Release Notes for NVIDIA OpenGL Shading Language Support

November 9, 2006

These release notes explain the implementation status of the OpenGL Shading Language (GLSL) functionality in NVIDIA's Release 95 drivers.

First, these notes discuss the status of the GLSL standard and NVIDIA's commitment to GLSL. Second, these notes explain levels of GLSL functionality in Release 95 drivers and how to enable advanced functionality using the NVemulate control panel. Third, these notes discuss unresolved issues, caveats, and extensions for NVIDIA's GLSL implementation.

Status of the GLSL Standard

The original GLSL standard consisted of three OpenGL extensions and a shading language specification. These specifications have been approved by the OpenGL Architectural Review Board.

The OpenGL 2.0 specification incorporated a version of these extensions into the core OpenGL standard. During this process, the names of functions were slightly changed, and the ARB suffix was dropped from function names and tokens. Also the type for shader and program objects was changed to `GLuint` from `GLhandleARB`, but the underlying data type remains an unsigned 32-bit integer.

The OpenGL 2.1 specification included the 1.2 version of the shading language which added additional capabilities to the shading language. The changes included in this version of the shading language include additional type conversion rules, better initializer support, and additional types. The changes are available by placing "#version 120" at the beginning of a shader.

In addition to the base GLSL 1.2 version, the extensions GL_EXT_gpu_shader4, GL_EXT_geometry_shader4, and GL_EXT_bindable_uniform have been adopted by multiple vendors. These extensions extend GLSL with fourth generation shading capabilities including bit-wise integer operations, shaders that can operate on entire primitives, and additional texture functions. These capabilities are appearing in beta form with this driver, so they must be enabled by turning on the G80 compiler target as described later in this document.

NVIDIA's Support for Multiple Shading Languages

NVIDIA supports an array of shading languages in addition to GLSL. Among these are NVIDIA's Cg and Microsoft's HLSL. All are actively developed and supported on NVIDIA platforms, and all have associated advantages and drawbacks. NVIDIA utilizes

a Unified Compiler Architecture (UCA) to produce the final hardware binaries for all these languages. For the latest Cg specific information, one should query the Cg documents on NVIDIA's developer website.

Shading Language Features

	GLSL	**Cg**	**HLSL**
Supports OpenGL	Yes	Yes	No
Supports FX files	No	Yes	Yes
Supports D3D	No	Yes	Yes
Multivendor	Yes	No*	Yes

* - Cg supports non-NVIDIA platforms through the generation of other standard languages

Release 95 Driver Support for GLSL

The GLSL extensions and OpenGL 2.1 are advertised by default when a Release 95 driver is installed on a Windows or Linux PC.

You can enable the enhanced GLSL capabilities of the GeForce 8 series hardware using a special control panel called NVemulate (`nvemulate.exe`) that you can obtain from the NVIDIA Developers website. These enhancements include the GeForce 8 series compilation profiles and associated extensions. On pre-GeForce 8 series hardware, this also requires enabling feature set emulation, and the newer feature will result in software emulation. These GeForce 8 series enhancements are considered to be in beta form, and once sufficient testing has completed, they will be advertised by default like the rest of GLSL.

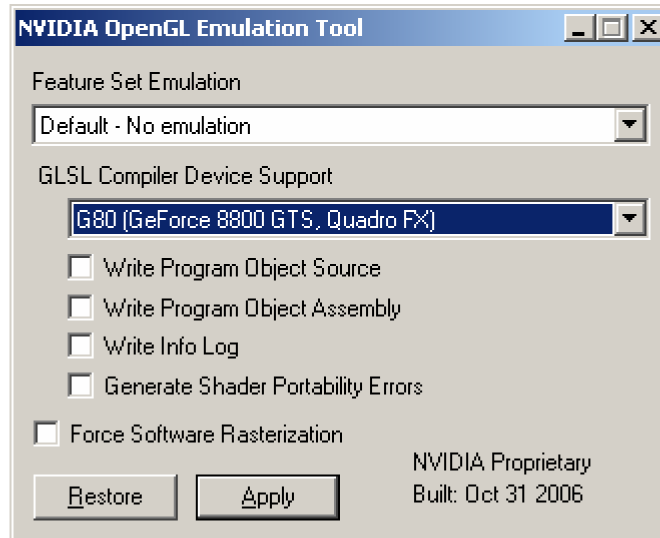
In addition to these new features, the Release 95 drivers also improve adherence to the OpenGL specification. Whenever #version 110 or #version 120 is found at the beginning of a shader, it is compiled such that items formerly flagged as portability warnings are now marked as errors. This means that some shaders using #version 110 that did not strictly conform to the OpenGL specification may no longer compile with theses drivers. Shaders that do not specify #version 110 or #version 120 will continue to compile as before. Additional information on which features these changes effect can be found in the section "NVIDIA's GLSL Enhancements" later in this document.

