Chapter 11. NVIDIA Shading

Mark Kilgard

Chapter 11: NVIDIA Shading

Mark J. Kilgard
NVIDIA Corporation
Austin, Texas

This chapter provides details about NVIDIA's GPU hardware architecture and API support. NVIDIA's latest GPUs are designed to fully support the rendering and shading features of both DirectX 9.0c and OpenGL 2.0. NVIDIA provides 3D game and application developers your choice of high-level shading languages (Cg, OpenGL Shading Language, or DirectX 9 HLSL) as well as full support for low-level assembly interfaces to shading.

Collected in this chapter are the following articles:

- *GeForce 6 Architecture*: This paper, re-printed from *GPU Gems 2*, is the most detailed publicly available description of NVIDIA GeForce 6 Series of GPUs.

- *NVIDIA GPU Historical Data*: This two page table collects performance data over a 7-year period on NVIDIA GPUs. This table presents the historical basis for expecting continuing graphics hardware performance improvements. What do the financial types always say? Past performance is not a guarantee of future return.

- *NVIDIA OpenGL 2.0 Support:* The GeForce 6 Series has the broadest hardware support for OpenGL 2.0 available at the time these notes were prepared. Key OpenGL 2.0 hardware-accelerated features include fully-general non-power-of-two textures, multiple draw buffers (also known as multiple render targets or MRT), two-sided stencil testing, OpenGL Shading Language (GLSL), GLSL support for vertex textures, GLSL support for both per-vertex *and* per-fragment dynamic branching, separate blend equations, and points sprites.

The GeForce 6 Series GPU Architecture

Emmett Kilgariff
NVIDIA Corporation

Randima Fernando
NVIDIA Corporation

<u>Notice</u>: This article is reprinted with permission from Chapter 30 of *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (ISBN: 0321335597, edited by Matt Pharr). References to other chapters within the text refer to chapters within the book, not these notes.

The previous chapter [of *GPU Gems 2*] described how GPU architecture has changed as a result of computational and communications trends in microprocessing. This chapter describes the architecture of the GeForce 6 Series GPUs from NVIDIA, which owe their formidable computational power to their ability to take advantage of these trends. Most notably, we focus on the GeForce 6800 (NVIDIA's flagship GPU at the time of writing, shown in Figure 30-1), which delivers hundreds of gigaflops of single-precision floating-point computation, as compared to approximately 12 gigaflops for high-end CPUs. We start with a general overview of where the GPU fits into the overall computer system, and then we describe the architecture along with details of specific features and performance characteristics.

Figure 30-1. The GeForce 6800 Microprocessor

30.1 How the GPU Fits into the Overall Computer System

The CPU in a modern computer system communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all command, texture, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at 66 MHz and was 32 bits wide, giving a transfer rate of 264 MB/sec. AGP 2×, 4×, and 8× followed, each doubling the available bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec available to and from the GPU. (Your mileage may vary; currently available motherboard chipsets fall somewhat below this limit—around 3.2 GB/sec or less.)

It is important to note the vast differences between the GPU's memory interface bandwidth and bandwidth in other parts of the system, as shown in Table 30-1.

Table 30-1. Available memory bandwidth in different parts of the computer system

Component	Bandwidth
GPU Memory Interface	35 GB/sec
PCI Express Bus (×16)	8 GB/sec
CPU Memory Interface (800 MHz Front-Side Bus)	6.4 GB/sec

Table 30-1 reiterates some of the points made in the preceding chapter: there is a vast amount of bandwidth available internally on the GPU. Algorithms that map to the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements.

30.2 Overall System Architecture

The next two subsections go into detail about the architecture of the GeForce 6 Series GPUs. Section 30.2.1 describes the architecture in terms of its graphics capabilities. Section 30.2.2 describes the architecture with respect to the general computational capabilities that it provides. See Figure 30-2 for an illustration of the system architecture.

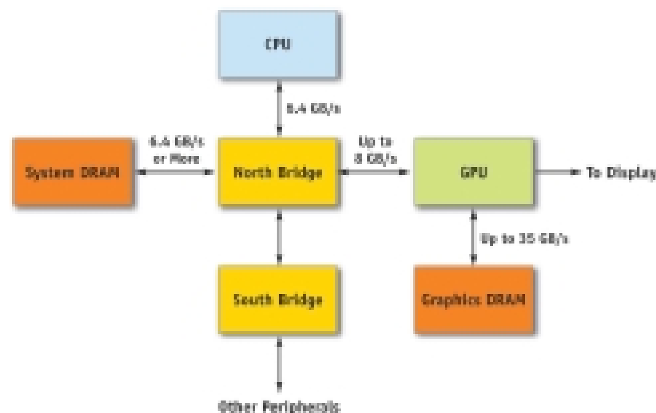


Figure 30-2. The Overall System Architecture of a PC

30.2.1 Functional Block Diagram for Graphics Operations

Figure 30-3 illustrates the major blocks in the GeForce 6 Series architecture. In this section, we take a trip through the graphics pipeline, starting with input arriving from the CPU and finishing with pixels being drawn to the frame buffer.

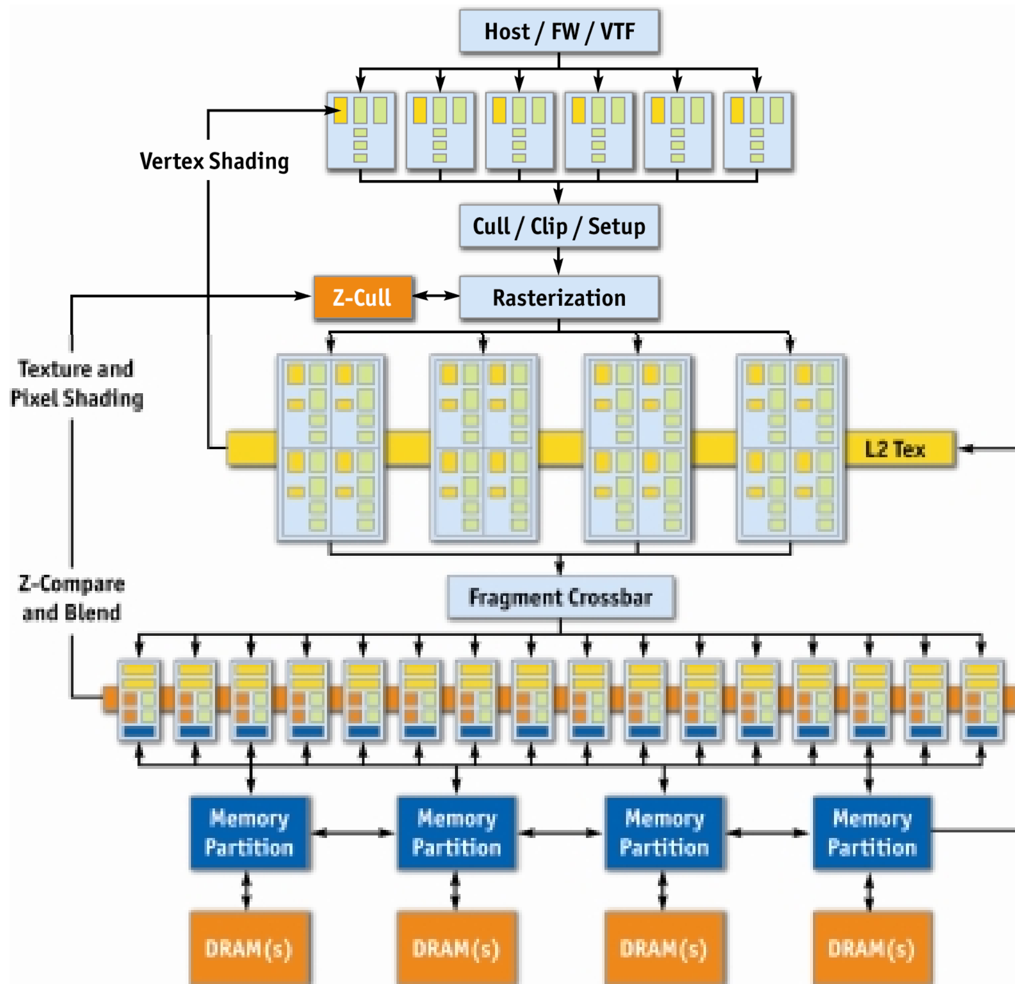


Figure 30-3. A Block Diagram of the GeForce 6 Series Architecture

First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. A command stream is written by the CPU, which initializes and modifies state, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages.

The vertex shading units, shown in Figure 30-4, allow for a program to be applied to each vertex in the object, performing transformations, skinning, and any other per-vertex operation the user specifies. For the first time, the GeForce 6 Series allows for vertex programs to fetch texture data. All operations are done in 32-bit floating-point (fp32) precision per component. The GeForce 6

Series architecture supports scalable vertex-processing horsepower, allowing the same architecture to service multiple price/performance points.

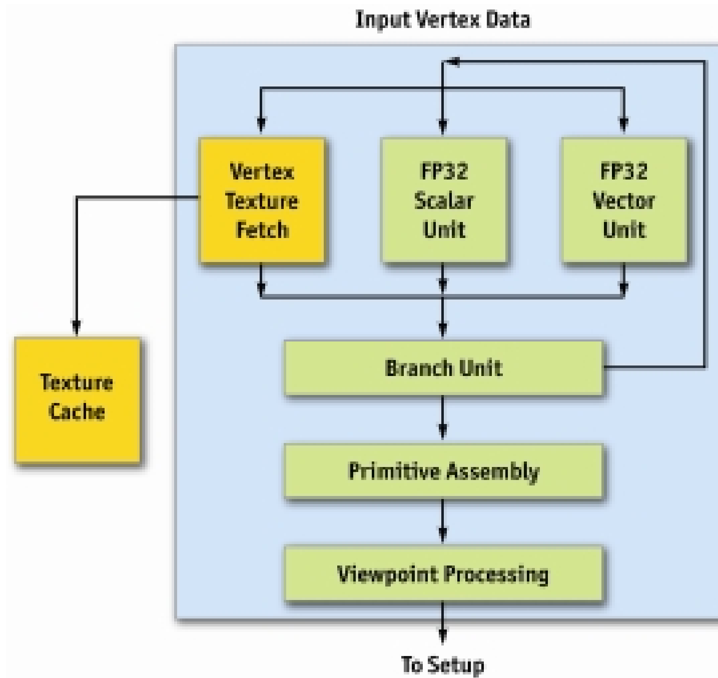


Figure 30-4. The GeForce 6 Series Vertex Processor

Because the vertex shader permits texture accesses, the vertex engines are connected to the texture cache, which is shared with the pixel shaders. In addition, there is a vertex cache that stores vertex data both before and after the vertex shader, reducing fetch and computation requirements. This means that if a vertex index occurs twice in a draw call (for example, in a triangle strip), the entire vertex program doesn't have to be rerun for the second instance of the vertex—the cached result is used instead.

Vertices are then grouped into primitives, which are points, lines, or triangles. The Cull/Clip/Setup blocks perform per-primitive operations, removing primitives that aren't visible at all, clipping primitives that intersect the view frustum, and performing edge and plane equation setup on the data in preparation for rasterization.

The rasterization block calculates which pixels (or samples, if multisampling is enabled) are covered by each primitive, and it uses the z-cull block to quickly discard pixels (or samples) that are occluded by objects with a nearer depth value.

Figure 30-5 illustrates the pixel shader and texel pipeline. The texture and pixel shading units operate in concert to apply a shader program to each pixel independently. The GeForce 6 Series architecture supports a scalable amount of pixel-processing horsepower. Another popular way to say this is that the GeForce 6 Series architecture can have a varying number of *pixel pipelines*. Similar to the vertex shader, texture data is cached on-chip to reduce bandwidth requirements and improve performance.

