OpenGL Performance FAQ for NVIDIA GPUs v2.0

John Spitzer
NVIDIA Corporation
JSpitzer@nvidia.com

This document refers to the performance of OpenGL on the NVIDIA GeForce 256, Quadro, GeForce2 MX and GeForce2 GTS, running the Release 5 (5.XX) series of drivers.

I. Geometry

1. What are the fastest transfer mechanisms for geometry?
2. What are the fastest primitives to use?
3. Which vertex array calls should I use?
4. How does processor speed and/or bus bandwidth (AGP/AGP2X/AGP4X) affect this?
5. What is the best manner in which to organize my geometry in memory?
6. What do vertex arrays buy me in terms of performance?
7. What do compiled vertex arrays (CVAs) buy me in terms of performance?
8. How can I maximize performance with the vertex_array_range extension?
9. Is there an optimal size vertex array to define/use?
10. Should I use display lists for static geometry?
11. Will I get better performance if I chain together separate triangles with degenerate triangle strips?
12. I've heard that NVIDIA GPUs have a vertex cache – how do I use it?

II. Lighting

13. What lighting mode is fastest?
14. Which ones should I avoid?
15. How many lights should I use?
16. Should I turn normalization on or off for maximum performance?
17. Should I use the rescale normal extension to increase performance?
18. Is it faster if I only want to calculate the diffuse component, not the specular?

III. Texture Coordinate Generation

19. Which TexGen modes are hardware accelerated?
20. Is the texture matrix hardware accelerated?
21. Is there hardware acceleration for two sets of texture coordinates?

IV. Clipping and Culling

22. Should I perform any clipping myself?
23. Are user-defined clip planes hardware accelerated?
24. How many user-defined clip planes are hardware accelerated?

V. Texturing

25. How can I maximize texture downloading performance?
26. Should I use texture compression?
27. How can I maximize texture rendering performance?
28. What filtering modes should I use?
29. How much performance will anisotropic filtering take away?
30. What kind of performance increase can I expect from using paletted textures?

VI. Other Fragment Operations

31. What are the performance implications of polygon stippling?
32. What fragment operations should I avoid?

VII. Pixel Transfers

33. What are the best formats/types to use with glReadPixels and glDrawPixels?
34. I want to read back the depth buffer for incremental updates; how should I do this?

VIII. Miscellaneous

35. How much will Full Scene Anti-Aliasing (FSAA) slow me down?
36. Is context switching expensive?
37. What about state changes?
38. Why is my GeForce 256 running at a fraction of the speed of my TNT2?
39. Should I use a unified back buffer (UBB) or not?

I. Geometry

1. What are the fastest transfer mechanisms for geometry?

Fastest	DrawElements/DrawArrays Using wglAllocateMemoryNV(size,0,0,1)	Saves data in video memory, eliminating any bus bottleneck. **Very poor** read/write access.
	DrawElements/DrawArrays Using wglAllocateMemoryNV(size,0,0,.5)	Saves data in AGP (uncached) memory, and allows hardware to pull it directly. **Very poor** read access, must write sequentially (see below)
	Display Lists	Can encapsulate data in the most efficient manner for hardware, though they are immutable (i.e. once created, you can’t alter them in any way).
	DrawElements using Compiled Vertex Arrays (glLockArraysEXT)	Copies locked vertices to AGP memory, so that the hardware can then pull it directly. Only one mode is supported (see q, 7 below).
	DrawElements and DrawArrays using Vertex Arrays with Common Data Formats	Optimized to assemble primitives as efficiently as possible, and minimizes function call overhead. 13 formats supported (see q. 6).
	Immediate Mode	Multiple function calls required per primitive results in relatively poor performance compared to other options above.
Slowest	All Other Vertex Arrays	Must be copied from application memory to AGP memory before the hardware can pull it. Since data can change between calls, data must be copied every time, which is expensive.