NVIDIA is committed to GLSL support for Linux. NVIDIA's Release 95 Linux drivers support GLSL in the manner described in this document.

Controlling GLSL Support with NVemulate

GLSL support is enabled by default in Release 95 drivers, but you can enable/disable the enhanced GLSL support as well as manipulate software emulation of GPU feature sets and control the dumping of GLSL source and assembly text for debugging,

The NVemulate control panel looks (something) like this:



To enable the enhanced GLSL functionality, select the "G80(GeForce 8800 GTS, Quadro FX)" option from the GLSL compiler device support drop box *and* press the "Apply" button. No changes to NVemulate settings take effect until the "Apply" button is pressed. Pressing the "Restore" button reverts the display settings to the driver's current applied state.

With the G80 functionality enabled, the driver will export an additional set of extension strings exposing the enhanced capabilities. The extensions string will now include GL_EXT_gpu_shader4, GL_EXT_geometry_shader4, and GL_EXT_bindable_uniform. Additionally, all GLSL shaders not using these capabilities will now be compiled to the more capable G80 target. This enables better dynamic flow control and array indexing, among other benefits.

As noted earlier, some graphics cards are not capable of GLSL or enhanced GLSL in hardware, so this may require enabling a higher hardware emulation level. (A minimum of NV30 for GLSL and a minimum of G80 for the enhancements)

Controlling GLSL Support on Linux, Solaris, and FreeBSD

As NVemulate is not available on Linux, Solaris, or FreeBSD, one must manually edit the X configuration file. To enable the enhanced GLSL support on Linux, add the following option to the Screen or Device section in the X configuration file:

Option "RegistryDwords" "8F773984=7"

For more on modifying your X configuration file, please see the XF86Config or xorg.conf manpages, or the "Configuring X for the NVIDIA Driver" section of the README:

http://us.download.nvidia.com/XFree86/Linux-x86/1.0-9742/README/chapter-03.html

The enhanced GLSL support for Linux, Solaris, and FreeBSD is only available with driver version 1.0-9742 and later. This driver version can be found on the nzone website at the link below:

http://www.nzone.com/object/nzone_downloads_rel70betadriver.html

NVemulate Troubleshooting

If you do not see the enhanced extensions advertised, make sure of the following:

- You are using a Release 95 driver, preferably version 97.02 or later on Windows or version 9742 or later on Linux, Solaris, or FreeBSD
    - Check this by going to the NVIDIA Control Panel, selecting "Help", then "System Information". In the "Details" panel, the "ForceWare version" should be 97.02 or better.
    - You can also check this by making sure the version string returned by `glGetString(GL_VERSION)` is 2.1.0 or better.
    - You are using a Quadro or GeForce card

If you installed a new NVIDIA graphics driver, the GLSL-related settings are reset to their defaults (GeForce 6/7 level GLSL support) so you must enable G80 GLSL again with NVemulate.

NVemulate Options to Aid Debugging

Other check boxes in the NVemulate control panel are designed to aid debugging of GLSL programs as well as help developers report GLSL issues so they can be readily addressed in future drivers.