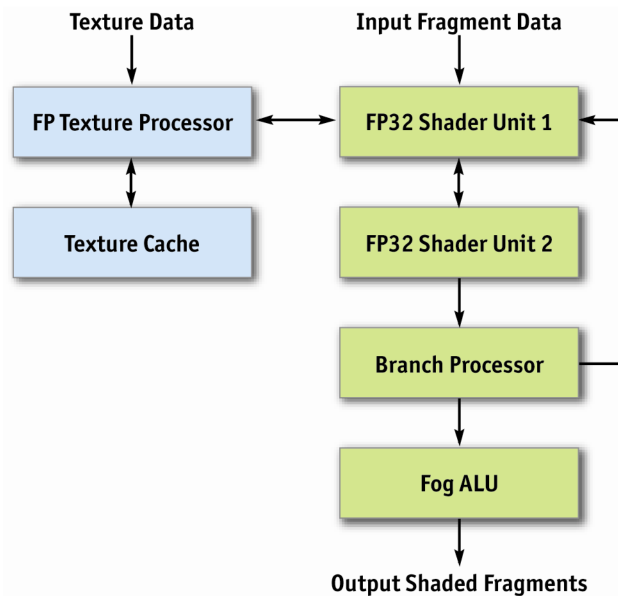


Figure 30-5. The GeForce 6 Series Pixel Shader and Texel Pipeline

The texture and pixel shading unit operates on squares of four pixels (called *quads*) at a time, allowing for direct computation of derivatives for calculating texture level of detail. Furthermore, the pixel shader works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with each pixel shader engine working on one pixel concurrently), hiding the latency of texture fetch from the computational performance of the pixel shader.

The pixel shader uses the texture unit to fetch data from memory, optionally filtering the data before returning it to the pixel shader. The texture unit supports many source data formats (see Section 30.3.3, "Supported Data Storage Formats"). Data can be filtered using bilinear, tri-linear, or anisotropic filtering. All data is returned to the pixel shader in fp32 or fp16 format. A texture can be viewed as a 2D or 3D array of data that can be read by the texture unit at arbitrary locations and filtered to reconstruct a continuous function. The GeForce 6 Series supports filtering of fp16 textures in hardware.

The pixel shader has two fp32 shader units per pipeline, and pixels are routed through both shader units and the branch processor before re-circulating through the entire pipeline to execute the next series of instructions. This rerouting happens once for each core clock cycle. Furthermore, fp32 shader unit 1 can be used for perspective correction of texture coordinates when needed (by dividing by *w*), or for general-purpose multiply operations. In general, it is possible to perform eight or more math operations in the pixel shader during each clock cycle, or four math operations if a texture fetch occurs in the first shader unit.

On the final pass through the pixel shader pipeline, the fog unit can be used to blend fog in fixed-point precision with no performance penalty. Fog blending happens often in conventional graphics applications and uses the following function:

```
out = FogColor * fogFraction + SrcColor * (1 - fogFraction)
```

This function can be made fast and small in fixed-precision math, but in general IEEE floating point, it requires two full multiply-adds to do effectively. Because fixed point is efficient and sufficient for fog, it exists in a separate small unit at the end of the shader. This is a good example

of the trade-offs in providing flexible programmable hardware while still offering maximum performance for legacy applications.

Pixels leave the pixel shading unit in the order that they are rasterized and are sent to the z-compare and blend units, which perform depth testing (z comparison and update), stencil operations, alpha blending, and the final color write to the target surface (an off-screen render target or the frame buffer).

The memory system is partitioned into up to four independent memory partitions, each with its own dynamic random-access memories (DRAMs). GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economies and thereby reduce cost. Having smaller, independent memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred. All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four independent memory partitions give the GPU a wide (256 bits), flexible memory subsystem, allowing for streaming of relatively small (32-byte) memory accesses at near the 35 GB/sec physical limit.

30.2.2 Functional Block Diagram for Non-Graphics Operations

As graphics hardware becomes more and more programmable, applications unrelated to the standard polygon pipeline (as described in the preceding section) are starting to present themselves as candidates for execution on GPUs.

Figure 30-6 shows a simplified view of the GeForce 6 Series architecture, when used as a graphics pipeline. It contains a programmable vertex engine, a programmable pixel engine, a texture load/filter engine, and a depth-compare/blending data write engine.

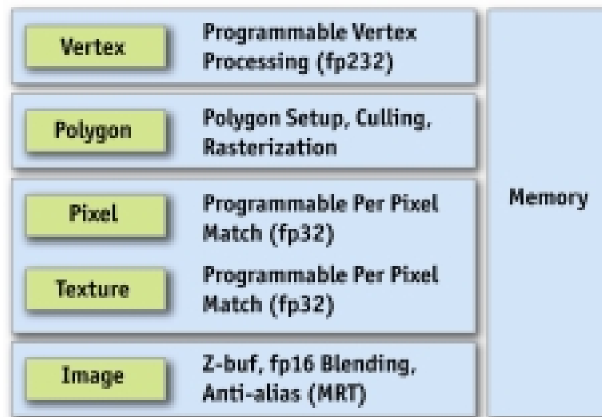


Figure 30-6. The GeForce 6 Series Architecture Viewed as a Graphics Pipeline

In this alternative view, a GPU can be seen as a large amount of programmable floating-point horsepower and memory bandwidth that can be exploited for compute-intensive applications completely unrelated to computer graphics.

Figure 30-7 shows another way to view the GeForce 6 Series architecture. When used for non-graphics applications, it can be viewed as two programmable blocks that run serially: the vertex shader and the pixel shader, both with support for fp32 operands and intermediate values. Both

use the texture unit as a random-access data fetch unit and access data at a phenomenal 35 GB/sec (550 MHz memory clock × 256 bits per clock cycle). In addition, both the vertex and the pixel shader are highly computationally capable. (Performance details follow in Section 30.4.)

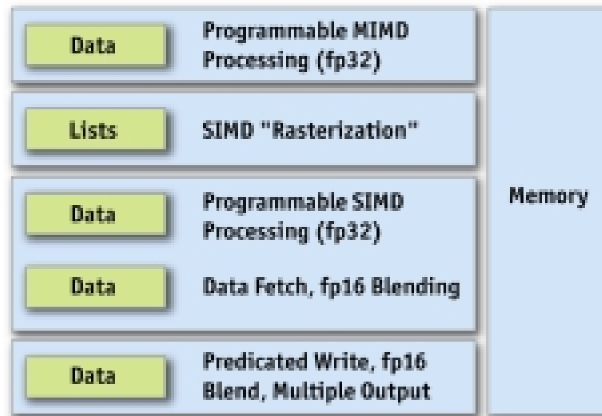


Figure 30-7. The GeForce 6 Series Architecture for Non-Graphics Applications

The vertex shader stores its data by passing it directly to the pixel shader, or by using the SIMD rasterizer to expand the data into interpolated values. At this point, each triangle (or point) from the vertex shader has become one or more *fragments*. Think of a fragment as a "candidate pixel": that is, it will pass through the pixel shader and several tests, and if it gets through all of them, it will end up carrying depth and color information to a pixel on the frame buffer (or render target).

Before a pixel reaches the pixel shader, the z-cull unit compares the pixel's depth with the values that already exist in the depth buffer. If the pixel's depth is greater, the pixel will not be visible, and there is no point shading that pixel, so the pixel shader isn't even executed. (This optimization happens only if it's clear that the pixel shader isn't going to modify the fragment's depth.) Thinking in a general-purpose sense, this *early culling* feature makes it possible to quickly decide to skip work on specific fragments based on a scalar test. Chapter 34 of this book, "GPU Flow Control Idioms," explains how to take advantage of this feature to efficiently predicate work for general-purpose computations.

After the pixel shader runs on a potential pixel (still a "fragment" because it has not yet reached the frame buffer), the fragment must pass a number of tests in order to move farther down the pipeline. (There may also be more than one fragment that comes out of the pixel shader if multiple render targets (MRTs) are being used. Up to four MRTs can be used to write out large amounts of data—up to 16 scalar floating-point values at a time, for example—plus depth.)

First, the scissor test rejects the fragment if it lies outside a specified subrectangle of the frame buffer. Although the popular graphics APIs define scissoring at this location in the pipeline, it is more efficient to perform the scissor test in the rasterizer. The x and y scissoring actually happen in the rasterizer, before pixel shading, and z scissoring happens during z-cull. This avoids all pixel shader work on scissored (rejected) pixels. Scissoring is rarely useful for general-purpose computation because general-purpose programmers typically draw rectangles to perform computations in the first place.

Next, the fragment's depth is compared with the depth in the frame buffer. If the depth test passes, the depth value in the frame buffer can optionally be replaced, and the fragment moves on in the pipeline.

After this, the fragment can optionally test and modify what is known as the stencil buffer, which stores an integer value per pixel. The stencil buffer was originally intended to allow programmers to mask off certain pixels (for example, to restrict drawing to a cockpit's windshield), but it has found other uses as a way to count values by incrementing or decrementing the existing value. This is used in stencil shadow volumes, for example.

If the fragment passes the depth and stencil tests, it can then optionally modify the contents of the frame buffer by using the blend function. A blend function can be described as

```
out = src * srcOp + dst * dstOp
```

where `source` is the fragment color flowing down the pipeline, `dst` is the color value in the frame buffer, and the `srcOp` and `dstOp` can be specified to be constants, source color components, or destination color components. Full blend functionality is supported for all pixel formats up to fp16×4. However, fp32 frame buffers don't support blending—only updating the buffer is allowed.

Finally, a feature called *occlusion query* makes it possible to quickly determine if any of the fragments that would be rendered in a particular computation would cause results to be written to the frame buffer. (Recall that fragments that do not pass the z-test don't have any effect on the values in the frame buffer.) Traditionally, the occlusion query test is used to allow graphics applications to avoid making draw calls for occluded objects, but it is very useful for GPGPU applications as well. For instance, if the depth test is used to tell which outputs need to be updated in a sparse array, updating depth can be used to tell when a given output has converged and no further work is needed. In this case, occlusion query can be used to tell when all output calculations are done. See Chapter 34 of this book, "GPU Flow Control Idioms," for further information about this idea.

30.3 GPU Features

This section covers both the fixed-function features and Shader Model 3.0 support (described in detail later) in GeForce 6 Series GPUs. As we describe the various pieces, we focus on the many new features that are meant to make applications shine (in terms of both visual quality and performance) on the GeForce 6 Series GPUs.

30.3.1 Fixed-Function Features

Geometry Instancing

With Shader Model 3.0, the capability for sending multiple batches of geometry with one Direct3D call has been added, greatly reducing driver overhead in these cases. The hardware feature that enables instancing is *vertex stream frequency*—the ability to read vertex attributes at a frequency less than once every output vertex, or to loop over a subset of vertices multiple times. Instancing is most useful when the same object is drawn multiple times with different positions and textures, for example, when rendering an army of soldiers or a field of grass.

Early Culling/Clipping

GeForce 6 Series GPUs are able to cull non-visible primitives before shading at a higher rate and clip partially visible primitives at full speed. Previous NVIDIA products would cull non-visible primitives at primitive-setup rates, and clip partially visible primitives at full speed.

Rasterization

Like previous NVIDIA products, GeForce 6 Series GPUs are capable of rendering the following objects:

- Point sprites
- Aliased and antialiased lines
- Aliased and antialiased triangles

Multisample antialiasing is also supported, allowing accurate antialiased polygon rendering. Multisample antialiasing supports all rasterization primitives. Multisampling is supported in previous NVIDIA products, though the 4× multisample pattern was improved for GeForce 6 Series GPUs.

Z-Cull

NVIDIA GPUs since GeForce3 have technology, called *z-cull*, that allows hidden surface removal at speeds much faster than conventional rendering. The GeForce 6 Series z-cull unit is the third generation of this technology, which has increased efficiency for a wider range of cases. Also, in cases where stencil is not being updated, early stencil reject can be employed to remove rendering early when stencil test (based on equals comparison) fails.

Occlusion Query

Occlusion query is the ability to collect statistics on how many fragments passed or failed the depth test and to report the result back to the host CPU. Occlusion query can be used either while rendering objects or with color and z-write masks turned off, returning depth test status for the objects that would have been rendered, without destroying the contents of the frame buffer. This feature has been available since the GeForce3 was introduced.

