Chapter 2

NVIDIA

Bill Mark

State of the Art in Hardware Shading – NVIDIA

SIGGRAPH 2002 Course Notes

William R. Mark

April 4, 2002

Programmable graphics hardware, APIs, and shading languages are evolving towards greater generality and performance at a rate that is extremely rapid even by the standards of the computer industry. Because course notes must be prepared almost four months in advance of the SIGGRAPH conference, we can't include truly state-of-the-art material in the course notes. Instead, we will prepare a web site at `http://www.nvidia.com/siggraph2002` that will provide course attendees with material to complement the course presentation.

We are including in these course notes a copy of NVIDIA's OpenGL extension specification for the interface to the programmable vertex hardware in the GeForce3 and GeForce4. The `NV_vertex_program` extension is similar to assembly-language-level interfaces that will provide access to the capabilities of future NVIDIA GPUs.

Name

NV_vertex_program

Name Strings

GL_NV_vertex_program

Contact

Mark J. Kilgard, NVIDIA Corporation (mjk 'at' nvidia.com)

Notice

IP Status

NVIDIA Proprietary.

Status

Version 1.6

Version

NVIDIA Date: February 25, 2002
$Date$ $Revision$
$Id: //sw/main/docs/OpenGL/specs/GL_NV_vertex_program.txt#16 $

Number

233

Dependencies

Written based on the wording of the OpenGL 1.2.1 specification and requires OpenGL 1.2.1.

Requires support for the ARB_multitexture extension with at least two texture units.

EXT_point_parameters affects the definition of this extension.

EXT_secondary_color affects the definition of this extension.

EXT_fog_coord affects the definition of this extension.

EXT_vertex_weighting affects the definition of this extension.

ARB_imaging affects the definition of this extension.

Overview

Unextended OpenGL mandates a certain set of configurable per-vertex computations defining vertex transformation, texture coordinate generation and transformation, and lighting. Several extensions have added further per-vertex computations to OpenGL. For example, extensions have defined new texture coordinate generation modes (ARB_texture_cube_map, NV_texgen_reflection, NV_texgen_emboss), new vertex transformation modes (EXT_vertex_weighting), new lighting modes (OpenGL 1.2's separate specular and rescale normal functionality), several modes for fog distance generation (NV_fog_distance), and

eye-distance point size attenuation (EXT_point_parameters).

Each such extension adds a small set of relatively inflexible per-vertex computations.

This inflexibility is in contrast to the typical flexibility provided by the underlying programmable floating point engines (whether micro-coded vertex engines, DSPs, or CPUs) that are traditionally used to implement OpenGL's per-vertex computations. The purpose of this extension is to expose to the OpenGL application writer a significant degree of per-vertex programmability for computing vertex parameters.

For the purposes of discussing this extension, a vertex program is a sequence of floating-point 4-component vector operations that determines how a set of program parameters (defined outside of OpenGL's begin/end pair) and an input set of per-vertex parameters are transformed to a set of per-vertex output parameters.

The per-vertex computations for standard OpenGL given a particular set of lighting and texture coordinate generation modes (along with any state for extensions defining per-vertex computations) is, in essence, a vertex program. However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.

This extension provides an explicit mechanism for defining vertex program instruction sequences for application-defined vertex programs. In order to define such vertex programs, this extension defines a vertex programming model including a floating-point 4-component vector instruction set and a relatively large set of floating-point 4-component registers.

The extension's vertex programming model is designed for efficient hardware implementation and to support a wide variety of vertex programs. By design, the entire set of existing vertex programs defined by existing OpenGL per-vertex computation extensions can be implemented using the extension's vertex programming model.

Issues

What should this extension be called?

RESOLUTION: NV_vertex_program. DirectX 8 refers to its similar functionality as "vertex shaders". This is a confusing term because shaders are usually assumed to operate at the fragment or pixel level, not the vertex level.

Conceptually, what the extension defines is an application-defined program (admittedly limited by its sequential execution model) for processing vertices so the "vertex program" term is more accurate.

Additionally, some of the API machinery in this extension for describing programs could be useful for extending other OpenGL operations with programs (though other types of programs would likely look very different from vertex programs).