Writing Program Object Assembly

When the "Write Program Object Assembly" box is checked, the driver outputs `fasm_%d_%d.txt` and `vasm_%d_%d.txt` files into the application's working directory when fragment and vertex shaders respectively are linked where the first `%d` represents the application's handle number for the linked program object, and the second %d is a unique integer representing this link instance. These files are in a form similar to what the standalone Cg compiler (`cgc.exe`) outputs. The assembly text conforms to one of the vertex or fragment program extensions depending on the shader type and hardware

capabilities. For G80 targeted shaders, the output will be in the NV_gpu_program4 family of languages.

Before you submit a GLSL bug report to NVIDIA, double check that the assembly generated for your successfully linked program object matches what you expect. Sometimes you may find bugs in your own shader source code by seeing how the driver's Cg compiler technology translated your shader into assembly form. If the assembly seems correct or it clearly does not correspond to your high-level shader source, please include the assembly output with your bug report.

Writing Program Object Source

When the "Write Program Object Source" box is checked, the driver outputs `fsrc_%d_%d_%d.txt` and `vsrc_%d_%d_%d.txt` files into the application's working directory when fragment and vertex shaders respectively are linked where the first `%d` represents the application's handle number for the linked program object, the second `%d` is the shader object handle, and the third `%d` is a unique integer representing this link instance. These files contain the concatenation of all your shader object source text. This is the actual GLSL source text that is compiled and linked.

These files are generated whether or not the program object links successfully.

Check that your source code text has been properly transferred to the driver. If you still suspect an OpenGL driver bug, please send the source and assembly files.

Writing Info Log

When the "Write Info Log" box is checked, the driver outputs `ilog_%d.txt` files when a program object is linked where the `%d` represents the handle number for application's program object being linked. The info log for a program object contains both vertex and fragment shader related errors so a single `ilog_%d_%d.txt` file is generated per program object and link instance.

If this file is empty (zero length), that typically means no errors were generated and the program object linked successfully.

Check your source code and fix any errors reported in the info log that reflect errors or warnings in your shader source code. If you still cannot resolve the messages in the info log, please send the info log, source, and assembly files.

Forcing Software Rasterization

Driver bugs in GLSL programs may be due to the compiler translating your shader incorrectly into a GPU executable form or the problem may be a more basic driver bug. One way to get a "second opinion" about the behavior of your application is to force software rasterization. In this mode, all OpenGL rendering is done with the CPU rather

than the GPU. This is extremely slow, particularly when per-fragment programmability is involved. However, if the results of hardware rendering and the software rasterizer do not match, that is an important clue as to the nature of a possible driver bug.

Additionally, if the hardware rendering and the software rasterizer results match, you may want to review once more whether the problem lies with your shader source before reporting a bug.

Strict Shader Portability Warnings

In the past, NVIDIA has provided extensions to GLSL to make the language more compatible with other high-level shading languages. While this can be convenient, it also can cause portability problems. To aid developers in writing portable code, NVIDIA now flags the usage of these capabilities with warnings by default. This behavior can be strengthened by checking the "Generate Shader Portability Errors" checkbox in NVemulate. Additionally, with version 1.20, the GLSL spec has adopted many of these language extensions. As a result, NVIDIA has further tightened the syntax checking on the compiler if "#version 120" or "#version 110" are specified in the shader, such that the use of language extensions not included in version 1.20 will be flagged as an error.

NVIDIA's GLSL Limitations

Various limitations and caveats of NVIDIA's current GLSL implementation are discussed. These limitations will be addressed in future driver releases unless otherwise noted.

Linking by Concatenation

GLSL provides for multiple shader objects to be created, assigned GLSL source text, compiled, be attached to a program object, and then link the program object.

NVIDIA's current driver doesn't fully compile shader objects until the program object link. At this time all the source for a single target is concatenated and then compiled.

This means (currently) there is no efficiency from compiling shader objects once and linking them in multiple program objects. Unlike earlier drivers, the code will be parsed and syntax checked during the compile phase to allow the immediate reporting of errors. Some errors may still be deferred until link, but most should be available at compile time.

gl_FrontFacing Is Not Available on all Platforms

The built-in fragment shader varying parameter `gl_FrontFacing` is not supported on all hardware. It is only available when compiling for NV40 and later targets.

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.

