This section discusses optimizing rasterization. While it points out a few things to watch out for, it also provides information on features that were expensive on other systems but are acceptable on RealityEngine systems:

- After a clear (or a command to fill a large polygon), send primitives to the geometry engine for processing. Geometry can be prepared as the clear or fill operations take place.
- Texturing is free on a RealityEngine if you use a 16-bit texel internal texture format. There are 16-bit texel formats for each number of components. Using a 32-bit texel format yields half the fill rate of the 16-bit texel formats. See "The Texture Extension" on page 85 for more information on internal formats.
- The use of detail texture and sharpen texture usually incurs no additional cost and can greatly improve image quality. Note, however, that texture management can become expensive if a detail texture is applied to many base textures. Use detail texture but keep detail and base paired and detail only a few base textures. See "The Sharpen Texture Extension" on page 107 and "The Detail Texture Extension" on page 112.
- If textures are changing frequently, use the subtexture extension to incrementally load texture data. RealityEngine systems are optimized for 32 x 32 subimages. See "The Subtexture Extension" on page 96.
- There is no penalty for using the highest-quality mipmap filter (GL_LINEAR_MIPMAP_LINEAR) if 16-bit texels are used (for example, the GL)RGBA4_EXT internal format). See "The Texture Extension" on page 85.
- Local lighting or multiple lights are possible without an unacceptable degradation in performance. As you turn on more lights, performance degrades slowly.
- Simultaneous clear of depth and color buffers is optimized in hardware.
- Antialiased lines and points are hardware accelerated.

Optimizing Use of the Vertex Array Extension

If you use the vertex array extension (“The Vertex Array Extension” on page 151), the following cases are currently accelerated for RealityEngine (each line corresponds to a different special case). To get the accelerated routine, you need to make sure your vertices correspond to the given format by using the correct size and type in your enable routines, and also by enabling the proper arrays:

- Vertex2f
- Normal3f Vertex3f
- Color3f Vertex3f
- Color4f Vertex3f
- Normal3f Vertex3f
- TexCoord2f Vertex3f
- Color4f TexCoord2f Vertex3f
- Color3f Normal3f Vertex3f
- Color4f Normal3f Vertex3f
- Normal3f TexCoord2f Vertex3f
- Color4f TexCoord2f Normal3f Vertex3f

Optimizing Multisampling and Transparency

Multisampling provides full-scene antialiasing with performance sufficient for a real-time visual simulation application. However, it isn’t free and it adds to the cost of some fill operations. With RealityEngine graphics, some fragment processing operations (blending, depth buffering, stenciling) are essentially free if you aren’t multisampling, but do reduce performance if you use a multisample-capable visual. Texturing is an example of a fill operation that can be free on a RealityEngine and isn’t affected by the use of multisampling. Note that when using a multisample-capable visual, you pay the cost even if you disable multisampling.

Below are guidelines for optimizing performance for multisampling:

- Multisampling offers an additional performance optimization that helps balance its cost: a virtually free screen clear. Technically, this operation doesn’t really clear the screen, but rather allows you to set the depth values in the framebuffer to be undefined. Therefore, use of this clear requires that every pixel in the window be rendered every frame. This clear is invoked with **glTagSampleBufferSGIX()** (see the reference page for more information).

- When multisampling, using a smaller number of samples and color resolution results in better performance. 8 samples with 8-bit RGB components and a 24-bit depth buffer usually result in good performance and quality. (32-bit depth buffers are rarely needed.)
- Multisampling with stencilling is expensive. If it becomes too expensive, use the polygon offset extension to deal with decal tasks (for example runway strips). See "The Polygon Offset Extension" on page 145.
- There are two ways of achieving transparency on a RealityEngine system: alpha-blending and subpixel screen-door transparency using **glSampleMaskSGIS().** Alpha blending may be slower, since more buffer memory may need to be accessed. For more information about screen-door transparency, see "The Multisample Extension" on page 154.

Optimizing the Imaging Pipeline

Here are some points that help you optimize the imaging pipeline:

- Unsigned color types are faster than signed or float types.
- Smaller component types (for example, GL_UNSIGNED_BYTE) require less bandwidth from the host to the graphics pipeline and are faster than larger types.
- The slow pixel drawing path is used when fragment operations (depth or alpha testing, and so on) are used, or when the format is GL_DEPTH_COMPONENT, or when multisampling is enabled and the visual has a multisample buffer.

Using the Color Matrix and the Color Writemask

Your application might perform RGBA imaging operations (for example convolution, histogram, and such) on a single-component basis. This is the case either when processing gray scale (monochrome) images, or when different color components are processed differently.

RealityEngine systems currently do not support RGBA-capable monochrome visuals (a feature that is introduced by the framebuffer configuration extension; see "The Framebuffer Configuration Extension" on page 171). You must therefore use a four-component RGBA visual even when performing monochrome processing. Even when monochrome RGBA-capable visuals are supported, you may find it beneficial to use four-component visuals in some cases, depending on your application, to avoid the overhead of the **glXMakeCurrent()** or **glXMakeCurrentReadSGI()** call.

On RealityEngine systems, monochrome imaging pipeline operations are about four times as fast as the four-component processing. This is because only a quarter of the data has to be processed or transported either from the host to graphics subsystem—for example, for **glDrawPixels()**—or from the framebuffer to the graphics engines—for example, for **glCopyPixels().**

The RealityEngine implementation detects monochrome processing by examining the color matrix (see "Tuning the Imaging Pipeline" on page 246) and the color writemask.

The following operations are optimized under the set of circumstances listed below:

- **glDrawPixels()** with convolution enabled and
  - the pixel format is GL_LUMINANCE or GL_LUMINANCE_ALPHA
  - the color matrix is such that the active source component is red
- **glCopyPixels()** and the absolute value of GL_ZOOM_X and GL_ZOOM_Y is 1.

Here's the set of circumstances to be met:

- All pixel maps and fragment operations are disabled.
- The color matrix does not scale any of the components.
- The post color matrix scales and biases for all components are 1 and 0, respectively.
- Either write is enabled only for a single component (R, G, B, or A), or alpha-component write is disabled.