What terms are important to this specification?

vertex program mode - when vertex program mode is enabled, vertices are transformed by an application-defined vertex program.

conventional GL vertex transform mode - when vertex program mode is disabled (or the extension is not supported), vertices are

transformed by GL's conventional texgen, lighting, and transform state.

provoke - the verb that denotes the beginning of vertex transformation by either vertex program mode or conventional GL vertex transform mode. Vertices are provoked when either glVertex or glVertexAttribNV(0, ...) is called.

program target - a type or class of program. This extension supports two program targets: the vertex program and the vertex state program. Future extensions could add other program targets.

vertex program - an application-defined vertex program used to transform vertices when vertex program mode is enabled.

vertex state program - a program similar to a vertex program. Unlike a vertex program, a vertex state program runs outside of a glBegin/glEnd pair. Vertex state programs do not transform a vertex. They just update program parameters.

vertex attribute - one of 16 4-component per-vertex parameters defined by this extension. These attributes alias with the conventional per-vertex parameters.

per-vertex parameter - a vertex attribute or a conventional per-vertex parameter such as set by glNormal3f or glColor3f.

program parameter - one of 96 4-component registers available to vertex programs. The state of these registers is shared among all vertex programs.

What part of OpenGL do vertex programs specifically bypass?

Vertex programs bypass the following OpenGL functionality:

- Normal transformation and normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation
- The texture matrix
- The normalization of AUTO_NORMAL evaluated normals
- The modelview and projection matrix transforms
- The per-vertex processing in EXT_point_parameters
- The per-vertex processing in NV_fog_distance
- Raster position transformation
- Client-defined clip planes

Operations not subsumed by vertex programs

- The view frustum clip
- Perspective divide (division by w)

- The viewport transformation
- The depth range transformation
- Clamping the primary and secondary color to [0,1]
- Primitive assembly and subsequent operations
- Evaluator (except the AUTO_NORMAL normalization)

How specific should this specification be about precision?

RESOLUTION: Reasonable precision requirements are incorporated into the specification beyond the often vague requirements of the core OpenGL specification.

This extension essentially defines an instruction set and its corresponding execution environment. The instruction set specified may find applications beyond the traditional purposes of 3D vertex transformation, lighting, and texture coordinate generation that have fairly lax precision requirements. To facilitate such possibly unexpected applications of this functionality, minimum precision requirements are specified.

The minimum precision requirements in the specification are meant to serve as a baseline so that application developers can write vertex programs with minimal worries about precision issues.

What about when the "execution environment" involves support for other extensions?

This extension assumes support for functionality that includes a fog distance, secondary color, point parameters, and multiple texture coordinates.

There is a trade-off between requiring support for these extensions to guarantee a particular extended execution environment and requiring lots of functionality that everyone might not support.

Application developers will desire a high baseline of functionality so that OpenGL applications using vertex programs can work in the full context of OpenGL. But if too much is required, the implementation burden mandated by the extension may limit the number of available implementations.

Clearly we do not want to require support for 8 texture units even if the machinery is there for it. Still multitexture is a common and important feature for using vertex programs effectively. Requiring at least two texture units seems reasonable.

What do we say about the alpha component of the secondary color?

RESOLUTION: When vertex program mode is enabled, the alpha component of csec used for the color sum state is assumed always zero. Another downstream extension may actually make the alpha component written into the COL1 (or BFC1) vertex result register available.

Should client-defined clip planes operate when vertex program mode is enabled?

RESOLUTION. No.

OpenGL's client-defined clip planes are specified in eye-space. Vertex programs generate homogeneous clip space positions. Unlike the conventional OpenGL vertex transformation mode, vertex program mode requires no semantic equivalent to eye-space.

Applications that require client-defined clip planes can simulate OpenGL-style client-defined clip planes by generating texture coordinates and using alpha testing or other per-fragment tests such as NV_texture_shader's CULL_FRAGMENT_NV program to discard fragments. In many ways, these schemes provide a more flexible mechanism for clipping than client-defined clip planes.

Unfortunately, vertex programs used in conjunction with selection or feedback will not have a means to support client-defined clip planes because the per-fragment culling mechanisms described in the previous paragraph are not available in the selection or feedback render modes. Oh well.

Finally, as a practical concern, client-defined clip planes greatly complicate clipping for various hardware rasterization architectures.