2. What are the fastest primitives to use?

Fastest	GL_TRIANGLE_STRIP GL_TRIANGLE_FAN GL_QUAD_STRIP	These maximize reuse of the vertices shared within a given graphics primitive, and are all similarly fast.
	GL_TRIANGLES GL_QUADS	These aggregate (potentially multiple) disjoint triangles and quads, and amortize function overhead over multiple primitives.
Slowest	GL_POLYGON	A bit slower than the independent triangles and

		quads.

The GeForce2 GTS is able to setup primitives much faster than GeForce 256 or Quadro, so that all primitives are equally fast when accessing vertices in the vertex cache (see vertex cache question below for other details).

3. Which vertex array calls should I use?

Fastest	glDrawElements	Can take advantage of shared vertices and conserve front-side bus bandwidth by merely sending indices to the data.
	glDrawArrays	The most efficient way to send vertices that are **not shared**, though much slower than glDrawElements in the common case of shared vertices.
Slowest	glArrayElement	Call overhead per vertex severely impacts performance. Avoid if at all possible.

4. How does processor speed and/or bus bandwidth (AGP/AGP2X/AGP4X) affect this?

Unlike TNT2, NVIDIA GPUs all have hardware T&L, and processor speed is not nearly so important for attaining good T&L performance. Basically, only in immediate mode will processor speed play any significant role in determining T&L performance. Bus bandwidth is another matter, however, as it ultimately limits how quickly the data can pass from system memory to the GPU. AGP4X is needed for optimal performance in many of the transfer modes utilizing the bus, but even in these modes, AGP2X will yield performance close to AGP4X. Standard AGP, on the other hand, creates a bottleneck for many of the transfer modes, and can result in performance far less than optimal.

5. What is the best manner in which to organize my geometry in memory?

There is no inherent advantage, nor disadvantage, to using glInterleavedArrays versus glVertexPointer, glTexCoordPointer, etc. Similarly, there is typically no performance advantage to interleaving data versus keeping the components in separate, disjoint arrays.

6. What do vertex arrays buy me in terms of performance?

Vertex arrays minimize the number of OpenGL calls that must be made to send geometry down the pipeline. Some data formats are specifically optimized for use within regular vertex arrays:

Vertex Size/Type	**Normal Type**	**Color Size/Type**	**Texture Unit 0 Size/Type**	**Texture Unit 1 Size/Type**	**Fog Coord Type**
3/GLfloat	-	-	-	-	-
3/GLfloat	-	-	2/GLfloat	-	-
3/GLfloat	-	-	2/GLfloat	2/GLfloat	-
3/GLfloat	GLfloat	-	-	-	-
3/GLfloat	GLfloat	-	2/GLfloat	-	-
3/GLfloat	GLfloat	-	2/GLfloat	2/GLfloat	-
3/GLfloat	-	3/GLfloat	-	-	-
3/GLfloat	-	3/GLfloat	2/GLfloat	-	-
3/GLfloat	-	3/GLfloat	2/GLfloat	2/GLfloat	-

3/GLfloat	-	4/GLubyte	-	-	-
3/GLfloat	-	4/GLubyte	2/GLfloat	-	-
3/GLfloat	-	4/GLubyte	2/GLfloat	2/GLfloat	-
3/GLfloat	-	-	2/GLfloat	3/GLfloat	-
3/GLfloat	-	-	-	-	GLfloat
3/GLfloat	-	-	2/GLfloat	-	GLfloat
3/GLfloat	-	-	2/GLfloat	2/GLfloat	GLfloat

Other formats are likely to be slower than immediate mode.

7. What do compiled vertex arrays (CVAs) buy me in terms of performance?

Although your mileage may vary, compiled vertex arrays can yield a large increase in performance over other modes of transport – specifically, if you frequently reuse vertices within a vertex array, have the appropriate arrays enabled and use glDrawElements. Only one data format is specifically optimized for use within CVAs:

Vertex Size/Type	Normal Type	Color Size/Type	Texture Unit 0 Size/Type	Texture Unit 1 Size/Type
3/GLfloat	-	4/GLubyte	2/GLfloat	2/GLfloat

Note that there is no corresponding glInterleavedArrays enumerant for this format (i.e. you must use glVertexPointer, glColorPointer and glTexCoordPointer to specify the arrays).