Texturing

Like previous GPUs, GeForce 6 Series GPUs support bilinear, tri-linear, and anisotropic filtering on 2D and cube-map textures of various formats. Three-dimensional textures support bilinear, tri-linear, and quad-linear filtering, with and without mipmapping. New texturing features on GeForce 6 Series GPUs are these:

- Support for all texture types (2D, cube map, 3D) with fp16×2, fp16×4, fp32×1, fp32×2, and fp32×4 formats
- Support for all filtering modes on fp16×2 and fp16×4 texture formats
- Extended support for non-power-of-two textures to match support for power-of-two textures, specifically:
  - Mipmapping
  - Wrapping and clamping
  - Cube map and 3D texture support

Shadow Buffer Support

NVIDIA graphics supports shadow buffering directly. The application first renders the scene from the light source into a separate z-buffer. Then during the lighting phase, it fetches the

shadow buffer as a projective texture and performs z-compares of the shadow buffer data against an iterated value corresponding to the distance from the light. If the texel passes the test, it's in light; if not, it's in shadow. NVIDIA GPUs have dedicated transistors to perform four z-compares per pixel (on four neighboring z-values) per clock, and to perform bilinear filtering of the pass/fail data. This more advanced variation of percentage-closer filtering saves many shader instructions compared to GPUs that don't have direct shadow buffer support.

High-Dynamic-Range Blending Using fp16 Surfaces, Texture Filtering, and Blending

GeForce 6 Series GPUs allow for fp16×4 (four components, each represented by a 16-bit float) filtered textures in the pixel shaders; they also allow performing all alpha-blending operations on fp16×4 filtered surfaces. This permits intermediate rendered buffers at a much higher precision and range, enabling high-dynamic-range rendering, motion blur, and many other effects. In addition, it is possible to specify a separate blending function for color and alpha values. (The lowest-end member of the GeForce 6 Series family, the GeForce 6200 TC, does not support floating-point blending or floating-point texture filtering because of its lower memory bandwidth, as well as to save area on the chip.)

30.3.2 Shader Model 3.0 Programming Model

Along with the fixed-function features listed previously, the capabilities of the vertex and the pixel shader have been enhanced in GeForce 6 Series GPUs. With Shader Model 3.0, the programming models for vertex and pixel shaders are converging: both support fp32 precision, texture lookups, and the same instruction set. Specifically, here are the new features that have been added.

Vertex Shader

- **Increased instruction count.** The total instruction count is now 512 static instructions and 65,536 dynamic instructions. The static instruction count represents the number of instructions in a program as it is compiled. The dynamic instruction count represents the number of instructions actually executed. In practice, the dynamic count can be vastly higher than the static count due to looping and subroutine calls.
- **More temporary registers.** Up to 32 four-wide temporary registers can be used in a vertex shader program.
- **Support for instancing.** This enhancement was described earlier.
- **Dynamic flow control.** Branching and looping are now part of the shader model. On the GeForce 6 Series vertex engine, branching and looping have minimal overhead of just two cycles. Also, each vertex can take its own branches without being grouped in the way pixel shader branches are. So as branches diverge, the GeForce 6 Series vertex shader still operates efficiently.
- **Vertex texturing.** Textures can now be fetched in a vertex program, although only nearest-neighbor filtering is supported in hardware. More advanced filters can of course be implemented in the vertex program. Up to four unique textures can be accessed in a vertex program, although each texture can be accessed multiple times. Vertex textures generate latency for fetching data, unlike true constant reads. Therefore, the best way to use vertex textures is to do a texture fetch and follow it with many arithmetic operations to hide the latency before using the result of the texture fetch.

Each vertex engine is capable of simultaneously performing a four-wide SIMD `MAD` (multiply-add) instruction and a scalar special function per clock cycle. Special function instructions include:

- Exponential functions: `EXP`, `EXPP`, `LIT`, `LOG`, `LOGP`
- Reciprocal instructions: `RCP`, `RSQ`
- Trigonometric functions: `SIN`, `COS`

Pixel Shader

- **Increased instruction count.** The total instruction count is now 65,535 static instructions and 65,535 dynamic instructions. There are limitations on how long the operating system will wait while the shader finishes working, so a long shader program working on a full screen of pixels may time-out. This makes it important to carefully consider the shader length and number of pixels rendered in one draw call. In practice, the number of instructions exposed by the driver tends to be smaller, because the number of instructions can expand as code is translated from Direct3D pixel shaders or OpenGL fragment programs to native hardware instructions.
- **Multiple render targets.** The pixel shader can output to up to four separate color buffers, along with a depth value. All four separate color buffers must be the same format and size. MRTs can be particularly useful when operating on scalar data, because up to 16 scalar values can be written out in a single pass by the pixel shader. Sample uses of MRTs include particle physics, where positions and velocities are computed simultaneously, and similar GPGPU algorithms. Deferred shading is another technique that computes and stores multiple four-component floating-point values simultaneously: it computes all material properties and stores them in separate textures. So, for example, the surface normal and the diffuse and specular material properties could be written to textures, and the textures could all be used in subsequent passes when lighting the scene with multiple lights. This is illustrated in Figure 30-8.

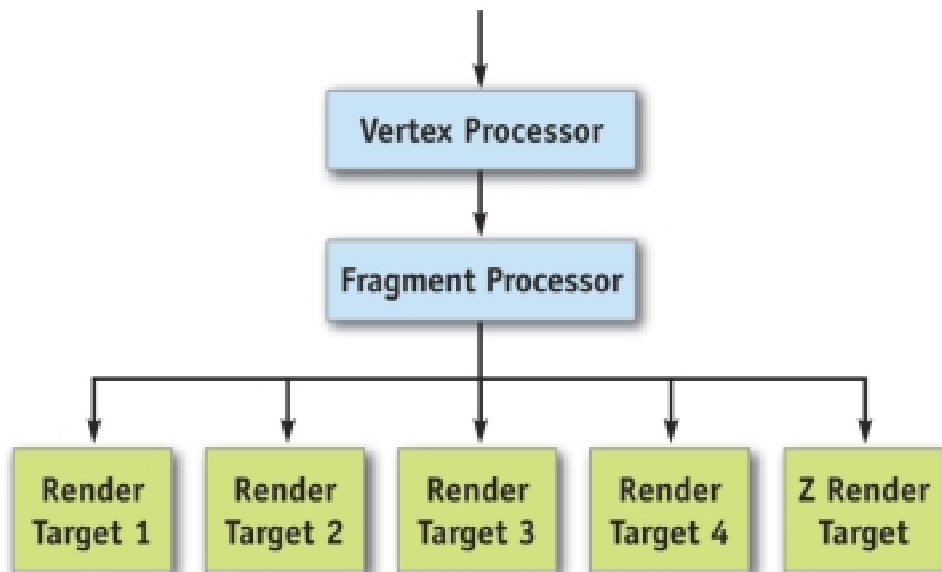


Figure 30-8. Using MRTs for Deferred Shading

- **Dynamic flow control (branching).** Shader Model 3.0 supports conditional branching and looping, allowing for more flexible shader programs.
- **Indexing of attributes.** With Shader Model 3.0, an index register can be used to select which attributes to process, allowing for loops to perform the same operation on many different inputs.
- **Up to ten full-function attributes.** Shader Model 3.0 supports ten full-function attributes/texture coordinates, instead of Shader Model 2.0's eight full-function attributes plus specular color and diffuse color. All ten Shader Model 3.0 attributes are interpolated at full fp32 precision, whereas Shader Model 2.0's diffuse and specular color were interpolated at only 8-bit integer precision.
- **Centroid sampling.** Shader Model 3.0 allows a per-attribute selection of center sampling, or *centroid sampling*. Centroid sampling returns a value inside the covered portion of the fragment, instead of at the center, and when used with multisampling, it can remove some artifacts associated with sampling outside the polygon (for example, when calculating diffuse or specular color using texture coordinates, or when using texture atlases).
- **Support for fp32 and fp16 internal precision.** Pixel shaders can support full fp32-precision computations and intermediate storage or partial-precision fp16 computations and intermediate storage.
- **3:1 and 2:2 co-issue.** Each four-component-wide vector unit is capable of executing two independent instructions in parallel, as shown in Figure 30-9: either one three-wide operation on RGB and a separate operation on alpha, or one two-wide operation on red-green and a separate two-wide operation on blue-alpha. This gives the compiler more opportunity to pack scalar computations into vectors, thereby doing more work in a shorter time.

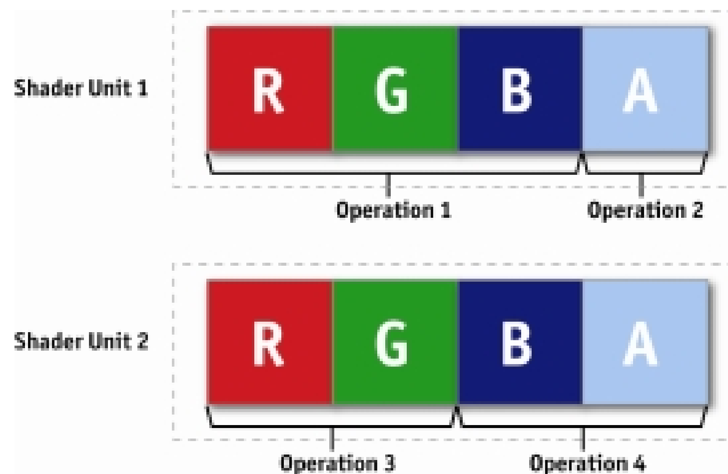


Figure 30-9. How Co-issue Works

- **Dual issue.** Dual issue is similar to co-issue, except that the two independent instructions can be executed on different parts of the shader pipeline. This makes the pipeline easier to schedule and, therefore, more efficient. See Figure 30-10.

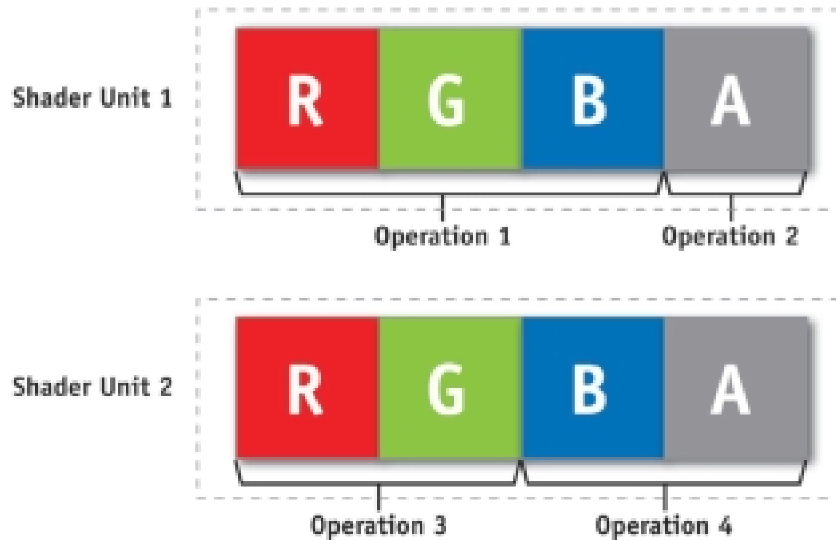


Figure 30-10. How Dual Issue Works

Shader Performance

The GeForce 6 Series shader architecture has the following performance characteristics:

- Each shader pipeline is capable of performing a four-wide, co-issue-able multiply-add (`MAD`) or four-term dot product (`DP4`), plus a four-wide, co-issue-able and dual-issue-able multiply instruction per clock in series, as shown in Figure 30-11. In addition, a multifunction unit that performs complex operations can replace the alpha channel `MAD` operation. Operations are performed at full speed on both fp32 and fp16 data, although store and bandwidth limitations can favor fp16 performance sometimes. In practice, it is sometimes possible to execute eight math operations as well as a texture lookup in a single cycle.

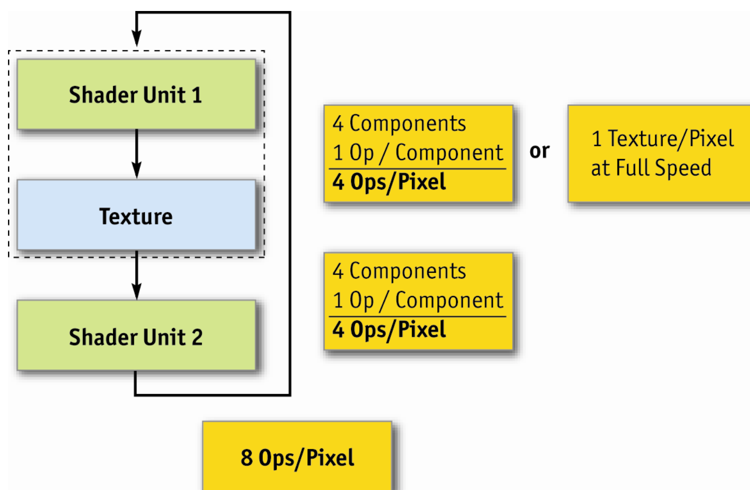


Figure 30-11. Shader Units and Capabilities

- Dedicated fp16 normalization hardware exists, making it possible to normalize a vector at fp16 precision in parallel with the multiplies and `MAD`s just described.