Flow Control Limitations and Caveats

NVIDIA's GLSL implementation does not (currently) provide for arbitrary branching and looping in the fragment domain on the NV30 and NV40 targets. Branching and looping is more general in the vertex domain. Loops will perform best when the iteration count can be determined statically at compile time. Loops based on uniforms are restricted to 255 iterations on NV40, and they are not supported on NV30. It may be best to compile multiple versions with constant loop counts rather than attempting to use a uniform value to control the loop count. The G80 target is significantly more capable with branching and dynamic flow control, so it does not face these caveats.

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.

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

The compiler will automatically assign vertex shader attribute variables not pre-assigned by `glBindAttribLocation` to locations that do not collide with any built-in attribute variables used by the vertex shader. The assigned locations can be queries with `glGetAttribLocation`. This means that a developer only needs to worry about collisions when they are explicitly requesting an attribute to be bound to a specific location.

NVIDIA's GLSL Enhancements

Cg Data Types Supported

GLSL supports vector data types of the form `vec2`, `vec3`, `vec4`.

Instead, Cg and HLSL use the vector data types names `float2`, `float3`, and `float4`. Similarly, Cg supports half-precision floating-point data types using the names `half`, `half2`, `half3`, and `half4`. Cg also provides fixed-point data types using the names `fixed`, `fixed2`, `fixed3`, and `fixed4`. The scalar `half` and `fixed` data types are not guaranteed to be a particular size, but the `half` data type must have as much or less floating-point range and precision as the float data type. The `fixed` data type must be signed, have at least 10 bits of fractional precision, and a range of at least [-2,2). The `fixed` data type may be implemented with floating-point. In fact, the vertex domain implements both the `half` and `fixed` data types the same as `float`. Using the half and fixed data types when acceptable can greatly improve the performance of fragment shaders on some platforms. (Most notable NV30 and NV40) These data types are particularly useful for quantities such as colors, blend factors, and normalized vectors that tend to have bounded range and precision requirements.

GLSL versions 1.00 and 1.10 support matrix data types of the form `mat2`, `mat3`, `mat4`. GLSL version 1.20 adds support for non-square matrixes: mat2x3, mat2x4, mat3x2, mat3x4, mat4x2, mat4x3

Cg and HLSL support matrix data types of the form `float3x3`, `float2x4`, `half4x4`, etc.

When not specifying a shader version number, NVIDIA's GLSL implementation supports both GLSL-style and Cg/HLSL-style scalar, vector, and matrix data types. Use of these extended types will result in a portability warning. When #version has been specified, these extended types will be treated as errors.

In future GLSL-enabled drivers, the preprocessor name `__GLSL_CG_DATA_TYPES` will be defined if these Cg data types are supported to allow GLSL developers to write code like this:

```
#ifndef __GLSL_CG_DATA_TYPES
# define half2 vec2
# define half3 vec3
# define half4 vec4
#endif
```

Cg Standard Library Support

When not specifying a shader version number, the NVIDIA driver will permit the use of the Cg standard library by default, but the usage of the Cg standard library specific functions will result in portability warnings. When specifying a shader version number, usage of these functions will be flagged as an error.

The Cg standard library contains many functions not found in the GLSL standard library. Some examples: `cosh`, `exp`, `log`, `determinant`, `fresnel`, `isfinite`, `isinf`, `isnan`, `lit`, `log10`, `refract`, `round`, `saturate`, `sincos`, `sinh`, `tanh`, `transpose`, etc.

Additionally, the Cg standard library contains certain functions with slightly different names from the equivalent GLSL function names. Examples (GLSL name first, then Cg name): `inversesqrt`/`rsqrt`, `texture1DProj`/`tex1Dproj`, `fract`/`frac`, `dFdx`/`ddx`, etc.

In future GLSL-enabled drivers, the preprocessor name `__GLSL_CG_STDLIB` will be defined if these Cg standard library functions are supported to allow GLSL developers to write code depending on whether the Cg standard library is present or not.

Permissive Constant Conversions

The GLSL grammar for versions 1.00 and 1.10 does not technically allow 2 or 0 be used in place of the floating-point values 2.0 and 0.0 as is possible in other languages. With version 1.20, these are now permitted. By default, NVIDIA's GLSL implementation will warn when this is detected in an unversioned shader.