How are edge flags handled?

RESOLUTION: Passed through without the ability to be modified by a vertex program. Applications are free to send edge flags when vertex program mode is enabled.

Should vertex attributes alias with conventional per-vertex parameters?

RESOLUTION. YES.

This aliasing should make it easy to use vertex programs with existing OpenGL code that transfers per-vertex parameters using conventional OpenGL per-vertex calls.

It also minimizes the number of per-vertex parameters that the hardware must maintain.

See Table X.2 for the aliasing of vertex attributes and conventional per-vertex parameters.

How should vertex attribute arrays interact with conventional vertex arrays?

RESOLUTION: When vertex program mode is enabled, a particular vertex attribute array will be used if enabled, but if disabled, and the corresponding aliased conventional vertex array is enabled (assuming that there is a corresponding aliased conventional vertex array for the particular vertex array), the conventional vertex array will be used.

This matches the way immediate mode per-vertex parameter aliasing works.

This does slightly complicate vertex array validation in program mode, but programmers using vertex arrays can simply enable vertex program mode without reconfiguring their conventional vertex arrays and get what they expect.

Note that this does create an asymmetry between immediate mode and vertex arrays depending on whether vertex program mode is

enabled or not. The immediate mode vertex attribute commands operate unchanged whether vertex program mode is enabled or not. However the vertex attribute vertex arrays are used only when vertex program mode is enabled.

Supporting vertex attribute vertex arrays when vertex program mode is disabled would create a large implementation burden for existing OpenGL implementations that have heavily optimized conventional vertex arrays. For example, the normal array can be assumed to always contain 3 and only 3 components in conventional OpenGL vertex transform mode, but may contain 1, 2, 3, or 4 components in vertex program mode.

There is not any additional functionality gained by supporting vertex attribute arrays when vertex program mode is disabled, but there is lots of implementation overhead. In any case, it does not seem something worth encouraging so it is simply not supported. So vertex attribute arrays are IGNORED when vertex program mode is not enabled.

Ignoring VertexAttribute commands or treating VertexAttribute commands as an error when vertex program mode is enabled would likely add overhead for such a conditional check. The implementation overhead for supporting VertexAttribute commands when vertex program mode is disabled is not that significant. Additionally, it is likely that setting persistent vertex attribute state while vertex program mode is disabled may be useful to applications. So vertex attribute immediate mode commands are PERMITTED when vertex program mode is not enabled.

Colors and normals specified as ints, uints, shorts, ushorts, bytes, and ubytes are converted to floating-point ranges when supplied to core OpenGL as described in Table 2.6. Other per-vertex attributes such as texture coordinates and positions are not converted. How does this mix with vertex programs where all vertex attributes are supposedly treated identically?

RESOLUTION: Vertex attributes specified as bytes and ubytes are always converted as described in Table 2.6. All other formats are not converted according to Table 2.6 but simply converted directly to floating-point.

The ubyte type is converted because those types seem more useful for passing colors in the [0,1] range.

If an application desires a conversion, the conversion can be incorporated into the vertex program itself.

This also applies to vertex attribute arrays. However, by enabling a color or normal vertex array and not enabling the corresponding aliased vertex attribute array, programmers can get the conventional conversions for color and normal arrays (but only for the vertex attribute arrays that alias to the conventional color and normal arrays and only with the sizes/types supported by these color and normal arrays).

Should programs be C-style null-terminated strings?

RESOLUTION: No. Programs should be specified as an array of GLubyte with an explicit length parameter. OpenGL has no precedent for passing null-terminated strings into the API (though glGetString returns null-terminated strings). Null-terminated strings are problematic for some languages.

Should all existing OpenGL transform functionality and extensions be implementable as vertex programs?

RESOLUTION: Yes. Vertex programs should be a complete superset of what you can do with OpenGL 1.2 and existing vertex transform extensions.

To implement EXT_point_parameters, the GL_VERTEX_PROGRAM_POINT_SIZE_NV enable is introduced.

To implement two-sided lighting, the GL_VERTEX_PROGRAM_TWO_SIDE_NV enable is introduced.

How does glPointSize work with vertex programs?

RESOLUTION: If GL_VERTEX_PROGRAM_POINT_SIZE_NV is disabled, the size of points is determine by the glPointSize state. If enabled, the point size is determined per-vertex by the clamped value of the vertex result PSIZ register.

