# Release Notes for NVIDIA OpenGL Shading Language Support

August 30, 2004

These release notes explain the implementation status of the OpenGL Shading Language (GLSL) functionality in NVIDIA's Release 60 drivers.

First, these notes discuss the status of the GLSL standard and NVIDIA's commitment to GLSL. Second, these notes explain how to enable GLSL functionality in Release 60 drivers 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 GLSL standard consists 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 incorporates a version of these extensions into the core OpenGL standard. There are slightly different function names. The ARB suffix is dropped from function names and tokens, and some function names have been changed (for example, dropping the extraneous word Object from the command name). Also the type for shader and program objects is `GLuint`, rather than `GLhandleARB`, but the underlying data type remains an unsigned 32-bit integer.

## *NVIDIA's Support for Multiple Shading Languages*

NVIDIA is well known in the industry for developing the Cg language for programmable real-time graphics. NVIDIA and Microsoft initially collaborated to provide a common shading language syntax and semantics. The fruits of this collaboration became NVIDIA's Cg and Microsoft's High-Level Shading Language (HLSL) implementations. Although both languages are very similar there are technical differences as Cg 1.3 continues to evolve to support the workstation market

NVIDIA continues to develop Cg to improve the expressiveness, functionality, and performance of the language. Cg 1.3 integrates new "sub-shader objects" functionality (introduced by Cg 1.2) for a more object-oriented, modular means of authoring shaders. Cg 1.3 improves the quality of its generated code and supports NVIDIA's latest GPUs including the latest GeForce 6 Series and Quadro FX 4000 GPUs. These latest GPUs support new *vp40* and *fp40* profiles that include support for vertex textures and fragment-level branching respectively. Cg also provides a meta-language known as CgFX (similar to Microsoft's FX format) that encapsulates Cg programs along with the related rendering state necessary to apply the shader. Cg is not tied to a single 3D API so implementations of Cg are available for a wide variety of current and future 3D APIs, GPUs, and OSes. Cg

is the ideal choice for developers that want cutting-edge shading language and meta-language features, support for the latest GPU functionality, and compatibility with HLSL.

GLSL is the result of a multi-vendor standardization process for OpenGL. GLSL is the best shading language choice for developers that seek a shading language supported by multiple hardware vendors and do not mind their shaders being tied to OpenGL. GLSL currently does not provide a FX or CgFX style capability so if those features are required Cg 1.3 may be a better solution for OpenGL development.

NVIDIA supports Cg, HLSL and GLSL through the same underlying Unified Compiler Architecture (UCA). Whether you use GLSL or Cg or both in your OpenGL application is up to you. NVIDIA provides you three choices:

1. Use GLSL with its ARB-approved OpenGL APIs provided by the `ARB_shader_objects`, `ARB_fragment_shader`, and `ARB_vertex_shader` extensions (and eventually through the OpenGL 2.0 version of this functionality).
2. Use Cg with the same ARB-approved OpenGL extension APIs you use for GLSL via the `EXT_Cg_shader` OpenGL extension.
3. Use Cg with the Cg runtime library.

Choice 1 is most compatible with other OpenGL hardware vendors implementing the GLSL standard. Choice 2 can be used in conjunction with Choice 1 to permit multi-lingual 3D applications that can support either GLSL or Cg. Choice 2 is intended for Digital Content Creation (DCC) applications that are typically written with OpenGL but must develop 3D content intended to be deployed in Direct3D-based applications such as PC or Xbox video games. Choice 3 provides access to cutting-edge shading language features such as shader objects and the CgFX meta-language. The Cg runtime generates low-level programmable assembly for multi-vendor OpenGL extensions so Choice 3 can support NVIDIA and non-NVIDIA GPUs. Unlike driver-based shading language extensions where shading language compilation issues can vary by GLSL driver version and vendor, you can deploy a Cg runtime-based application with confidence that the shading compiler used during development and testing is the same one used when you deploy your application. And updating the shading compiler requires redistributing an updated Cg runtime shared library rather than requiring an end-user driver upgrade and reboot.

## *Release 60 Driver Support for GLSL*

The GLSL extensions are advertised by default when a Release 60 driver is installed on a Windows or Linux PC.

You can enable the GLSL extension using a special control panel called NVemulate (`nvemulate.exe`) that you can obtain from the NVIDIA Developers website.

NVIDIA's intention is to expose GLSL by default in future drivers after the GLSL developer preview period is completed.