- Independent reciprocal operation can be performed in parallel with the multiply, `MAD`, and fp16 normalization described previously.
- Because the GeForce 6800 has 16 shader pipelines, the overall available performance of the system is given by these values multiplied by 16 and then by the clock rate.
- There is some overhead to flow-control operations, as defined in Table 30-2.

Table 30-2. Overhead incurred when executing flow-control operations in fragment programs.

Instruction	Cost (Cycles)
`If/endif`	4
`If/else/endif`	6
`Call`	2
`Ret`	2
`Loop/endloop`	4

Furthermore, pixel shader branching is affected by the level of divergence of the branches. Because the pixel shader operates on hundreds of pixels per instruction, if a branch is taken by some pixels and not others, all pixels execute both branches, but only writing to the registers on the branches each pixel is supposed to take. For low-frequency and mid-frequency branch changes, this effect is hidden, although it can become a limiter as the branch frequency increases.

30.3.3 Supported Data Storage Formats

Table 30-3 summarizes the data formats supported by the graphics pipeline.

Table 30-3. Data storage formats supported by the GeForce 6 series of GPUs.

Format	Description of Data in Memory	Vertex Texture Support	Pixel Texture Support	Render Target Support
B8	One 8-bit fixed-point number	No	Yes	Yes
A1R5G5B5	A 1-bit value and three 5-bit unsigned fixed-point numbers	No	Yes	Yes
A4R4G4B4	Four 4-bit unsigned fixed-point numbers	No	Yes	No
R5G6B5	5-bit, 6-bit, and 5-bit fixed-point numbers	No	Yes	Yes
A8R8G8B8	Four 8-bit fixed-point numbers	No	Yes	Yes
DXT1	Compressed 4×4 pixels into 8 bytes	No	Yes	No
DXT2,3,4,5	Compressed 4×4 pixels into 16 bytes	No	Yes	No
G8B8	Two 8-bit fixed-point numbers	No	Yes	Yes
B8R8_G8R8	Compressed as YVYU; two pixels in 32 bits	No	Yes	No
R8B8_R8G8	Compressed as VYUY; two pixels in 32 bits	No	Yes	No
R6G5B5	6-bit, 5-bit, and 5-bit unsigned fixed-point numbers	No	Yes	No
DEPTH24_D8	A 24-bit unsigned fixed-point number and 8 bits of garbage	No	Yes	Yes
DEPTH24_D8 FLOAT	A 24-bit unsigned float and 8 bits of garbage	No	Yes	Yes
DEPTH16	A 16-bit unsigned fixed-point number	No	Yes	Yes
DEPTH16_FLOAT	A 16-bit unsigned float	No	Yes	Yes
X16	A 16-bit fixed-point number	No	Yes	No
Y16_X16	Two 16-bit fixed-point numbers	No	Yes	No
R5G5B5A1	Three unsigned 5-bit fixed-point numbers and a 1-bit value	No	Yes	Yes
HILO8	Two unsigned 16-bit values compressed into two 8-bit values	No	Yes	No
HILO_S8	Two signed 16-bit values compressed into two 8-bit values	No	Yes	No
W16_Z16_Y16_X16 FLOAT	Four fp16 values	No	Yes	Yes
W32_Z32_Y32_X32 FLOAT	Four fp32 values	Yes, unfiltered	Yes, unfiltered	Yes
X32_FLOAT	One 32-bit floating-point number	Yes, unfiltered	Yes, unfiltered	Yes
D1R5G5B5	1 bit of garbage and three unsigned 5-bit fixed-point numbers	No	Yes	Yes
D8R8G8B8	8 bits of garbage and three unsigned 8-bit fixed-point numbers	No	Yes	Yes
Y16_X16 FLOAT	Two 16-bit floating-point numbers	No	Yes	No

30.4 Performance

The GeForce 6800 Ultra is the flagship product of the GeForce 6 Series family at the time of writing. Its performance is summarized as follows.

- 425 MHz internal graphics clock
- 550 MHz memory clock
- 600 million vertices/second
- 6.4 billion texels/second
- 12.8 billion pixels/second, rendering z/stencil-only (useful for shadow volumes and shadow buffers)
- 6 four-wide fp32 vector `MAD`s per clock cycle in the vertex shader, plus one scalar multifunction operation (a complex math operation, such as a sine or reciprocal square root)
- 16 four-wide fp32 vector `MAD`s per clock cycle in the pixel shader, plus 16 four-wide fp32 multiplies per clock cycle
- 64 pixels per clock cycle early z-cull (reject rate)

As you can see, there's plenty of programmable floating-point horsepower in the pixel and vertex shaders that can be exploited for computationally demanding problems.

30.5 Achieving Optimal Performance

While graphics hardware is becoming more and more programmable, there are still some tricks to ensuring that you exploit the hardware fully to get the most performance. This section lists some common techniques that you may find helpful. A more detailed discussion of performance advice is available in the *NVIDIA GPU Programming Guide*, which is freely available in several languages from the NVIDIA Developer Web Site (http://developer.nvidia.com/object/gpu_programming_guide.html).

30.5.1 Use Z-Culling Aggressively

Z-cull avoids work that won't contribute to the final result. It's better to determine early that a computation doesn't matter and save doing the work. In graphics, this can be done by rendering the z-values for all objects first, before shading. For general-purpose computation, the z-cull unit can be used to select which parts of the computation are still active, culling computational threads that have already resolved. See Section 34.2.3 of Chapter 34, "GPU Flow Control Idioms," for more details on this idea.

30.5.2 Exploit Texture Math When Loading Data

The texture unit filters data while loading it into the shader, thus reducing the total data needed by the shader. The texture unit's bilinear filtering can frequently be used to reduce the total work done by the shader if it's performing more sophisticated shading.

Often, large filter kernels can be dissected into groups of bilinear footprints, which are scaled and accumulated to build the large kernel. A few caveats here, most notably that all filter coefficients must be positive for bilinear footprint assembly to work properly. (See Chapter 20, “Fast Third-Order Texture Filtering,” for more information about this technique.)

30.5.3 Use Branching in Pixel Shaders Judiciously

Because the pixel shader is a SIMD machine operating on many pixels at a time, if some pixels in a given group take one branch and other pixels in that group take another branch, the pixel shader needs to take both branches. Also, there is a six-cycle overhead for if-else-endif control structures. These two effects can reduce the performance of branching programs if not considered carefully. Branching can be very beneficial, as long as the work avoided outweighs the cost of branching. Alternatively, conditional writes (that is, write if a condition code is set) can be used when branching is not performance-effective. In practice, the compiler will use the method that delivers higher performance when possible.

30.5.4 Use fp16 Intermediate Values Wherever Possible

Because GeForce 6 Series GPUs support a full-speed fp16 normalize instruction in parallel with the multiplies and adds, and because fp16 intermediate values reduce internal storage and data path requirements, using fp16 intermediate values wherever possible can be a performance win, saving fp32 intermediate values for cases where the precision is needed.

Excessive internal storage requirements can adversely affect performance in the following way: The shader pipeline is optimized to keep hundreds of pixels in flight given a fixed amount of register space per pixel (four fp32×4 registers or eight fp16×4 registers). If the register space is exceeded, then fewer pixels can remain in flight, reducing the latency tolerance for texture fetches, and adversely affecting performance. The GeForce 6 Series shader will have the maximum number of pixels in flight when shader programs use up to four fp32×4 temporary registers (or eight fp16×4 registers). That is, at any one time, a maximum of four temporary fp32×4 (or eight fp16×4 registers) are in use. This decision was based on the fact that for the overwhelming majority of analyzed shaders, four or fewer simultaneously active fp32×4 registers proved to be the sweet spot during the shaders’ execution. In addition, the architecture is designed so that performance degrades slowly if more registers are used.

Similarly, the register file has enough read and write bandwidth to keep all the units busy if reading fp16×4 values, but it may run out of bandwidth to feed all units if using fp32×4 values exclusively. NVIDIA’s compiler technology is smart enough to reduce this effect substantially, but fp16 intermediate values are never slower than fp32 values; because of the resource restrictions and the fp16 normalize hardware, they can often be much faster.

30.6 Conclusion

GeForce 6 Series GPUs provide the GPU programmer with unparalleled flexibility and performance in a product line that spans the entire PC market. After reading this chapter, you should have a better understanding of what GeForce 6 Series GPUs are capable of, and you should be able to use this knowledge to develop applications—either graphical or general purpose—in a more efficient way.

NVIDIA GPU Historical Data

Source: NVIDIA Corporation

December 2004

Estimate for CPU-based vertex processing.

Year	Product	New Features	OpenGL version	Direct3D version	Core Clk (Mhz)	Mem Clk (Mhz)	Mtri/sec	Mtri/sec per-year Increase	Mvert/sec	Mvert/sec per-year Increase
1998	Riva ZX	16-bit depth, color, and textures	1.1	DX5	100	100	3	-	1	-
1999	Riva TNT2	Dual-texturing, interpolated specular color, 32-bit depth/stencil, color, and texture	1.2	DX6	175	200	9	200%	2	100%
2000	GeForce2 GTS	Hardware transform & lighting, configurable fixed-point shading, cube maps, texture compression, 2x anisotropic texture filtering	1.3	DX7	166	333	25	178%	24	1086%
2001	GeForce3	Programmable vertex transformation, 4 texture units, dependent textures, 3D textures, shadow maps, multisampling, occlusion queries	1.4	DX8	200	460	30	20%	33	41%
2002	GeForce4 Ti 4600	Early Z culling, dual-monitor	1.4	DX8.1	300	650	60	100%	100	200%
2003	GeForce FX	Vertex program branching, floating-point fragment programs, 16 texture units, limited floating-point textures, color and depth compression	1.5	DX9	500	1000	167	178%	375	275%
2004	GeForce 6800 Ultra	Vetex textures, structured fragment branching, non-power-of-two textures, generalized floating-point textures, floating-point texture filtering and blending	2.0	DX9c	425	1100	170	2%	638	70%
					Increase over 6 years		56.7		637.5	

Mhz for DDR memory is doubled.

NVIDIA GPU Historical Data

Source: NVIDIA Corporation
December 2004

Year	Product	Single Texture Fill Mpix/sec	Single Texture Fill per-year Increase	Depth Stencil Only Fill Mpix/sec	4x FSAA Single Texture Fill Mpix/sec	4x FSAA Dual Texture Fill Mpix/sec	Texture Rate Mtex/sec	Texture Rate per-year increase	BW (GB/sec)	Effective BW (compressed) (GB/sec)	Process (um)	Transistor count (M)
1998	Riva ZX	100	-	100	25	13	100	-	1.6	1.6	0.35	4
1999	Riva TNT2	350	250%	350	88	44	350	250%	3.2	3.2	0.22	9
2000	GeForce2 GTS	664	90%	664	166	166	1328	279%	5.3	10.7	0.18	25
2001	GeForce3	800	20%	800	400	400	1600	20%	7.4	22.1	0.18	57
2002	GeForce4 Ti 4600	1200	50%	1200	600	600	2400	50%	10.4	31.2	0.15	63
2003	GeForce FX	2000	67%	4000	2000	2000	4000	67%	16.0	64.0	0.13	121
2004	GeForce 6800 Ultra	6800	240%	13600	6800	3400	6800	70%	35.2	140.8	0.13	222
		68.0		136.0	272.0	272.0	68.0		22.0	88.0		55.5

Notes:

- 4x FSAA Single Texture Fill: assumes supersampling since no multisampling in ZX, TNT2 or GeForce2 GTS.
- 4x FSAA Dual Texture Fill: assumes multi-pass since ZX has no multitexture.
- Texture Rate per-year increase (1998): Mipmap bilinear on ZX and TNT2.
- Texture Rate (2000): Double texture rate assumes two mipmap bilinear textures (not trilinear).
- Texture Rate (2001): Dual-textured trilinear.
- BW: 1,000,000 bytes per gigabyte (not 2^20 bytes)
- Effective BW: Assumes 1/3 T, 1/3 C, 1/3 Z bandwidth base. Assumes 4:1 compression in all cases.
- Effective BW (1998): No compression.
- Effective BW (2000): DXTC texture compression.
- Effective BW (2003): DXTC texture compression, Z compression, color compression (for multisampling).