When using compiled vertex arrays with this format, it's important to maximize use of the vertices that have been locked. For example, if you lock down 100 vertices and only use 25 of them in subsequent glDrawElements calls before unlocking, you will have relatively poor performance.

For more flexibility in accelerated data formats, it's recommended that vertex_array_range extension be used (see below).

8. How can I maximize performance with the vertex_array_range extension?

Currently, you should only use the vertex_array_range with memory allocated by wglAllocateMemoryNV (or glXAllocateMemoryNV) given the following settings:

Memory Allocated	ReadFrequency	WriteFrequency	Priority
AGP Memory	[0, .25)	[0, .25)	(.25, .75]
Video Memory	[0, .25)	[0, .25)	(.75, 1]

All other settings will yield relatively poor performance. Use video memory sparingly, and only for static geometry. You may use AGP memory for dynamic geometry, but write your data to these buffers sequentially to maximize memory bandwidth (it is uncached memory, and sequentially writing is **essential** to take advantage of the write combiners within the CPU that batch up multiple writes into a single, efficient block write). And being uncached, read access will be very, very slow – it may be best to keep two buffers, one allocated by standard malloc for general R/W access and the other allocated by wglAllocateMemoryNV that is only written to – synchronization would copy data from the R/W buffer sequentially into the AGP memory. Keep the vertex array strides to a reasonable length (less than 256), and mind the necessary alignment restrictions in the extension specification. Also, do not use wglAllocateMemoryNV unless you use and enable the vertex_array_range extension. If you do not heed those restrictions, you will certainly have less than optimal (if not poor) performance.

9. Is there an optimal size vertex array to define/use?
There is no hard and fast rule for vertex array size with respect to performance. Allocating a huge amount of AGP memory is probably not wise, however, since that memory will not be available to the OS (possibly causing unnecessary thrashing).

10. Should I use display lists for static geometry?
Yes, they are simple to use and the driver will choose the optimal way to transfer the data to the GPU.

11. Will I get better performance if I chain together separate triangles with degenerate triangle strips?
No. Draw what you can with triangle strips (or quad strips) and triangle fans, then draw the remaining independent triangles using glBegin(GL_TRIANGLES)…glEnd().

12. I've heard that NVIDIA GPUs have a vertex cache – how do I use it?
All NVIDIA GPUs have a 16 element post-T&L vertex cache (also called the "vertex file"), though the effective size is closer to 10 elements when you consider pipelining. You must adhere to a few rules to take advantage of the vertex cache:

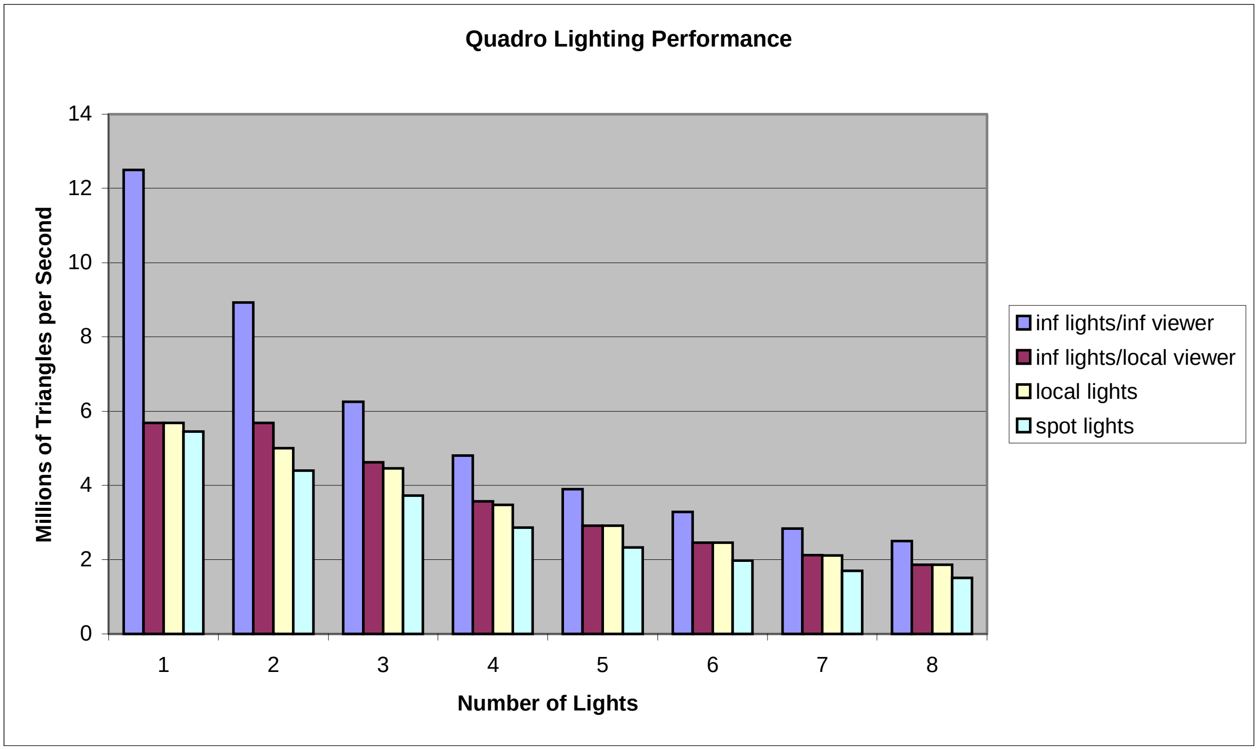
1. Use glDrawElements or glDrawRangeElements
2. Use the NV_vertex_array_range extension (see question 8 above) or the optimized compiled vertex array format (see question 7 above).
3. Ensure your vertices are shared between multiple primitives, and that you have decent vertex cache coherency (i.e. adjacent triangles are drawn together)

<u>II. Lighting</u>

13. What lighting mode is fastest?
Directional (AKA infinite) lights, with infinite (i.e. non-local) viewer and one-sided lighting.

14. Which ones should I avoid?
When using directional lights, avoid using local viewer, because it may cut your performance in half. However, when using local lights, you can use local viewer "for free" – that is, the GPU can calculate the local viewer at the same performance as infinite viewer. Two-sided lighting will be slower than one-sided lighting, and should only be used when absolutely necessary.

15. How many lights should I use?
In general, use as few as possible. When using local lights with attenuation, far-off lights will often not contribute to a given surface, although many calculations will still have to be made by the GPU. You can optimize for this by reducing the number of enabled lights to those in an object's immediate vicinity. Reference the graph below to see how much additional lights cost.



16. Should I turn normalization on or off for maximum performance?

The GPU performs normalization very efficiently, so that the cost for enabling it is negligible. Since unexpectedly bright or dim lighting can occur if normalization is disabled (with non-unit length normals), it's recommended that you always enable normalization.

17. Should I use the rescale normal extension to increase performance?

No, enable normalization instead. [See question above]