Can the currently bound vertex program object name be deleted or reloaded?

RESOLUTION. Yes. When a vertex program object name is deleted or reloaded when it is the currently bound vertex program object, it is as if a rebind occurs after the deletion or reload.

In the case of a reload, the new vertex program object will be used from then on. In the case of a deletion, the current vertex program object will be treated as if it is nonexistent.

Should program objects have a mechanism for managing program residency?

RESOLUTION: Yes. Vertex program instruction memory is a limited hardware resource. glBindProgramNV will be faster if binding to a resident program. Applications are likely to want to quickly switch between a small collection of programs.

glAreProgramsResidentNV allows the residency status of a group of programs to be queried. This mimics glAreTexturesResident.

Instead of adopting the glPrioritizeTextures mechanism, a new glRequestResidentProgramsNV command is specified instead. Assigning priorities to textures has always been a problematic endeavor and few OpenGL implementations implemented it effectively. For the priority mechanism to work well, it requires the client to routinely update the priorities of textures.

The glRequestResidentProgramsNV indicates to the GL that a set of programs are intended for use together. Because all the programs are requesting residency as a group, drivers should be able to attempt to load all the requested programs at once (and remove from residency programs not in the group if necessary). Clients can use glAreProgramsResidentNV to query the relative success of the request.

glRequestResidentProgramsNV should be superior to loading programs on-demand because fragmentation can be avoided.

What happens when you execute a nonexistent or invalid program?

RESOLUTION: glBegin will fail with a GL_INVALID_OPERATION if the currently bound vertex program is nonexistent or invalid. The same applies to glRasterPos and any command that implies a glBegin.

Because the glVertex and glVertexAttribNV(0, ...) are ignored outside of a glBegin/glEnd pair (without generating an error) it is impossible to provoke a vertex program if the current vertex program is nonexistent or invalid. Other per-vertex parameters (for examples those set by glColor, glNormal, and glVertexAttribNV when the attribute number is not zero) are recorded since they are legal outside of a glBegin/glEnd.

For vertex state programs, the problem is simpler because glExecuteProgramNV can immediately fail with a GL_INVALID_OPERATION when the named vertex state program is nonexistent or invalid.

What happens when a matrix has been tracked into a set of program parameters, but then glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, addr, GL_NONE, GL_IDENTITY_NV) is performed?

RESOLUTION: The specified program parameters stop tracking a matrix, but they retain the values of the matrix they were last tracking.

Can rows of tracked matrices be queried by querying the program parameters that track them?

RESOLUTION: Yes.

Discussing matrices is confusing because of row-major versus column-major issues. Can you give an example of how a matrix is tracked?

```
GLfloat matrix[16] = { 1, 5, 9,  13,
                       2, 6, 10, 14,
                       3, 7, 11, 15,
                       4, 8, 12, 16 };
GLfloat row1[4], row2[4];

glMatrixMode(GL_MATRIX0_NV);
glLoadMatrixf(matrix);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MATRIX0_NV, GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_MATRIX0_NV, GL_TRANSPOSE_NV);
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 5,
  GL_PROGRAM_PARAMETER_NV, row1);
/* row1 is now [ 2 6 10 14 ] */
glGetProgramParameterfvNV(GL_VERTEX_PROGRAM_NV, 9,
  GL_PROGRAM_PARAMETER_NV, row2);
/* row2 is now [ 5 6 7 8 ] because the tracked matrix is transposed */
```

Should evaluators be extended to evaluate arbitrary vertex attributes?

RESOLUTION: Yes. We'll support 32 new maps (16 for MAP1 and 16 for MAP2) that take priority over the conventional maps that they might alias to (only when vertex program mode is enabled).

These new maps always evaluate all four components. The rationale for this is that if we supported 1, 2, 3, or 4 components, that would add 128 (16*4*2) enumerants which is too many. In addition, if you wanted to evaluate two 2-component vertex attributes, you could instead generate one 4-component vertex attribute and use

the vertex program with swizzling to treat this as two-components.

Moreover, we are assuming 4-component vector instructions so less than 4-component evaluations might not be any more efficient than 4-component evaluations. Implementations that use vector instructions such as Intel's SSE instructions will be easier to implement since they can focus on optimizing just the 4-component case.