NVIDIA OpenGL 2.0 Support

Mark J. Kilgard
February 2, 2005

These release notes explain NVIDIA's support for OpenGL 2.0. These notes are written mainly for OpenGL programmers writing OpenGL 2.0 applications. These notes may also be useful for OpenGL-savvy end-users seeking to understand the OpenGL 2.0 capabilities of NVIDIA GPUs.

This document addresses

- What is OpenGL 2.0?
- What NVIDIA Drivers and GPUs support OpenGL 2.0?
- Programmable Shading API Updates for OpenGL 2.0
- Correctly Detecting OpenGL 2.0 in Applications
- Enabling OpenGL 2.0 Emulation on Older GPUs
- Key Known Issues
- OpenGL 2.0 API Declarations
- Distinguishing NV3xGL-based and NV4xGL-based Quadro FX GPUs by Product Names

1. What is OpenGL 2.0?

OpenGL 2.0 is the latest core revision of the OpenGL graphics system. The OpenGL 2.0 specification was finalized September 17, 2004 by the OpenGL Architectural Review Board (commonly known as "the ARB").

OpenGL 2.0 incorporates the following functionality into the core OpenGL standard:

- **High-level Programmable Shading.** The OpenGL Shading Language (commonly called GLSL) and the related APIs for creating, managing, and using shader and program objects defined with GLSL is now a core feature of OpenGL.

  This functionality was first added to OpenGL as a collection of ARB extensions, namely `ARB_shader_objects`, `ARB_vertex_shader`, and `ARB_fragment_shader`. OpenGL 2.0 updated the API from these original ARB

extensions. These API updates are discussed in section 3.

- **Multiple Render Targets.** Previously core OpenGL supported a single RGBA color output from fragment processing. OpenGL 2.0 specifies a maximum number of draw buffers (though the maximum can be 1). When multiple draw buffers are provided, a low-level assembly fragment program or GLSL fragment shader can output multiple RGBA color outputs that update a specified set of draw buffers respectively. This functionality matches the `ARB_draw_buffers` extension.

- **Non-Power-Of-Two Textures.** Previously core OpenGL required texture images (not including border texels) to be a power-of-two size in width, height, and depth. OpenGL 2.0 allows arbitrary sizes for width, height, and depth. Mipmapping of such textures is supported. The functionality matches the `ARB_texture_non_power_of_two` extension.

- **Point Sprites.** Point sprites override the single uniform texture coordinate set values for a rasterized point with interpolated 2D texture coordinates that blanket the point with the full texture image. This allows application to define a texture pattern for rendered points. The functionality matches the `ARB_point_sprite` extension but with additional origin control.

- **Two-Sided Stencil Testing.** Previously core OpenGL provided a single set of stencil state for both front- and back-facing polygons. OpenGL 2.0 introduces separate front- and back-facing state. This can improve the performance of certain shadow volume and Constructive Solid Geometry (CSG) rendering algorithms. The functionality merges the capabilities of the `EXT_stencil_two_side` and `ATI_stencil_separate` extensions.

- **Separate RGB and Alpha Blend Equations.** Previously core OpenGL provided a blend equation (add, subtract, reverse subtract, min, or max) that applied to both the RGB and alpha components of a blended pixel. OpenGL 1.4 allowed separate RGB and alpha components to support distinct source and destination functions. OpenGL 2.0 generalizes the control to provide separate RGB and alpha blend equations.

- **Other Specification Changes.** OpenGL 2.0 includes several minor revisions and corrections to the specification. These changes are inconsequential to OpenGL programmers as the changes did not change the understood and implemented behavior of OpenGL. See appendix I.6 of the OpenGL 2.0 specification for details.

2. What NVIDIA Drivers and GPUs support OpenGL 2.0?

NVIDIA support for OpenGL 2.0 begins with the Release 75 series of drivers. GeForce FX (NV3x), GeForce 6 Series (NV4x), NV3xGL-based Quadro FX and NV4xGL-based Quadro FX GPUs, and all future NVIDIA GPUs support OpenGL 2.0.

Prior to Release 75, drivers for these OpenGL 2.0-capable GPUs advertised OpenGL 1.5 support but also exposed the feature set of OpenGL 2.0 through the corresponding extensions listed in section 1.

Earlier GPUs (such as GeForce2, GeForce3, and GeForce4) continue to support OpenGL 1.5 with no plans to ever support OpenGL 2.0 because the hardware capabilities of these GPUs are not sufficient to accelerate the OpenGL 2.0 feature set properly.

However, NVIDIA provides an option with Release 75 drivers to emulate OpenGL 2.0 features on these earlier GPUs. This option is further discussed in section 5. This emulation option is *not recommended for general users* because OpenGL 2.0 features will be emulated in software very, very slowly. OpenGL 2.0 emulation may be useful for developers and students without access to the latest NVIDIA GPU hardware.

2.1. Acceleration for GeForce 6 Series and NV4xGL-based Quadro FX

All key OpenGL 2.0 features are hardware-supported by NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs. These GPUs offer the best OpenGL 2.0 hardware acceleration available from any vendor today.

2.1.1. Fragment-Level Branching

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs support structured fragment-level branching. Structured branching allows standard control-flow mechanisms such as loops, early exit from loops (comparable to a `break` statement in C), *if-then-else* decision making, and function calls. Specifically, the hardware can support data-dependent branching such as a loop where the different fragments early exit the loop after a varying number of iterations.

Much like compilers for CPUs, NVIDIA's Cg-based compiler technology decides automatically whether to use the hardware's structured branching capabilities or using simpler techniques such as conditional assignment, unrolling loops, and inlining functions.

Hardware support for fragment-level branching is not as general as vertex-level branching. Some flow control constructs are too complicated or cannot be expressed by the hardware's structured branching capabilities. A few restrictions of note:

- Function calls can be nested at most 4 calls deep.
- *If-then-else* decision making can be nested at most 47 branches deep.
- Loops cannot be nested more than 4 loops deep.
- Each loop can have at most 255 iterations.

The compiler can often generate code that avoids these restrictions, but if not, the program object containing such a fragment shader will fail to compile. These restrictions are also discussed in the `NV_fragment_program2` OpenGL extension specification.

2.1.2. Vertex Textures

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs accelerate texture fetches by GLSL vertex shaders.

The implementation-dependent constant `GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS` is advertised to be 4 meaning these GPUs provide a maximum of 4 vertex texture image units.

2.1.2.1. Hardware Constraints

Keep in mind these various hardware constraints for vertex textures:

- While 1D and 2D texture targets for vertex textures are supported, the 3D, cube map, and rectangle texture targets are not hardware accelerated for vertex textures.
- Just these formats are accelerated for vertex textures: `GL_RGBA_FLOAT32_ARB`, `GL_RGB_FLOAT32_ARB`, `GL_ALPHA_FLOAT32_ARB`, `GL_LUMINANCE32_ARB`, `GL_INTENSITY32_ARB`, `GL_FLOAT_RGBA32_NV`, `GL_FLOAT_RGB32_NV`, `GL_FLOAT_RG32_NV`, or `GL_FLOAT_R32_NV`.
- Vertex textures with border texels are not hardware accelerated.
- Since no depth texture formats are hardware accelerated, shadow mapping by vertex textures is not hardware accelerated

The vertex texture functionality precluded from hardware acceleration in the above list will still operate as specified but it will require the vertex processing be performed on the CPU rather than the GPU. This will substantially slow vertex processing. However, rasterization and fragment processing can still be fully hardware accelerated.

2.1.2.2. Unrestricted Vertex Texture Functionality

These features are supported for hardware-accelerated vertex textures:

- All wrap modes for S and T including all the clamp modes, mirrored repeat, and the three "mirror once then clamp" modes. Any legal border color is allowed.
- Level-of-detail (LOD) bias and min/max clamping.
- Non-power-of-two sizes for the texture image width and height.
- Mipmapping.

Because vertices are processed as ideal points, vertex textures accesses require an explicit LOD to be computed; otherwise the base level is sampled. Use the bias parameter of GLSL's `texture2DLod`, `texture1DLod`, and related functions to specify an explicit LOD.

2.1.2.3. Linear and Anisotropic Filtering Caveats

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs do not hardware accelerate linear filtering for vertex textures. If vertex texturing is otherwise hardware accelerated, `GL_LINEAR` filtering operates as `GL_NEAREST`. The mipmap minification filtering modes (`GL_LINEAR_MIPMAP_LINEAR`, `GL_NEAREST_MIPMAP_LINEAR`, or `GL_LINEAR_MIPMAP_NEAREST`) operate as if `GL_NEAREST_MIPMAP_NEAREST` so as not to involve any linear filtering. Anisotropic filtering is also ignored for hardware-accelerated vertex textures.

These same rules reflect hardware constraints that apply to vertex textures whether used through GLSL or `NV_vertex_program3` or Cg's `vp40` profile.

2.1.3. Multiple Render Targets

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs support a maximum of 4 simultaneous draw buffers (indicated by `GL_MAX_DRAW_BUFFERS`). Typically, you should request a pixel format for your framebuffer with one or more auxiliary buffers (commonly called *aux* buffers) to take advantage of multiple render targets.

Most pixel formats have an option for 4 auxiliary buffers. These buffers are allocated lazily so configuring with a pixel format supporting 4 auxiliary buffers but using fewer buffers in your rendering will not require video memory be allocated to never used buffers.

2.1.4. Non-Power-Of-Two Textures

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs fully support non-power-of-two textures at the fragment level. All texture formats (including compressed formats) and all texture modes such as shadow mapping, all texture filtering modes (including anisotropic filtering), borders, LOD control, and all clamp modes work as expected with non-power-of-two texture sizes. Non-power-of-two textures are also supported for vertex textures but with the limitations discussed in section 2.1.2.

2.1.4.1. Rendering Performance and Texture Memory Usage

For memory layout and caching reasons, uncompressed non-power-of-two textures may be slightly slower than uncompressed power-of-two textures of comparable size. However, non-power-of-two textures *compressed with S3TC* should have very comparable rendering performance to similarly compressed power-of-two textures.

For non-mipmapped non-power-of-two textures (as well as non-mipmapped power-of-two textures), the size of the texture in memory is roughly the width × height × depth (if 3D) × bytes-per-texel as you would expect. So in this case, a 640x480 non-power-of-two-texture *without mipmaps*, will take just 59% of the memory required if the image was rounded up to the next power-of-two size (1024×512).

Mipmapped power-of-two sized 2D or 3D textures take up roughly four-thirds times (1.333x) the memory of a texture's base level memory size. However, mipmapped non-power-of-two 2D or 3D textures take up roughly **two times** (2x) the memory of a texture's base level memory size. For these reasons, developers are discouraged from changing (for example) a 128x128 mipmapped texture into a 125×127 mipmapped texture hoping the slightly smaller texture size is an advantage.

The compressed S3TC formats are based on 4×4 pixel blocks. This means the width and height of non-power-of-two textures compressed with S3TC are rounded up to the nearest multiple of 4. So for example, there is no memory footprint advantage to using a non-mipmapped 13×61 texture compressed with S3TC compared to a similarly compressed non-mipmapped 16×64 texture.

2.1.4.2. Mipmap Construction for Non-Power-of-Two-Textures

The size of each smaller non-power-of-two mipmap level is computed by halving the lower (larger) level's width, height, and depth and rounding down (flooring) to the next smaller integer while never reducing a size to less than one. For example, the level above a 17×10 mipmap level is 8×5. The OpenGL non-power-of-two mipmap reduction convention is identical to that of DirectX 9.

The standard `gluBuild1DMipmaps`, `gluBuild2DMipmaps`, and `gluBuild3DMipmaps` routines accept a non-power-of-two image but automatically rescale the image (using a box filter) to the next largest power-of-two in all dimensions if necessary. If you want to

specify true non-power-of-two mipmapped texture images, these routines should be avoided.

Instead, you can set the GL_GENERATE_MIPMAP texture parameter to GL_TRUE and let the OpenGL driver generate non-power-of-two mipmaps for you automatically.