18. Is it faster if I only want to calculate the diffuse component, not the specular?

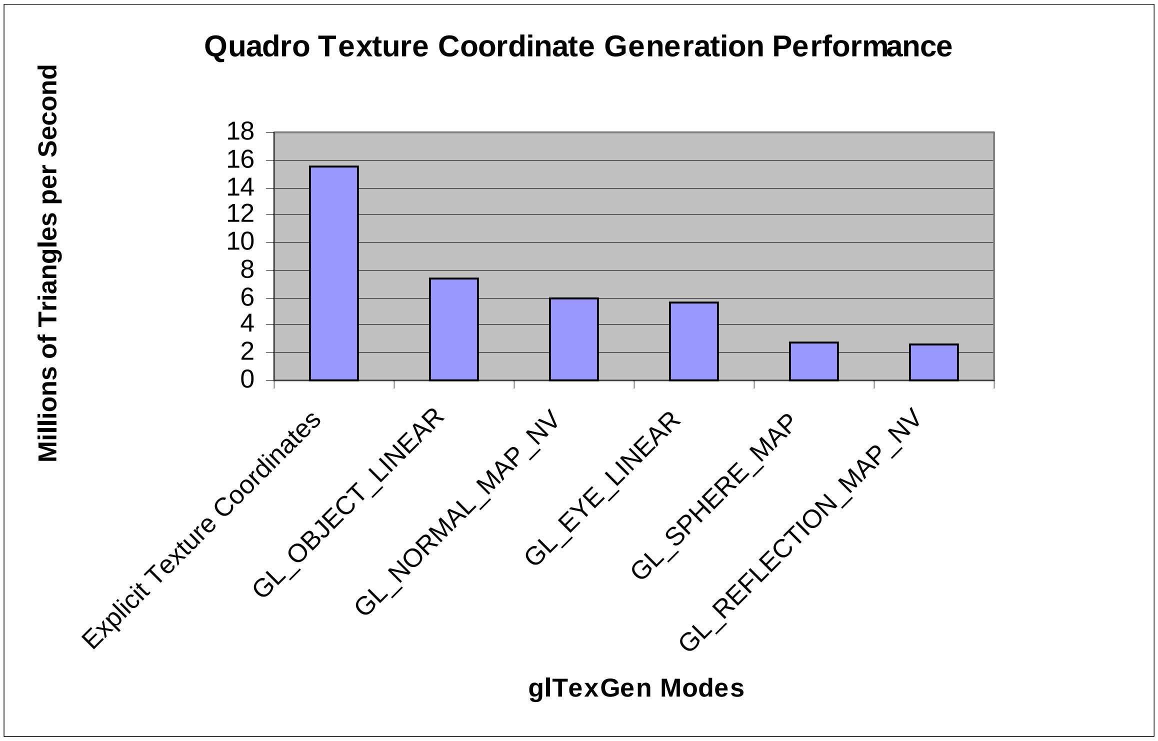
If you want to only compute the diffuse component – presumably by setting the specular material to black – additional performance will not be gained on NVIDIA GPUs, which include the specular calculation "for free". Separating the diffuse and specular colors and applying the specular component after texturing also incurs no additional T&L cost, though rasterization of large, non-Z buffered polygons may be slower.

Specifically, GeForce2 GTS interpolates the secondary/specular color at full speed. GeForce2 MX, GeForce 256 and Quadro all check if the specular color is constant across all the vertices of a triangle. If constant, the interpolation unit runs at full speed because there is no need to interpolate the secondary color. However, if the color varies over the triangle, the color interpolators have to be double pumped with the diffuse and specular color, which will cause that unit to run at half speed. Bear in mind that this will reduce overall performance only if this unit is already the bottleneck.

II. Texture Coordinate Generation

19. Which TexGen modes are hardware accelerated?

All 6 TexGen modes are hardware accelerated, but not at similar performance. See chart below.



20. Is the texture matrix hardware accelerated?

Yes, transformations for both texture units are performed in the GPU. There may be a performance penalty associated with the texture matrix. While the maximum performance on Quadro with an identity texture matrix is almost 16M triangles/second (see graph above), the performance will drop to around 10 M triangles/second with a non-identity texture matrix. If you are transfer-bound or raster-bound, you will not see any performance drop at all.

21. Is there hardware acceleration for two sets of texture coordinates?

Yes. This includes two TexGen units and two texture matrices.

III. Clipping and Culling

22. Should I perform any clipping myself?

No, it's fastest to allow OpenGL to handle it, since the GPU performs viewport clipping very efficiently. In order to take advantage of this clipping, applications should pass in unclipped geometry. Applications should continue to perform gross culling against the view frustum before sending complex objects, and some intelligent scene occlusion culling, such as a BSP.