How should GL_AUTO_NORMAL work with vertex programs?

RESOLUTION: GL_AUTO_NORMAL should NOT guarantee that the generated analytical normal be normalized. In vertex program mode, the current vertex program can easily normalize the normal if required.

This can lead to greater efficiency if the vertex program transforms the normal to another coordinate system such as eye-space with a transform that preserves vector length. Then a single normalize after transform is more efficient than normalizing after evaluation and also normalizing after transform.

Conceptually, the normalize mandated for AUTO_NORMAL in section 5.1 is just one of the many transformation operations subsumed by vertex programs.

Should the new vertex program related enables push/pop with GL_ENABLE_BIT?

RESOLUTION: Yes. Pushing and popping enable bits is easy. This includes the 32 new evaluator map enable bits. These evaluator enable bits are also pushed and popped using GL_EVAL_BIT.

Should all the vertex attribute state push/pop with GL_CURRENT_BIT?

RESOLUTION: Yes. The state is aliased with the conventional per-vertex parameter state so it really should push/pop.

Should all the vertex attrib vertex array state push/pop with GL_CLIENT_VERTEX_ARRAY_BIT?

RESOLUTION: Yes.

Should all the other vertex program-related state push/pop somehow?

RESOLUTION: No.

The other vertex program doesn't fit well with the existing bits. To be clear, GL_ALL_ATTRIB_BITS does not push/pop vertex program state other than enables.

Should we generate a GL_INVALID_OPERATION operation if updating a vertex attribute greater than 15?

RESOLUTION: Yes.

The other option would be to mask or modulo the vertex attribute index with 16. This is cheap, but it would make it difficult to increase the number of vertex attributes in the future.

If we check for the error, it should be a well predicted branch for immediate mode calls. For vertex arrays, the check is only required at vertex array specification time.

Hopefully this will encourage people to use vertex arrays over immediate mode.

Should writes to program parameter registers during a vertex program be supported?

RESOLUTION. No.

Writes to program parameter registers from within a vertex program would require the execution of vertex programs to be serialized with respect to each other. This would create an unwarranted implementation penalty for parallel vertex program execution implementations.

However vertex state programs may write to program parameter registers (that is the whole point of vertex state programs).

Should we support variously sized immediate mode byte and ubyte commands? How about for vertex arrays?

RESOLUTION. Only support the 4ub mode.

There are simply too many glVertexAttribNV routines. Passing less than 4 bytes at a time is inefficient. We expect the main use for bytes to be for colors where these will be unsigned bytes. So let's just support 4ub mode for bytes. This applies to vertex arrays too.

Should we support integer, unsigned integer, and unsigned short formats for vertex attributes?

RESOLUTION: No. It's just too many immediate mode entry points, most of which are not that useful. Signed shorts are supported however. We expect signed shorts to be useful for passing compact texture coordinates.

Should we support doubles for vertex attributes?

RESOLUTION: Yes. Some implementation of the extension might support double precision. Lots of math routines output double precision.

Should there be a way to determine where in a loaded program string the first parse error occurs?

RESOLUTION: Yes. You can query PROGRAM_ERROR_POSITION_NV.

Should program objects be shared among rendering contexts in the same manner as display lists and texture objects?

RESOLUTION: Yes.

How should this extension interact with color material?

RESOLUTION: It should not. Color material is a conventional OpenGL vertex transform mode. It does not have a place for vertex programs. If you want to emulate color material with vertex programs, you would simply write a program where the material parameters feed from the color vertex attribute.

Should there be a glMatrixMode or glActiveTextureARB style selector for vertex attributes?

RESOLUTION: No. While this would let us reduce a lot of enumerants down, it would make programming a hassle in lots of cases. Consider having to change the vertex attribute mode to enable a set of vertex arrays.

How should gets for vertex attribute array pointers?

RESOLUTION: Add new get commands. Using the existing calls would require adding 4 sets of 16 enumerants stride, type, size, and pointer. That's too many gets.

Instead add glGetVertexAttribNV and glGetVertexAttribPointervNV. glGetVertexAttribNV is also useful for querying the current vertex attribute.

glGet and glGetPointerv will not return vertex attribute array pointers.