For GLSL development, a Quadro FX or GeForce FX-based graphics card (NV3x) is highly recommended. If you enable GLSL on a non-FX Quadro or GeForce graphics card (NV1x or NV2x), you can use vertex shaders but not fragment shaders. This reflects the limited per-fragment programmability of non-FX GeForce graphics cards. Specifically, the `ARB_shader_objects` and `ARB_vertex_shader` extensions will be advertised but not the `ARB_fragment_shader` and `EXT_Cg_shader` extensions.

While the performance is extremely slow, non-FX users can rely on software emulation of the per-fragment programmability available in hardware on a Quadro FX or GeForce FX-based graphics card to still play with GLSL support.

NVIDIA TNT-based graphics cards simply do not support GLSL; the NVemulate settings are ignored by TNT-based graphics cards.

NVIDIA is committed to GLSL support for Linux. NVIDIA's Release 60 Linux drivers support GLSL.

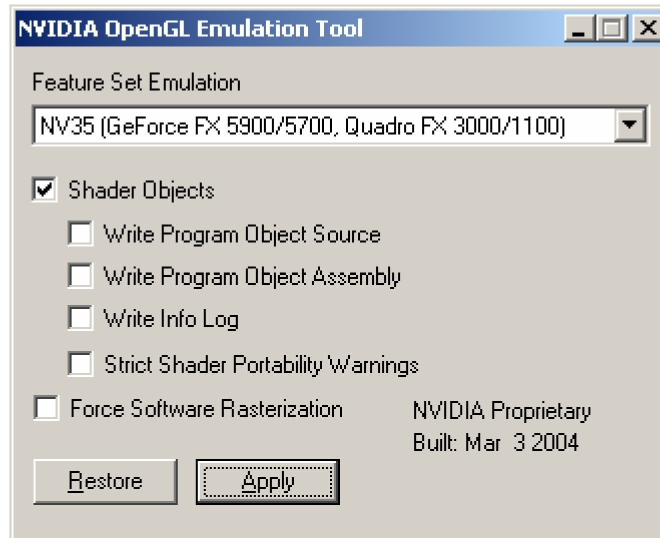
## Planned Release 65 Driver Support for GLSL

Release 65 drivers for GeForce 6 Series and Quadro FX 4*xxx* GPUs support limited textures accesses for vertex textures, dynamic flow control based on both uniform and varying values for fragment shaders, and use of the gl_FrontFacing varying input to fragment shaders. These features are *not* supported even on GeForce 6 Series and Quadro FX 4*xxx* GPUs with Release 60 drivers.

## *Controlling GLSL Support with NVemulate*

GLSL support is enabled **by default** in Release 60 drivers, but you can disable/enable GLSL support as well as manipulate software emulation of GPU feature sets, control the dumping of GLSL source and assembly text for debugging,

The NVemulate control panel looks (something) like this:




To enable the GLSL function, check the "Shader Objects" check 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.

Once you have clicked the "Shader Objects" check box and pressed "Apply", then the next time you start an OpenGL program, GLSL extensions should be available for you to use. You should see the following extensions advertised in the OpenGL extensions string returned by `glGetString(GL_EXTENSIONS)`:

- `ARB_shader_objects` providing the API infrastructure for GLSL
- `ARB_shading_language_100` is a placeholder extension indicating support for the first revision (1.00) of GLSL
- `ARB_vertex_shader` provides per-vertex programmability with GLSL
- `ARB_fragment_shader` provides per-fragment programmability with GLSL
- `EXT_Cg_shader` provides Cg language support for the ARB_shader_objects API

As noted earlier, per-fragment programmability is not available on non-FX Quadro and GeForce graphics cards unless 30 or better is enabled for the "Feature Set Emulation" option list. The `ARB_fragment_shader` and `EXT_Cg_shader` extensions are enabled only if per-fragment programmability is available (whether by hardware support or software emulation).

## NVemulate Troubleshooting

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

- You are using a Release 60 driver, preferably version 61.77 or later.
    - Check this by going to “Display Properties”, then “Advanced”, then “GeForce XXX” or “Quadro XXX” and make sure the version numbers listed from the driver components are 6.14.10.6177 or better.
    - You can also check this by making sure the version string returned by `glGetString(GL_VERSION)` is `1.5.1` or better.
- You are using a Quadro or GeForce graphics card (preferably an FX card).
    - TNT graphics cards do not support GLSL.
- If you installed a new NVIDIA graphics driver, the GLSL-related settings are reset to their defaults (GLSL support disabled) so you must enable GLSL again with NVemulate.
- The Shader Objects setting may be unchecked the first time you run it because no registry entries are set. This may be the case even if GLSL is actually supported by default. If you apply a setting with nvemulate, make sure to check Shader Objects since applying with Shader Objects unchecked will disable GLSL.

