# NVIDIA OpenGL 2.0 Support

Mark J. Kilgard
April 26, 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



## 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,<br>`glDeleteProgram` for program objects |
| `glDetachObjectARB` | `glDetachShader` |
| `glGetAttachedObjectsARB` | `glGetAttachedShaders` |
| `glGetHandleARB` | `glGetIntegerv(GL_CURRENT_PROGRAM, &retval)` |
| `glGetInfoLogARB` | `glGetShaderInfoLog` for shader objects,<br>`glGetProgramInfoLog` for program objects |
| `glGetObjectParameterfvARB` | *No equivalent* |
| `glGetObjectParameterivARB` | `glGetShaderiv` for shader objects,<br>`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 when the `GL_VERSION` string changes.

### 4.2.1. Warning about Proper Extensions String Parsing Too

This is also a good opportunity for OpenGL developers to re-examine the code they use to detect OpenGL extensions. Be sure your code makes no assumption about the length of the string returned by `glGetString(GL_EXTENSIONS)`. Almost every major OpenGL driver release adds new OpenGL extensions as does every major hardware generation so you can expect this string to grow in length over time. If you `strcpy` the `GL_EXTENSIONS` string into static buffer with only 8192 bytes, you *will* get burned eventually.

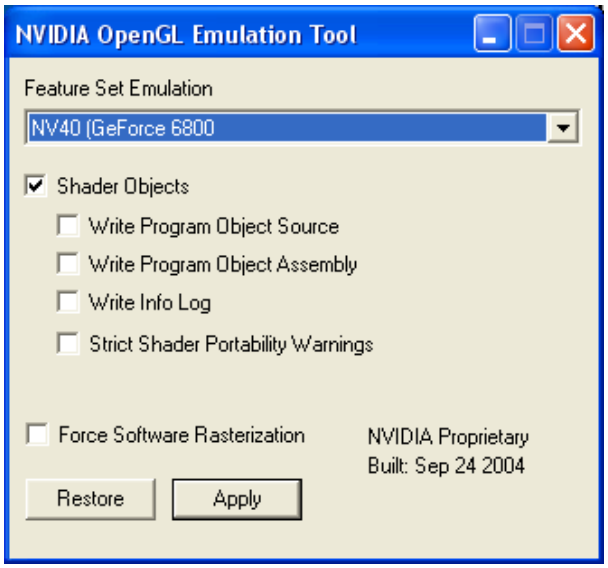
One recurring mistake is passing the string returned by `glGetString(GL_EXTENSIONS)` to a `printf`-like or GUI routine that has an internal buffer of a limited size. For example, an application might display the OpenGL extensions string in an "About Renderer…" dialog box. The routine to set the dialog box string may assume that no one would ever specify a string longer than 1,000 characters.

# 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. Known Issues

## *6.1. OpenGL Shading Language Issues*

NVIDIA's GLSL implementation is a work-in-progress and still improving. Various limitations and known bugs are discussed in the *Release Notes for NVIDIA OpenGL Shading Language Support*.

### 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 Token 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 Token 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 Token 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 Token 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 Token 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.