NVIDIA's OpenGL driver today uses a slow-but-correct recursive box filter (each iteration is equivalent to what gluScaleImage does) when generating non-power-of-two mipmap chains. Expect driver mipmap generation for non-power-of-two textures to be measurably slower than driver mipmap generation for non-power-of-two textures. Future driver releases may optimize non-power-of-two mipmap generation.

Applications using static non-power-of-two textures can reduce time spent generating non-power-of-two mipmap chains by loading pre-computing mipmap chains.

2.1.5. Point Sprites

OpenGL 2.0 introduces a new point sprite mode called GL_POINT_SPRITE_COORD_ORIGIN that can be set to GL_UPPER_LEFT (the default) or GL_LOWER_LEFT. The earlier ARB_point_sprite and NV_point_sprite extensions lack this mode.

When rendering to windows, leave the GL_POINT_SPRITE_COORD_ORIGIN state set to its default GL_UPPER_LEFT setting. Using GL_LOWER_LEFT with windowed rendering will force points to be transformed on the CPU.

When rendering to pixel buffers (commonly called *pbuffers*) or frame buffer objects (commonly called FBOs), change the GL_POINT_SPRITE_COORD_ORIGIN state set to GL_LOWER_LEFT setting for fully hardware accelerated rendering. Using GL_UPPER_LEFT with pbuffer and FBO rendering will force points to be transformed on the CPU.

NVIDIA supports (on all its GPUs) the NV_point_sprite extension that provides one additional point sprite control beyond what OpenGL 2.0 provides. This extension provides an additional GL_POINT_SPRITE_R_MODE_NV that controls how the R texture coordinate is set for points. You have the choice to zero R (GL_ZERO, the default), use the vertex's S coordinate for R prior to S being overridden for the point sprite mode (GL_S), or the vertex's R coordinate (GL_R).

2.1.6. Two-Sided Stencil Testing

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs fully support all OpenGL 2.0 two-sided stencil testing modes.

NVIDIA drivers support both the EXT_stencil_two_side extension and the OpenGL 2.0 functionality. Two sets of back-sided stencil state are maintained. The EXT extension's state is set by glStencil* commands when glActiveStencilFaceEXT is set

to `GL_BACK`. The 2.0 back-facing state is set by the `glStencil*Separate` commands when the `face` parameter is `GL_BACK` (or `GL_FRONT_AND_BACK`). When `GL_STENCIL_TWO_SIDE_EXT` is enabled, the EXT back-facing stencil state takes priority.

2.1.7. Separate RGB and Alpha Blend Equations

NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs fully support *all* OpenGL 2.0 blend modes including separate RGB and alpha blend equations.

2.2. Acceleration for GeForce FX and NV3xGL-based Quadro FX

2.2.1. Fragment-Level Branching

Unlike NVIDIA's GeForce 6 Series and NV4xGL-based Quadro FX GPUs, GeForce FX and NV3xGL-based Quadro FX GPUs do not have hardware support for fragment-level branching.

Apparent support for flow-control constructs in GLSL (and Cg) is based entirely on conditional assignment, unrolling loops, and inlining functions.

The compiler can often generate code for programs with flow control that can be simulated with conditional assignment, unrolling loops, and inlining functions, but for more complex flow control, the program object containing a fragment shader may simply fail to compile. The hardware's branch-free execution model with condition-code instructions is discussed in the `NV_fragment_program` OpenGL extension specification.

2.2.2. Vertex Textures

NVIDIA's GeForce FX and NV3xGL-based Quadro FX GPUs lack hardware support for vertex fetching. GLSL vertex shaders that perform vertex texture fetches will fail to compile.

The implementation-dependent constant `GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS` is advertised as zero (OpenGL 2.0 allows a minimum of zero to be advertised).

Future driver revisions may successfully compile GLSL vertex shaders with texture fetches but perform the vertex shader completely or partially with the CPU. In this case, the GPU can still accelerate rasterization and fragment processing.

2.2.3. Multiple Render Targets

NVIDIA's GeForce FX and NV3xGL-based Quadro FX GPUs can output only a single RGBA color per fragment processed so the maximum number of draw buffers (indicated by `GL_MAX_DRAW_BUFFERS`) is just one.

Effectively, this means GeForce FX and NV3xGL-based Quadro FX GPUs do not support the spirit of multiple render targets. However, OpenGL 2.0 permits an implementation to advertise support for just one draw buffer (see the 1+ minimum for `GL_MAX_DRAW_BUFFERS` in table 6.35 of the OpenGL 2.0 specification).

Applications that desire rendering to multiple rendering targets must resort to multiple rendering passes using `glDrawBuffer` to switch to a different buffer for each pass.

2.2.4. Non-Power-Of-Two Textures

GeForce FX and NV3xGL-based Quadro FX GPUs lack hardware support for non-power-of-two textures (excepting the texture rectangle functionality provided by the `ARB_texture_rectangle` extension). If any texture unit could sample a bound 1D, 2D, 3D, or cube map texture object with non-power-of-two size, the driver will automatically render with software rendering, which is correct but extremely slow.

To determine if non-power-of-two textures are slow, examine the `GL_EXTENSIONS` string. If `GL_VERSION` reports that 2.0 but `GL_ARB_texture_non_power_of_two` is not listed in the `GL_EXTENSIONS` string, assume that using non-power-of-two-textures is slow and avoid the functionality.

The discussion in section 2.1.4.2 about non-power-of-two mipmap discussion apply to GeForce FX and NV3xGL-based Quadro FX GPUs too even if these GPUs do not hardware accelerate non-power-of-two texture rendering.

2.2.5. Point Sprites

GeForce FX and NV3xGL-based Quadro FX GPUs have the identical caveats as the GeForce 6 Series and NV4xGL-based Quadro FX GPUs discussed in section 2.1.5.

2.2.6. Two-Sided Stencil Testing

GeForce FX and NV3xGL-based Quadro FX GPUs have full hardware acceleration for two-sided stencil testing just like the GeForce 6 Series and NV4xGL-based Quadro FX GPUs. The same discussion in section 2.1.6 applies.

2.2.7. Separate RGB and Alpha Blend Equations

GeForce FX and NV3xGL-based Quadro FX GPUs lack hardware support for separate RGB and alpha blend equations. If the RGB and alpha blend equations are different, the driver will automatically render with full software rasterization, which is correct but extremely slow.

To determine if separate blend equations is slow, examine the `GL_EXTENSIONS` string. If `GL_VERSION` reports that 2.0 but `GL_EXT_blend_equation_separate` is not listed in the

GL_EXTENSIONS string, assume that using separate distinct blend equations is slow and avoid the functionality.

3. Programmable Shading API Updates for OpenGL 2.0

The command and token names in the original ARB extensions for programmable shading with GLSL are verbose and used an object model inconsistent with other types of objects (display lists, texture objects, vertex buffer objects, occlusion queries, etc.).

3.1. Type Name Changes

The GLhandleARB type has been deprecated in preference to GLuint for program and shader object names. The underlying type for the GLhandleARB is a 32-bit unsigned integer so the two types have compatible representations.

Old ARB extensions type	New OpenGL 2.0 type
GLhandleARB	GLuint

3.2. Token Name Changes

Old ARB extensions tokens	New OpenGL 2.0 tokens
GL_PROGRAM_OBJECT_ARB	*Unnecessary*
GL_SHADER_OBJECT_ARB	*Unnecessary*
GL_OBJECT_TYPE_ARB	*Instead* glIsProgram *and* glIsShader
GL_OBJECT_SUBTYPE_ARB	GL_SHADER_TYPE
GL_OBJECT_DELETE_STATUS_ARB	GL_DELETE_STATUS
GL_OBJECT_COMPILE_STATUS_ARB	GL_COMPILE_STATUS
GL_OBJECT_LINK_STATUS_ARB	GL_LINK_STATUS
GL_OBJECT_VALIDATE_STATUS_ARB	GL_VALIDATE_STATUS
GL_OBJECT_INFO_LOG_LENGTH_ARB	GL_INFO_LOG_LENGTH
GL_OBJECT_ATTACHED_OBJECTS_ARB	GL_ATTACHED_SHADERS
GL_OBJECT_ACTIVE_UNIFORMS_ARB	GL_ACTIVE_UNIFORMS
GL_OBJECT_ACTIVE_UNIFORM_MAX_LENGTH_ARB	GL_ACTIVE_UNIFORM_MAX_LENGTH
GL_OBJECT_SHADER_SOURCE_LENGTH_ARB	GL_SHADER_SOURCE_LENGTH
No equivalent	GL_CURRENT_PROGRAM

For other ARB_shader_objects, ARB_vertex_shader, and ARB_fragment_shader tokens, the OpenGL 2.0 names are identical to the ARB extension names except without the ARB suffix.

3.3. Command Name Changes

Old ARB extensions commands	New OpenGL 2.0 commands
glAttachObjectARB	glAttachShader
glCreateProgramObjectARB	glCreateProgram
glCreateShaderObjectARB	glCreateShader
glDeleteObjectARB	glDeleteShader for shader objects, glDeleteProgram for program objects
glDetachObjectARB	glDetachShader
glGetAttachedObjectsARB	glGetAttachedShaders
glGetHandleARB	glGetIntegerv(GL_CURRENT_PROGRAM, &retval)
glGetInfoLogARB	glGetShaderInfoLog for shader objects, glGetProgramInfoLog for program objects
glGetObjectParameterfvARB	*No equivalent*
glGetObjectParameterivARB	glGetShaderiv for shader objects, glGetProgramiv for program objects
No equivalent	glIsProgram
No equivalent	glIsShader
glUseProgramObjectARB	glUseProgram

For other `ARB_shader_objects`, `ARB_vertex_shader`, and `ARB_fragment_shader` commands, the OpenGL 2.0 names are identical to the ARB extension names except without the ARB suffix.

4. Correctly Detecting OpenGL 2.0 in Applications

To determine if OpenGL 2.0 or better is supported, an application must parse the implementation-dependent string returned by calling `glGetString(GL_VERSION)`.

4.1. Version String Formatting

OpenGL version strings are laid out as follows:

<version number> <space> <vendor-specific information>

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The *release_number* and *vendor-specific information,* along with its preceding space, are optional. If present, the interpretation of the *release_number* and *vendor-specific information* depends on the vendor.

NVIDIA does not provide *vendor-specific information* but uses the *release_number* to indicate how many NVIDIA major driver releases (counting from zero) have supported this particular major and minor revision of OpenGL. For example, the drivers in the Release 75 series report `2.0.0` indicating Release 75 is the first driver series to support

OpenGL 2.0. Release 80 will likely advertise `2.0.1` for its `GL_VERSION` string. Major NVIDIA graphics driver releases typically increment by 5.

4.2. Proper Version String Parsing

Early application testing by NVIDIA has encountered a few isolated OpenGL applications that incorrectly parse the `GL_VERSION` string when the OpenGL version changed from 1.5 to 2.0.

OpenGL developers are *strongly* urged to examine their application code that parses the `GL_VERSION` string to make sure pairing the application with an OpenGL 2.0 implementation will not confuse or crash the application.

Use the routine below to correctly test if at least a particular major and minor version of OpenGL is available.

```
static int
supportsOpenGLVersion(int atLeastMajor, int atLeastMinor)
{
    const char *version;
    int major, minor;

    version = (const char *) glGetString(GL_VERSION);
    if (sscanf(version, "%d.%d", &major, &minor) == 2) {
        if (major > atLeastMajor)
            return 1;
        if (major == atLeastMajor && minor >= atLeastMinor)
            return 1;
    } else {
        /* OpenGL version string malformed! */
    }
    return 0;
}
```

For example, the above routine returns true if OpenGL 2.0 or better is supported (and false otherwise) when the routine is called like this:

```
int hasOpenGL20 = supportsOpenGLVersion(2, 0);
```

Be sure your OpenGL applications behave correctly with OpenGL 2.0.