## 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.txt` and `vasm_%d.txt` files into the application's working directory when fragment and vertex shaders respectively are linked where the `%d` represents the application’s handle number for the linked program object. These files are in a form similar to what the standalone Cg compiler (`cgc.exe`) outputs. The assembly text conforms to the `NV_vertex_program1_1`, `NV_vertex_program2`, or `NV_fragment_program` extensions depending on the shader type and hardware capabilities.

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.txt` and `vsrc_%d.txt` files into the application's working directory when fragment and vertex shaders respectively are linked where the `%d` represents the application's handle number for the linked program object. 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.txt` file is generated per program object.

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

NVIDIA provides extensions to GLSL to make the language more productive and functional. Use the "Strict Shader Portability Warnings" to obtain additional messages in the info log output of shader and program objects.

These messages are limited right now. Please report portability issues that are not warned about by the compiler so these additional warnings can be provided.

## 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 actually compile shader objects until the program object link. Currently, the GLSL source code for all shader objects for a given domain (vertex or shader) attached to a program object being linked are concatenated and then compiled.

This means (currently) there is no efficiency from compiling shader objects once and linking them in multiple program objects. It also means that (currently) the link status returned by `glCompileShaderARB` is always true even if the shader object source has errors. Errors and warnings will however be reported in the info log after the `glLinkProgramARB` command.

### gl_FrontFacing Is Not Available to Fragment Shaders

The built-in fragment shader varying parameter `gl_FrontFacing` is not supported.

As a workaround, enable with `glEnable` the `GL_VERTEX_PROGRAM_TWO_SIDE_ARB` 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.

### ftransform Does Not Guarantee Position Invariance for non-FX GPUs

The `ftransform` built-in vertex shader function is intended to guarantee the computation of a vertex clip position that is invariant with conventional vertex transformation. While

this is true on Quadro FX and GeForce FX GPUs (NV3x), it is not true (currently) of non-FX GPUs (NV1x and NV2x).

As a workaround, use a vertex shader instead of conventional vertex processing for all vertex processing passes to guarantee invariance on non-FX GPUs

## gl_ClipVertex Is Not Available to Vertex Shaders

GLSL supports a varying vertex shader output parameter called `gl_ClipVertex` used to output a vertex position to be used for clipping by any enabled clip planes.

A future GLSL enhancement will allow 6 clip coordinates (similar to `NV_vertex_program2`'s `o[CLPn]` outputs) to be generated. Expect these to be more efficient and general than `gl_ClipVertex`. Clip coordinates are more efficient because they are directly interpreted as distances from clip planes rather than requiring further dot products with the `glClipPlane` parameters of enabled clip planes. Clip coordinates are more general because they can be in arbitrary coordinate systems whereas the `gl_ClipVertex` and enabled clip planes should be logically in the same coordinate system.

## 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.

## Unsized Arrays Not Handled

GLSL makes allowances for unsized arrays.  NVIDIA's GLSL implementation (currently) does not support unsized arrays. There are likely to be semantic compilation issues with unsized arrays so their use is discouraged.

## Vertex Shaders Cannot Access Textures

NVIDIA's GLSL implementation does not (currently) support standard library routines for sampling textures such as `texture2D`.

## Limited Fragment Shader Control Flow Allowed

NVIDIA's GLSL implementation does not (currently) provide for arbitrary branching and looping in the fragment domain. Branching and looping is more general in the vertex domain (except for NV1x and NV2x GPUs where vertex domain control flow is limited).

## Control Flow Dependent on Uniform Parameters Is Not Allowed

NVIDIA's GLSL implementation does not (currently) allow control flow to depend on uniform parameters in the fragment domain. Some control flow dependent on uniform parameters is allowed in the vertex domain (except for NV1x and NV2x GPUs) but this is not recommended due to poor performance.

In general, control flow dependent on uniform parameters is not recommended because it may well require the expensive recompilation of a shader at run-time. Instead, you should compile and link a stable of program objects for the uniform values you expect to often use where the uniform value is instead handled as a constant.

## 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 `glBindAttribLocationARB`. For example, you should not use `gl_Normal` (a built-in vertex attribute) and also use `glBindAttribLocationARB` 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 |

## Confusion with Low-Level Assembly Reserved Words

Uniform parameters in fragment shaders should not (currently) use names that are reserved words for the `NV_fragment_program` assembly grammar. These include

instruction names such as MIN, register names such as R0 and FOGC, texture image targets such as CUBE, and certain single letters such as f, o, p, w, x, y, and z. As a workaround, use an alternative name.