Why is the address register numbered and why is it a vector register?

In the future, A0.y and A0.z and A0.w may exist. For this extension, only A0.x is useful. Also in the future, there may be more than one address register.

There's a nice consistency in thinking about all the registers as 4-component vectors even if the address register has only one usable component.

Should vertex programs and vertex state programs be required to have a header token and an end token?

RESOLUTION: Yes.

The "!!VP1.0" and "!!VSP1.0" tokens start vertex programs and vertex state programs respectively. Both types of programs must end with the "END" token.

The initial header token reminds the programmer what type of program they are writing. If vertex programs and vertex state programs are ever read from disk files, the header token can serve as a magic number for identifying vertex programs and vertex state programs.

The target type for vertex programs and vertex state programs can be distinguished based on their respective grammars independent of the initial header tokens, but the initial header tokens will make it easier for programmers to distinguish the two program target types.

We expect programs to often be generated by concatenation of program fragments. The "END" token will hopefully reduce bugs due to specifying an incorrectly concatenated program.

It's tempting to make these additional header and end tokens optional, but if there is a sanity check value in header and end tokens, that value is undermined if the tokens are optional.

What should be said about rendering invariances?

RESOLUTION: See the Appendix A additions below.

The justification for the two rules cited is to support multi-pass rendering when using vertex programs. Different rendering passes will likely use different programs so there must be some means of

guaranteeing that two different programs can generate particular identical vertex results between different passes.

In practice, this does limit the type of vertex program implementations that are possible.

For example, consider a limited hardware implementation of vertex programs that uses a different floating-point implementation than the CPU's floating-point implementation. If the limited hardware implementation can only run small vertex programs (say the hardware provides on 4 temporary registers instead of the required 12), the implementation is incorrect and non-conformant if programs that only require 4 temporary registers use the vertex program hardware, but programs that require more than 4 temporary registers are implemented by the CPU.

This is a very important practical requirement. Consider a multi-pass rendering algorithm where one pass uses a vertex program that uses only 4 temporary registers, but a different pass uses a vertex program that uses 5 temporary registers. If two programs have instruction sequences that given the same input state compute identical resulting vertex positions, the multi-pass algorithm should generate identically positioned primitives for each pass. But given the non-conformant vertex program implementation described above, this could not be guaranteed.

This does not mean that schemes for splitting vertex program implementations between dedicated hardware and CPUs are impossible. If the CPU and dedicated vertex program hardware used IDENTICAL floating-point implementations and therefore generated exactly identical results, the above described could work.

While these invariance rules are vital for vertex programs operating correctly for multi-pass algorithms, there is no requirement that conventional OpenGL vertex transform mode will be invariant with vertex program mode. A multi-pass algorithm should not assume that one pass using vertex program mode and another pass using conventional GL vertex transform mode will generate identically positioned primitives.

Consider that while the conventional OpenGL vertex program mode is repeatable with itself, the exact procedure used to transform vertices is not specified nor is the procedure's precision specified. The GL specification indicates that vertex coordinates are transformed by the modelview matrix and then transformed by the projection matrix. Some implementations may perform this sequence of transformations exactly, but other implementations may transform vertex coordinates by the composite of the modelview and projection matrices (one matrix transform instead of two matrix transforms in sequence). Given this implementation flexibility, there is no way for a vertex program author to exactly duplicate the precise computations used by the conventional OpenGL vertex transform mode.

The guidance to OpenGL application programs is clear. If you are going to implement multi-pass rendering algorithms that require certain invariances between the multiple passes, choose either vertex program mode or the conventional OpenGL vertex transform mode for your rendering passes, but do not mix the two modes.

What range of relative addressing offsets should be allowed?

RESOLUTION: -64 to 63.

Negative offsets are useful for accessing a table centered at zero without extra bias instructions. Having the offsets support much larger magnitudes just seems to increase the required instruction widths. The -64 to 63 range seems like a reasonable compromise.

When EXT_secondary_color is supported, how does the GL_COLOR_SUM_EXT enable affect vertex program mode?

RESOLUTION: The GL_COLOR_SUM_EXT enable has no affect when vertex program mode is enabled.