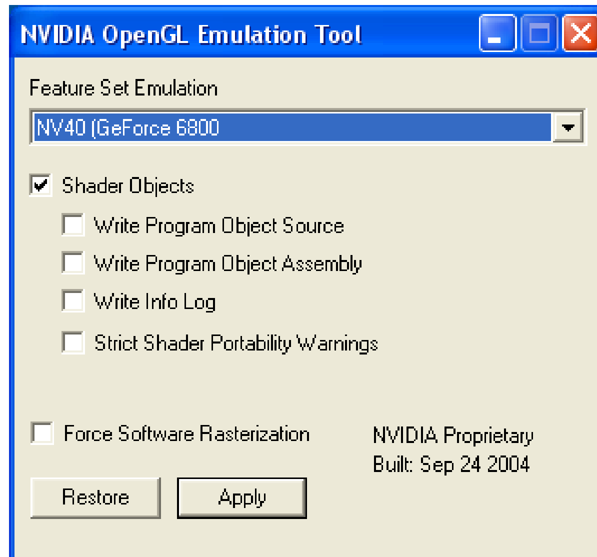
5. Enabling OpenGL 2.0 Emulation on Older GPUs

Developers and students using Microsoft Windows wishing to work with OpenGL 2.0 on pre-NV3x GPUs can use a utility called `nvemulate.exe` to force these older drivers to expose the feature sets of newer GPUs. When forcing emulation of an NV3x or NV4x GPU with a Release 75-based driver, you can expose OpenGL 2.0.

OpenGL features the GPU can support natively will be hardware accelerated as usual. But GPU features not natively supported will be slowly emulated by the OpenGL driver.

OpenGL extensions, implementation-dependent limits, and core OpenGL version for the GPU being emulated will be advertised.

So if you enable "NV40 (GeForce 6800)" emulation, as shown in the image below, on a old GeForce3 GPU, you will see OpenGL 2.0 advertised by the `GL_VERSION` string along with all the NV40 OpenGL extensions listed in the `GL_EXTENSIONS` string and implementation-dependent limits returned by `glGetIntegerv`.



5.1. Programmable Shading Debug Options

Additional check boxes can be enabled and disabled to aid in debugging GLSL applications. The "Shader Objects" check box determines whether the ARB extensions for programmable shading (`ARB_shader_objects`, etc.) are advertised or not.

The "Write Program Object Source" check box causes `vsrc_%u.txt` and `fsrc_%u.txt` files containing the concatenated source string for GLSL shaders to be written to the application's current directory where the %u is `GLuint` value for the shader name.

The "Write Program Object Assembly" check box causes `vasm_%u.txt` and `fasm_%u.txt` files containing the compiled assembly text for linked GLSL program objects to be written to the application's current directory where the %u is `GLuint` value for the program name.

The "Write Info Log" check box causes `ilog_%u.txt` files containing the info log contents for linked GLSL program objects to be written to the application's current directory where the %u is `GLuint` value for the program name.

The "Strict Shader Portability Warnings" causes the compiler to generate portability warnings in the info log output. These warnings are not particularly thorough yet.

5.2. Forcing the Software Rasterizer

With the “Force Software Rasterizer” check box set, the driver does **all** OpenGL rendering in software. If you suspect a driver bug resulting in incorrect rendering, you might try this option and see if the rendering anomaly manifests itself in the software rasterizer. This information is helpful when reporting bugs to NVIDIA.

If the hardware and software rendering paths behave more-or-less identically, it may be an indication that the rendering anomaly is due to your application mis-programming OpenGL state or incorrect expectations about how OpenGL should behave.

6. Key Known Issues

6.1. OpenGL Shading Language Issues

NVIDIA’s GLSL implementation is a work-in-progress and still improving. Current limitations and known bugs are discussed in the *Release Notes for NVIDIA OpenGL Shading Language Support*. Developers should be aware of these key issues:

6.1.1. Noise Functions Always Return Zero

The GLSL standard library contains several noise functions of differing dimensions: `noise1`, `noise2`, `noise3`, and `noise4`.

NVIDIA’s implementation of these functions (currently) always returns zero results.

6.1.2. Vertex Attribute Aliasing

GLSL attempts to eliminate aliasing of vertex attributes but this is integral to NVIDIA’s hardware approach and necessary for maintaining compatibility with existing OpenGL applications that NVIDIA customers rely on.

NVIDIA’s GLSL implementation therefore does not allow built-in vertex attributes to collide with a generic vertex attributes that is assigned to a particular vertex attribute index with `glBindAttribLocation`. For example, you should not use `gl_Normal` (a built-in vertex attribute) and also use `glBindAttribLocation` to bind a generic vertex attribute named “whatever” to vertex attribute index 2 because `gl_Normal` aliases to index 2.

This table below summarizes NVIDIA's vertex attribute aliasing behavior:

Built-in vertex attribute name	Incompatible aliased vertex attribute index
gl_Vertex	0
gl_Normal	2
gl_Color	3
gl_SecondaryColor	4
gl_FogCoord	5
gl_MultiTexCoord0	8
gl_MultiTexCoord1	9
gl_MultiTexCoord2	10
gl_MultiTexCoord3	11
gl_MultiTexCoord4	12
gl_MultiTexCoord5	13
gl_MultiTexCoord6	14
gl_MultiTexCoord7	15

Vertex attribute aliasing is also explained in the `ARB_vertex_program` and `NV_vertex_program` specifications.

6.1.3. gl_FrontFacing Is Not Available to Fragment Shaders

The built-in fragment shader varying parameter `gl_FrontFacing` is supported by GeForce 6 Series and NV4xGL-based Quadro FX GPUs but not GeForce FX and NV3xGL-based Quadro FX GPUs.

As a workaround, enable with `glEnable` the `GL_VERTEX_PROGRAM_TWO_SIDE` mode and, in your vertex shader, write a 1 to the alpha component of the front-facing primary color (`gl_FrontColor`) and 0 to the alpha component of the back-facing primary color (`gl_BackColor`). Then, read alpha component of the built-in fragment shader varying parameter `gl_Color`. Just like `gl_FrontFacing`, 1 means front-facing; 0 means back-facing.

6.1.4. Reporting GLSL Issues and Bugs

NVIDIA welcomes email pertaining to GLSL. Send suggestions, feedback, and bug reports to glsl-support@nvidia.com

7. OpenGL 2.0 API Declarations

NVIDIA provides `<GL/gl.h>` and `<GL/glext.h>` header files with the necessary OpenGL 2.0 API declarations on the OpenGL 2.0 page in NVIDIA's Developer web site, `developer.nvidia.com`

Your OpenGL 2.0 application will need to call `wglGetProcAddress` (Windows) or `glXGetProcAddress` (Linux) to obtain function pointers to the new OpenGL 2.0 commands just as is necessary for other OpenGL extensions.

7.1. Programmable Shading

7.1.1. New Tokens Defines

7.1.1.1. Program and Shader Object Management

```
#define GL_CURRENT_PROGRAM                 0x8B8D
#define GL_SHADER_TYPE                     0x8B4E
#define GL_DELETE_STATUS                   0x8B80
#define GL_COMPILE_STATUS                  0x8B81
#define GL_LINK_STATUS                     0x8B82
#define GL_VALIDATE_STATUS                 0x8B83
#define GL_INFO_LOG_LENGTH                 0x8B84
#define GL_ATTACHED_SHADERS                0x8B85
#define GL_ACTIVE_UNIFORMS                 0x8B86
#define GL_ACTIVE_UNIFORM_MAX_LENGTH       0x8B87
#define GL_SHADER_SOURCE_LENGTH            0x8B88
#define GL_VERTEX_SHADER                   0x8B31
#define GL_ACTIVE_ATTRIBUTES               0x8B89
#define GL_ACTIVE_ATTRIBUTE_MAX_LENGTH     0x8B8A
#define GL_FRAGMENT_SHADER                 0x8B30
```

7.1.1.2. Uniform Types

```
#define GL_FLOAT_VEC2                      0x8B50
#define GL_FLOAT_VEC3                      0x8B51
#define GL_FLOAT_VEC4                      0x8B52
#define GL_INT_VEC2                        0x8B53
#define GL_INT_VEC3                        0x8B54
#define GL_INT_VEC4                        0x8B55
#define GL_BOOL                            0x8B56
#define GL_BOOL_VEC2                       0x8B57
#define GL_BOOL_VEC3                       0x8B58
#define GL_BOOL_VEC4                       0x8B59
#define GL_FLOAT_MAT2                      0x8B5A
#define GL_FLOAT_MAT3                      0x8B5B
#define GL_FLOAT_MAT4                      0x8B5C
#define GL_SAMPLER_1D                      0x8B5D
#define GL_SAMPLER_2D                      0x8B5E
#define GL_SAMPLER_3D                      0x8B5F
#define GL_SAMPLER_CUBE                    0x8B60
#define GL_SAMPLER_1D_SHADOW               0x8B61
#define GL_SAMPLER_2D_SHADOW               0x8B62
```

7.1.1.3. Vertex Attrib Arrays

```
#define GL_VERTEX_ATTRIB_ARRAY_ENABLED     0x8622
#define GL_VERTEX_ATTRIB_ARRAY_SIZE        0x8623
#define GL_VERTEX_ATTRIB_ARRAY_STRIDE      0x8624
#define GL_VERTEX_ATTRIB_ARRAY_TYPE        0x8625
#define GL_VERTEX_ATTRIB_ARRAY_NORMALIZED 0x886A
#define GL_CURRENT_VERTEX_ATTRIB           0x8626
#define GL_VERTEX_ATTRIB_ARRAY_POINTER     0x8645
```

```
#define GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING 0x889F
```

7.1.1.4. Hints

```
#define GL_FRAGMENT_SHADER_DERIVATIVE_HINT 0x8B8B
```

7.1.1.5. Enables for Rasterization Control

```
#define GL_VERTEX_PROGRAM_POINT_SIZE       0x8642
#define GL_VERTEX_PROGRAM_TWO_SIDE         0x8643
```

7.1.1.6. Implementation Dependent Strings and Limits

```
#define GL_SHADING_LANGUAGE_VERSION        0x8B8C
#define GL_MAX_VERTEX_ATTRIBS              0x8869
#define GL_MAX_FRAGMENT_UNIFORM_COMPONENTS 0x8B49
#define GL_MAX_VERTEX_UNIFORM_COMPONENTS   0x8B4A
#define GL_MAX_VARYING_FLOATS              0x8B4B
#define GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS  0x8B4C
#define GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS 0x8B4D
#define GL_MAX_TEXTURE_COORDS              0x8871
#define GL_MAX_TEXTURE_IMAGE_UNITS         0x8872
```

7.1.2. New Command Prototypes

7.1.2.1. Shader Objects

```
void GLAPI glDeleteShader (GLuint shader);
void GLAPI glDetachShader (GLuint program, GLuint shader);
GLuint GLAPI glCreateShader (GLenum type);
void GLAPI glShaderSource (GLuint shader, GLsizei count, const GLchar* *string, const
GLint *length);
void GLAPI glCompileShader (GLuint shader);
```

7.1.2.2. Program Objects

```
GLuint GLAPI glCreateProgram (void);
void GLAPI glAttachShader (GLuint program, GLuint shader);
void GLAPI glLinkProgram (GLuint program);
void GLAPI glUseProgram (GLuint program);
void GLAPI glDeleteProgram (GLuint program);
void GLAPI glValidateProgram (GLuint program);
```

7.1.2.3. Uniforms

```
void GLAPI glUniform1f (GLint location, GLfloat v0);
void GLAPI glUniform2f (GLint location, GLfloat v0, GLfloat v1);
void GLAPI glUniform3f (GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void GLAPI glUniform4f (GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat
v3);
void GLAPI glUniform1i (GLint location, GLint v0);
void GLAPI glUniform2i (GLint location, GLint v0, GLint v1);
void GLAPI glUniform3i (GLint location, GLint v0, GLint v1, GLint v2);
void GLAPI glUniform4i (GLint location, GLint v0, GLint v1, GLint v2, GLint v3);
void GLAPI glUniform1fv (GLint location, GLsizei count, const GLfloat *value);
```

```
void GLAPI glUniform2fv (GLint location, GLsizei count, const GLfloat *value);
void GLAPI glUniform3fv (GLint location, GLsizei count, const GLfloat *value);
void GLAPI glUniform4fv (GLint location, GLsizei count, const GLfloat *value);
void GLAPI glUniform1iv (GLint location, GLsizei count, const GLint *value);
void GLAPI glUniform2iv (GLint location, GLsizei count, const GLint *value);
void GLAPI glUniform3iv (GLint location, GLsizei count, const GLint *value);
void GLAPI glUniform4iv (GLint location, GLsizei count, const GLint *value);
void GLAPI glUniformMatrix2fv (GLint location, GLsizei count, GLboolean transpose,
const GLfloat *value);
void GLAPI glUniformMatrix3fv (GLint location, GLsizei count, GLboolean transpose,
const GLfloat *value);
void GLAPI glUniformMatrix4fv (GLint location, GLsizei count, GLboolean transpose,
const GLfloat *value);
```