# 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. 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 supports matrix data types of the form mat2, mat3, mat4. These must be square matrices.

Cg and HLSL support matrix data types of the form float3x3, float2x4, half4x4, etc. These matrix data types are not required to be square.

NVIDIA's GLSL implementation supports both GLSL-style and Cg/HLSL-style scalar, vector, and matrix data types.

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 Supported

The Cg standard library (shared with HLSL) 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/HLSL standard library contains certain functions with slightly different names from the equivalent GLSL function names. Examples (GLSL name first, then Cg/HLSL 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 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 C and C++. NVIDIA's GLSL implementation uses the Cg rules for specifying constants to be more convenient to programmers and more consistent with C/C++.

## Permissive Scalar/Vector Expressions

GLSL does not technically allow a scalar to be multiplied by a vector. The scalar must be first converted to the appropriate vector first. Cg allows such usage because it is convenient, succinct, and mathematically sound. NVIDIA's GLSL implementation allows scalars to be multiplied by vectors as Cg does.

## Parameters to main Allowed

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

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

NVIDIA's GLSL implementation allows the `main` routine to take parameters as allowed in Cg. This usage makes it clearer what uniform and varying parameters a `main` function will actually use.

## Inverse Matrix Built-in Uniforms

In addition to these matrix built-in uniforms:

```
gl_ModelViewMatrix
gl_ModelViewProjectionMatrix
gl_ProjectionMatrix
gl_TextureMatrix[]
```

NVIDIA's GLSL implementation also provides inverse versions of these matrices respectively called:

```
gl_ModelViewMatrixInverse
gl_ModelViewProjectionMatrixInverse
gl_ProjectionMatrixInverse
gl_TextureMatrixInverse[]
```

## 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.

For this reason, there is no difference between the `sampler2DShadow` type and `sampler2D`. Likewise, the `texture2DProj` routine is identical to `shadow2DProj`. This behavior makes NVIDIA's GLSL more consistent with core OpenGL's shadow mapping functionality whereas other implementations fallback on the undefined behavior allowed in the specification language above.

## Texture Rectangle Samplers Work

GLSL reserves `sampler2DRect` as a keyword for the purpose of supporting the `NV_texture_rectangle` extension but does not define standard library functions for `sampler2DRect` samplers. NVIDIA's GLSL implementation does define such functions, namely: `texture2DRect` and `texture2DRectProj`.

In future GLSL-enabled drivers, the preprocessor name `__GLSL_SAMPLER_2D_RECT` will be defined if the `sampler2DRect` data types and related functions are supported.

### #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.

### EXT_Cg_shader

This extension provides a way to create shader objects from Cg source text rather than GLSL source text. Just as some compiler systems have multiple front-ends for Pascal, C, and FORTRAN, NVIDIA's GLSL-capable OpenGL driver can accept both GLSL and Cg source text through the `ARB_shader_objects` API. This provides OpenGL programmers the ability to accept shaders written in either language.

Two new defines are necessary:

```
/* EXT_Cg_shader */
#define GL_CG_VERTEX_SHADER_EXT            0x890E
#define GL_CG_FRAGMENT_SHADER_EXT          0x890F
```

You can pass `GL_CG_VERTEX_SHADER_EXT` to `glCreateShaderARB` instead of `GL_VERTEX_SHADER_ARB` to create a vertex shader object that will parse and compile its shader source with the Cg compiler front-end rather than the GLSL front-end. Likewise, you can pass `GL_CG_FRAGMENT_SHADER_EXT` to `glCreateShaderARB` instead of `GL_FRAGMENT_SHADER_ARB` to create a fragment shader object that will parse and compile its shader source with the Cg front-end rather than the GLSL front-end.

These Cg shaders can use GLSL and Cg standard library functions. They can also use `gl_`-prefixed built-in uniforms values supported by GLSL.

Cg 1.2 features such as sub-shaders are not supported (currently). CgFX is also not supported.

There is no need for an application to call or link with the Cg runtime when using the `EXT_Cg_shader` extension since the ARB_shader_objects API is used instead. The `ARB_shader_objects` API entry points are queried with `wglGetProcAddress` just like other OpenGL extensions.

## *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.

Because 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.

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 60 GLSL preview period. Send suggestions, feedback, and bug reports to glsl-support@nvidia.com