23. Are user-defined clip planes hardware accelerated?

Yes, a number of user-defined clip planes are hardware accelerated through use of texture mapping and special hardware features.

24. How many user-defined clip planes are hardware accelerated?

For every texture unit you have left unused, you get two hardware user-defined clip planes. The caveat is that enabling polygon stipple counts as using a texture unit if you are not already using both texture units (see question on polygon stippling below). For example, you can use 2 clip planes with single-texturing, and 4 clip planes with no texturing, assuming no polygon stipple.

If more clip planes are defined than can be implemented by the hardware, the driver falls back to software clipping. If lighting is disabled, the driver can use fairly fast clip routines. However, clip planes are harder when lighting is enabled, because you have to light the vertices and then apply the clip planes, interpolating the lighted vertex results to the clipped coordinates. If lighting is enabled, the driver must use fairly slow clipping code. Avoid this case, if at all possible. In fact, avoid user-defined clipping planes altogether, if possible.

IV. Texturing

25. How can I maximize texture downloading performance?

Best RGB/RGBA texture image formats/types in order of performance:

Image Format	Image Type	Texture Internal Format
GL_RGB	GL_UNSIGNED_SHORT_5_6_5	GL_RGB
GL_BGRA	GL_UNSIGNED_SHORT_1_5_5_5_REV	GL_RGBA
GL_BGRA	GL_UNSIGNED_SHORT_4_4_4_4_REV	GL_RGBA
GL_BGRA	GL_UNSIGNED_INT_8_8_8_8_REV	GL_RGBA
GL_RGBA	GL_UNSIGNED_INT_8_8_8_8	GL_RGBA

Bear in mind that the NVIDIA GPUs store all 24-bit texels in 32-bit entries, so try using the spare alpha channel for something worthwhile, or it will just be wasted space. Moreover, 32-bit texels can be downloaded at twice the speed of 24-bit texels. Single or dual component texture formats such as GL_LUMINANCE, GL_ALPHA and GL_LUMINANCE_ALPHA are also very effective, as well as space efficient, particularly when they are blended with a constant color (e.g. grass, sky, etc.). Most importantly, **always** use glTexSubImage2D instead of glTexImage2D (and glCopyTexSubImage2D instead of glCopyTexImage2D) when updating texture images. The former call avoids any memory freeing or allocation, while the latter call may be required to reallocate its texture buffer for the newly defined texture.

26. Should I use texture compression?

If image fidelity is not of utmost importance, you should definitely consider using texture compression via the GL_ARB_texture_compression (http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_compression.txt) and GL_texture_compression_s3tc (http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt) extensions. This technology allows larger textures to be put in a smaller space, and thus reduces the chance of texture thrashing. The compressed texture can also be downloaded much faster to the GPU. See the NVIDIA developer web site for a white paper and code example.

27. How can I maximize texture rendering performance?
In general, it's best to minimize the number of texture binds that must be performed. By sorting its objects by texture, an application can optimally render them in order. Moreover, try to eliminate multiple passes over the same object by using multitexture. Although performing 2 texture multitexture may be more expensive than a single texture on GeForce 256 and Quadro, it is still **much** cheaper than performing another pass. And under most conditions, GeForce 2 GTS can perform dual-texturing at the same rate as single-texturing. Also consider using the GL_NV_register_combiners extension to reduce the number of required passes. This extension provides a good deal of flexibility in combining RGB and ALPHA components, as well as exposing functions such as dot products on a per-pixel level. If using register combiners, try to use only one general combiner, since using two combiners may lower texturing performance. By all means, however, use two general combiners rather than create another rendering pass!

28. What filtering modes should I use?
Mipmapping is always advised, particularly for minified objects. Minified objects create a very large stride through the non-mipmapped texture image, yielding poor cache utilization. Mipmapping greatly reduces the stride and results in higher cache utilization, and in turn, higher performance. Choose the 4-tap GL_LINEAR_MIPMAP_NEAREST as a minification filter, as it will be faster than the 8-tap trilinear filter.