7.1.2.4. Attribute Locations

```
void GLAPI glBindAttribLocation (GLuint program, GLuint index, const GLchar *name);
GLint GLAPI glGetAttribLocation (GLuint program, const GLchar *name);
```

7.1.2.5. Vertex Attributes

```
void GLAPI glVertexAttrib1d (GLuint index, GLdouble x);
void GLAPI glVertexAttrib1dv (GLuint index, const GLdouble *v);
void GLAPI glVertexAttrib1f (GLuint index, GLfloat x);
void GLAPI glVertexAttrib1fv (GLuint index, const GLfloat *v);
void GLAPI glVertexAttrib1s (GLuint index, GLshort x);
void GLAPI glVertexAttrib1sv (GLuint index, const GLshort *v);
void GLAPI glVertexAttrib2d (GLuint index, GLdouble x, GLdouble y);
void GLAPI glVertexAttrib2dv (GLuint index, const GLdouble *v);
void GLAPI glVertexAttrib2f (GLuint index, GLfloat x, GLfloat y);
void GLAPI glVertexAttrib2fv (GLuint index, const GLfloat *v);
void GLAPI glVertexAttrib2s (GLuint index, GLshort x, GLshort y);
void GLAPI glVertexAttrib2sv (GLuint index, const GLshort *v);
void GLAPI glVertexAttrib3d (GLuint index, GLdouble x, GLdouble y, GLdouble z);
void GLAPI glVertexAttrib3dv (GLuint index, const GLdouble *v);
void GLAPI glVertexAttrib3f (GLuint index, GLfloat x, GLfloat y, GLfloat z);
void GLAPI glVertexAttrib3fv (GLuint index, const GLfloat *v);
void GLAPI glVertexAttrib3s (GLuint index, GLshort x, GLshort y, GLshort z);
void GLAPI glVertexAttrib3sv (GLuint index, const GLshort *v);
void GLAPI glVertexAttrib4Nbv (GLuint index, const GLbyte *v);
void GLAPI glVertexAttrib4Niv (GLuint index, const GLint *v);
void GLAPI glVertexAttrib4Nsv (GLuint index, const GLshort *v);
void GLAPI glVertexAttrib4Nub (GLuint index, GLubyte x, GLubyte y, GLubyte z, GLubyte
w);
void GLAPI glVertexAttrib4Nubv (GLuint index, const GLubyte *v);
void GLAPI glVertexAttrib4Nuiv (GLuint index, const GLuint *v);
void GLAPI glVertexAttrib4Nusv (GLuint index, const GLushort *v);
void GLAPI glVertexAttrib4bv (GLuint index, const GLbyte *v);
void GLAPI glVertexAttrib4d (GLuint index, GLdouble x, GLdouble y, GLdouble z,
GLdouble w);
void GLAPI glVertexAttrib4dv (GLuint index, const GLdouble *v);
void GLAPI glVertexAttrib4f (GLuint index, GLfloat x, GLfloat y, GLfloat z, GLfloat
w);
void GLAPI glVertexAttrib4fv (GLuint index, const GLfloat *v);
void GLAPI glVertexAttrib4iv (GLuint index, const GLint *v);
void GLAPI glVertexAttrib4s (GLuint index, GLshort x, GLshort y, GLshort z, GLshort
w);
void GLAPI glVertexAttrib4sv (GLuint index, const GLshort *v);
void GLAPI glVertexAttrib4ubv (GLuint index, const GLubyte *v);
void GLAPI glVertexAttrib4uiv (GLuint index, const GLuint *v);
void GLAPI glVertexAttrib4usv (GLuint index, const GLushort *v);
```

```
void GLAPI glVertexAttribPointer (GLuint index, GLint size, GLenum type, GLboolean
normalized, GLsizei stride, const GLvoid *pointer);
void GLAPI glEnableVertexAttribArray (GLuint index);
void GLAPI glDisableVertexAttribArray (GLuint index);
void GLAPI glGetVertexAttribdv (GLuint index, GLenum pname, GLdouble *params);
void GLAPI glGetVertexAttribfv (GLuint index, GLenum pname, GLfloat *params);
void GLAPI glGetVertexAttribiv (GLuint index, GLenum pname, GLint *params);
void GLAPI glGetVertexAttribPointerv (GLuint index, GLenum pname, GLvoid* *pointer);
```

7.1.2.6. Queries

```
GLboolean GLAPI glIsShader (GLuint shader);
GLboolean GLAPI glIsProgram (GLuint program);
void GLAPI glGetShaderiv (GLuint program, GLenum pname, GLint *params);
void GLAPI glGetProgramiv (GLuint program, GLenum pname, GLint *params);
void GLAPI glGetAttachedShaders (GLuint program, GLsizei maxCount, GLsizei *count,
GLuint *shaders);
void GLAPI glGetShaderInfoLog (GLuint shader, GLsizei bufSize, GLsizei *length, GLchar
*infoLog);
void GLAPI glGetProgramInfoLog (GLuint program, GLsizei bufSize, GLsizei *length,
GLchar *infoLog);
GLint GLAPI glGetUniformLocation (GLuint program, const GLchar *name);
void GLAPI glGetActiveUniform (GLuint program, GLuint index, GLsizei bufSize, GLsizei
*length, GLsizei *size, GLenum *type, GLchar *name);
void GLAPI glGetUniformfv (GLuint program, GLint location, GLfloat *params);
void GLAPI glGetUniformiv (GLuint program, GLint location, GLint *params);
void GLAPI glGetShaderSource (GLuint shader, GLsizei bufSize, GLsizei *length, GLchar
*source);
void GLAPI glGetActiveAttrib (GLuint program, GLuint index, GLsizei bufSize, GLsizei
*length, GLsizei *size, GLenum *type, GLchar *name);
```

7.2. Non-Power-Of-Two Textures

Support for non-power-of-two textures introduces no new tokens or commands. Rather the error conditions that previously restricted the width, height, and depth (excluding the border) to be power-of-two values is eliminated.

The relaxation of errors to allow non-power-of-two texture sizes affects the following commands: `glTexImage1D`, `glCopyTexImage1D`, `glTexImage2D`, `glCopyTexImage2D`, and `glTexImage3D`. You can also render to non-power-of-two pixel buffers (*pbuffers*) using the `WGL_ARB_render_texture` extension.

7.3. Multiple Render Targets

7.3.1. New Tokens Defines

```
#define GL_MAX_DRAW_BUFFERS                    0x8824
#define GL_DRAW_BUFFER0                        0x8825
#define GL_DRAW_BUFFER1                        0x8826
#define GL_DRAW_BUFFER2                        0x8827
#define GL_DRAW_BUFFER3                        0x8828
#define GL_DRAW_BUFFER4                        0x8829
#define GL_DRAW_BUFFER5                        0x882A
#define GL_DRAW_BUFFER6                        0x882B
#define GL_DRAW_BUFFER7                        0x882C
```

```
#define GL_DRAW_BUFFER8                     0x882D
#define GL_DRAW_BUFFER9                     0x882E
#define GL_DRAW_BUFFER10                    0x882F
#define GL_DRAW_BUFFER11                    0x8830
#define GL_DRAW_BUFFER12                    0x8831
#define GL_DRAW_BUFFER13                    0x8832
#define GL_DRAW_BUFFER14                    0x8833
#define GL_DRAW_BUFFER15                    0x8834
```

7.3.2. New Command Prototypes

```
void GLAPI glDrawBuffers (GLsizei n, const GLenum *bufs);
```

7.4. Point Sprite

7.4.1. New Tokens Defines

```
#define GL_POINT_SPRITE                     0x8861
#define GL_COORD_REPLACE                    0x8862
#define GL_POINT_SPRITE_COORD_ORIGIN        0x8CA0
#define GL_LOWER_LEFT                       0x8CA1
#define GL_UPPER_LEFT                       0x8CA2
```

7.4.2. Usage

Point sprite state is set with the `glPointParameteri`, `glPointParameteriv`, `glPointParameterf`, `glPointParameterfv` API originally introduced by OpenGL 1.4 to control point size attenuation.

7.5. Two-Sided Stencil Testing

7.5.1. New Tokens Defines

These tokens can be used with `glGetIntegerv` to query back-facing stencil state.

```
#define GL_STENCIL_BACK_FUNC                0x8800
#define GL_STENCIL_BACK_VALUE_MASK          0x8CA4
#define GL_STENCIL_BACK_REF                 0x8CA3
#define GL_STENCIL_BACK_FAIL                0x8801
#define GL_STENCIL_BACK_PASS_DEPTH_FAIL     0x8802
#define GL_STENCIL_BACK_PASS_DEPTH_PASS     0x8803
#define GL_STENCIL_BACK_WRITEMASK           0x8CA5
```

7.5.2. New Command Prototypes

```
void GLAPI glStencilFuncSeparate (GLenum face, GLenum func, GLint ref, GLuint mask);
void GLAPI glStencilOpSeparate (GLenum face, GLenum fail, GLenum zfail, GLenum zpass);
void GLAPI glStencilMaskSeparate (GLenum face, GLuint mask);
```

7.6. Separate RGB and Alpha Blend Equations

7.6.1. New Tokens Defines

These tokens can be used with `glGetIntegerv` to query blend equation state. The `GL_BLEND_EQUATION` token has the same value as the new `GL_BLEND_EQUATION_RGB`.

```
#define GL_BLEND_EQUATION_RGB            0x8009
#define GL_BLEND_EQUATION_ALPHA          0x883D
```

7.6.2. New Command Prototypes

```
void GLAPI glBlendEquationSeparate (GLenum modeRGB, GLenum modeAlpha);
```

A. Distinguishing NV3xGL-based and NV4xGL-based Quadro FX GPUs by Product Names

As discussed in section 2, while NV3x- and NV3xGL-based GPUs support OpenGL 2.0, the NV4x- and NV4xGL-based GPUs have the best industry-wide hardware-acceleration and support for OpenGL 2.0.

For the consumer GeForce product lines, GeForce FX and GeForce 6 Series GPUs are easily distinguished based on their product names and numbering. Any NVIDIA GPU product beginning with GeForce FX is NV3x-based. Such GPUs also typically have a 5000-based product number, such as 5200 or 5950. GeForce GPUs with a 6000-based product name, such as 6600 or 6800, are NV4x-based.

However, the Quadro FX product name applies to both NV3xGL-based and NV4xGL-based GPUs and there is no simple rule to differentiate NV3xGL-based and NV4xGL-based using the product name. The lists below will help OpenGL 2.0 developers correctly distinguish the two NV3xGL- and NV4xGL-based Quadro FX product lines.

A.1. NV3xGL-based Quadro FX GPUs

Quadro FX 330 (PCI Express)
Quadro FX 500 (AGP)
Quadro FX 600 (PCI)
Quadro FX 700 (AGP)
Quadro FX 1000 (AGP)
Quadro FX 1100 (AGP)
Quadro FX 1300 (PCI)
Quadro FX 2000 (AGP)
Quadro FX 3000 (AGP)

A.2. NV4xGL-based Quadro FX GPUs

Quadro FX 540 (PCI Express)
Quadro FX 1400 (PCI Express)
Quadro FX Go1400 (PCI Express)
Quadro FX 3400 (PCI Express)
Quadro FX 4000 (AGP)
Quadro FX 4400 (PCI Express)
Quadro FX 3450 (PCI Express)
Quadro FX 4450 (PCI Express)

A.3. How to Use and How to Not Use These Lists

These lists are for informational purposes to help OpenGL 2.0 developers instruct end-users as to which NVIDIA products will support OpenGL 2.0 and accelerate the OpenGL 2.0 feature set the best. These lists are not complete and are subject to change.

OpenGL developers should **not** query and parse the `GL_RENDERER` string returned by `glGetString` to determine if a given GPU supports NV3x-based or NV4x-based functionality and performance.

Instead, query the `GL_EXTENSIONS` string and look for the `GL_NV_fragment_program2` and/or `GL_NV_vertex_program3` substrings. These extensions imply the functional feature set of NV4x-based GPUs.