Parameters to main Allowed

GLSL technically requires the `main` routine of a shader to begin being defined like this:

```
void main(void)
{
```

When no shader version number has been specified, NVIDIA's GLSL implementation will allow the `main` routine to take parameters as allowed in Cg. The info log will contain a portability warning when this behavior is detected. When #version has been used, providing parameters to the shader's main function will result in a compile error.

Depth Textures Obey OpenGL's Depth Compare State

OpenGL 1.4 introduced depth textures, the depth texture mode, and the depth compare mode and function for shadow mapping. The `ARB_fragment_shader` specification says:

> Texture comparison requires the fragment shader to use the shadow versions of the texture lookup functions. Any other texture lookup function issued for a texture with a base internal format of `GL_DEPTH_COMPONENT` will result in *undefined* behavior. Any shadow texture lookup function issued for a texture with a base internal format other than `GL_DEPTH_COMPONENT` will also result in *undefined* behavior. Samplers of type `sampler1DShadow` or `sampler2DShadow` need to be used to indicate the texture image unit that has a depth texture bound to it.

NVIDIA's GLSL implementation simply abides by the texture parameter state indicating whether or not the texture format is a depth format and then whether the texture depth comparison mode is set to `GL_COMPARE_R_TO_TEXTURE`. Additionally, the depth texture mode and compare function work as expected.

#include Preprocessor Directive Works

The `#include` preprocessor directive is reserved in GLSL, but NVIDIA's GLSL implementation supports `#include` as it operates in Cg and C. The include path consists of just the application's working directory. While it presently works when #version 110 or #version 120 are specified, it is expected to be removed in a future driver, so it should not be used.

EXT_Cg_shader

This extension was used in the past as a way of identifying the expanded Cg capabilities of the GLSL compiler, and compiling straight Cg programs in the driver. With the changes to enable enhanced portability and the enhancements to Cg itself, this extension is less necessary. As a result, developers should consider this extension deprecated. It will continue to work, but to get the latest capabilities from Cg, developers should use the stand-alone Cg compilers.

Performance Advice

Currently, the compiling and linking of GLSL programs is not particularly fast. Because a coarse lock is held within the driver (currently), there's no benefit likely from compiling and linking GLSL programs in separate threads.

While performance tuning of the underlying Cg compiler technology is likely to improve the compilation time for GLSL programs, there will also be pressure to implement further optimizations to improve the generated code quality.

Once compiled, the performance of a compiled GLSL shader program compared to an equivalent Cg program should be equivalent because the underlying Cg compiler system is the same. Likewise, an equivalent low-level assembly vertex and fragment programs should also be comparable. There should be no run-time advantage to picking GLSL versus Cg versus low-level assembly if they reduce the same instruction sequences. Of course, hand-coding low-level assembly more efficiently than the compiler generated code will present opportunities to beat the compiler.

With pre-GeForce 8 hardware, the underlying programmable shading architecture is similar to the low-level vertex and fragment assembly instruction sets provided by the NV_vertex_program2 and NV_fragment_program extensions, you may find that you can improve your GLSL (and/or Cg) performance by using a vector instruction to combine multiple scalar operations into a single instruction. For example, a 4-component `min` function generates a single instruction as does a scalar `min` function. So if you use several scalar `min` functions, try to combine them with a single vector `min` function. GeForce 8 hardware uses a scalar instruction set, so merging operations into vectors will typically not be a major performance win.

To target these optimizations, a developer can dump the generated assembly using NVemulate and look for opportunities to rewrite your shaders in ways that exploit vector instructions, swizzles, absolute values, saturation (clamping to between [0,1]), and negation.

For NV3x GPUs, using `half`, `fixed`, and derived vector and matrix data types when the range and precision of these data types is acceptable for your purposes can substantially improve the performance of fragment shaders, particularly large ones.

Reporting GLSL-related Bugs

NVIDIA welcomes email pertaining to GLSL, particularly during this Release 95 GLSL preview period. Send suggestions, feedback, and bug reports to glsl-support@nvidia.com