29. How much performance will anisotropic filtering take away?
Anisotropic filtering comes at a slight performance penalty on NVIDIA GPUs (around 10%). It is most effectively used in a "Highest Quality" mode in concert with trilinear texture filtering.

30. What kind of performance increase can I expect from using paletted textures?
Paletted textures will not create a large increase in performance, and will possibly even decrease performance if shared palettes are not employed.

VI. Other Fragment Operations

31. What are the performance implications of polygon stippling?
Polygon stippling is implemented via texture mapping in NVIDIA GPUs, and though fast, it burns a texture unit. Performing polygon stippling with dual-texturing will force the driver to render in software.

32. What fragment operations should I avoid?
Try to curb use of blending, because it requires a read/modify/write operation. All non-zero blending modes cut fillrates in half (compared to non-blended rendering). Use alpha test instead of blending where feasible (e.g. to render sprites and such). One operation to stay away from is the color logical operator (as known as logic op). Only the GL_COPY operator is hardware accelerated on NVIDIA GPUs, relegating the rest to software rendering, at a mere fraction of the hardware's performance.

VII. Pixel Transfers

33. What are the best formats/types to use with glReadPixels and glDrawPixels?
For 32-bit glReadPixels, stick to using GL_UNSIGNED_BYTE type, with GL_RGB, GL_RGBA and GL_BGRA formats. For 16-bit glReadPixels, use GL_FLOAT type, with GL_RGB format. For 16/32-bit glDrawPixels, use GL_UNSIGNED_BYTE type, with GL_RGB and GL_RGBA formats. A type of GL_FLOAT will also give decent, though lower, performance with these formats.

34. I want to read back the depth buffer for incremental updates; how should I do this?
Writing to the depth buffer via glDrawPixels is quite slow (though reading the depth buffer via glReadPixels is moderately fast). For performing incremental updates to scenes by saving away the color and depth buffers, consider using the GL_KTX_buffer_region extension.

VIII. Miscellaneous

35. How much will Full Scene Anti-Aliasing (FSAA) slow me down?
Your mileage will vary depending upon which part of the system is the performance bottleneck. In general, if you are limited by anything but rasterization (e.g. your CPU’s speed, or T&L performance), FSAA should not incur any cost at all. If you’re limited by rasterization, however, your performance will drop in proportion to your super-sampling rate. For example, 2X FSAA requires just over two times the rasterization as no-FSAA, while 4X FSAA requires four times the rasterization as no-FSAA.

FSAA causes more video memory to be used, so texture thrashing may occur with FSAA enabled, where it did not occur with it disabled.

36. Is context switching expensive?
Yes. Context switching is often a problem in workstation applications, though not as commonly a problem in games. Reduce context switching to a minimum by reusing a single context, and bind it to separate windows, if necessary. It’s best to have merely a single window/context and use glViewport and glScissor to restrict rendering to specific “sub-windows”.

37. What about state changes?
State changes can severely impact performance. As such, they should be minimized by binning primitives with similar state (textures first, then lights, then blending modes, materials and so on) and drawing them all at the same time.

38. Why is my GeForce 256 running at a fraction of the speed of my TNT2?
Chances are, you have antialiased polygons enabled (i.e. glEnable(GL_POLYGON_SMOOTH)) and you’re running on 3.XX drivers. If you turn it off, performance will increase dramatically.

39. Should I use a unified back buffer (UBB) or not?
Quadro has the ability to enable a unified back buffer (in fact, it’s enabled by default). The unified back buffer is particularly useful for applications that use many (overlapping) windows and cannot afford to create separate back-buffers for each of them, since it uses too much framebuffer memory. UBB may be slightly slower for single windowed apps, so if you’re running games, it’s usually not a good idea to have it enabled. If you’re running workstation apps, however, it probably **is** a good idea to enable it.