When vertex program mode is enabled, the color sum operation is always in operation. A program can "avoid" the color sum operation by not writing the COL1 (or BFC1 when GL_VERTEX_PROGRAM_TWO_SIDE_NV) vertex result registers because the default values of all vertex result registers is (0,0,0,1). For the color sum operation, the alpha value is always assumed zero. So by not writing the secondary color vertex result registers, the program assures that zero is added as part of the color sum operation.

If there is a cost to the color sum operation, OpenGL implementations may be smart enough to determine at program bind time whether a secondary color vertex result is generated and implicitly disable the color sum operation.

Why must RCP of 1.0 always be 1.0?

This is important for 3D graphics so that non-projective textures and orthogonal projections work as expected. Basically when q or w is 1.0, things should work as expected.

Stronger requirements such as "RCP of -1.0 must always be -1.0" are encouraged, but there is no compelling reason to state such requirements explicitly as is the case for "RCP of 1.0 must always be 1.0".

What happens when the source scalar value for the ARL instruction is an extremely positive or extremely negative floating-point value? Is there a problem mapping the value to a constrained integer range?

RESOLUTION: It is not a problem. Relative addressing can by offset by a limited range of offsets (-64 to 63). Relative addressing that falls outside of the 0 to 95 range of program parameter registers is automatically mapped to (0,0,0,0).

Clamping the source scalar value for ARL to the range -64 to 160 inclusive is sufficient to ensure that relative addressing is out of range.

How do you perform a 3-component normalize in three instructions?

```
#
# R1 = (nx,ny,nz)
#
# R0.xyz = normalize(R1)
# R0.w   = 1/sqrt(nx*nx + ny*ny + nz*nz)
#
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, R1, R0.w;
```

How do you perform a 3-component cross product in two instructions?

```
  #
  # Cross product |  i     j     k   | into R2.
  #               | R0.x  R0.y  R0.z |
  #               | R1.x  R1.y  R1.z |
  #
  MUL R2, R0.zxyw, R1.yzxw;
  MAD R2, R0.yzxw, R1.zxyw, -R2;
```

How do you perform a 4-component vector absolute value in one instruction?

```
  #
  # Absolute value is the maximum of the negative and positive
  # components of a vector.
  #
  # R1 = abs(R0)
  #
  MAX R1, R0, -R0;
```

How do you compute the determinant of a 3x3 matrix in three instructions?

```
  #
  # Determinant of | R0.x  R0.y  R0.z | into R3
  #                | R1.x  R1.y  R1.z |
  #                | R2.x  R2.y  R2.z |
  #
  MUL R3, R1.zxyw, R2.yzxw;
  MAD R3, R1.yzxw, R2.zxyw, -R3;
  DP3 R3, R0, R3;
```

How do you transform a vertex position by a 4x4 matrix and then perform a homogeneous divide?

```
  #
  # c[20] = modelview row 0
  # c[21] = modelview row 1
  # c[22] = modelview row 2
  # c[23] = modelview row 3
  #
  # result = R5
  #
  DP4 R5.w, v[OPOS], c[23];
  DP4 R5.x, v[OPOS], c[20];
  DP4 R5.y, v[OPOS], c[21];
  DP4 R5.z, v[OPOS], c[22];
  RCP R11, R5.w;
  MUL R5,R5,R11;
```

How do you perform a vector weighting of two vectors using a single weight?

```
  #
  # R2        = vector 0
  # R3        = vector 1
  # v[WGHT].x = scalar weight to blend vectors 0 and 1
  # result    = R2 * v[WGHT].x + R3 * (1-v[WGHT])
  #
  # this is because A*B + (1-A)*C = A*(B-C) + C
  #
  ADD R4, R2, -R3;
  MAD R4, v[WGHT].x, R4, R3;
```

How do you reduce a value to some fundamental period such as 2*PI?

```
#
# c[36] = (1.0/(2*PI), 2*PI, 0.0, 0.0)
#
# R1.x = input value
# R2   = result
#
MUL R0, R1, c[36].x;
EXP R4, R0.x;
MUL R2, R4.y, c[36].y;
```

How do you implement a simple specular and diffuse lighting computation with an eye-space normal?

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[4-7]  = modelview inverse transpose
# c[32]   = normalized eye-space light direction (infinite light)
# c[33]   = normalized constant eye-space half-angle vector (infinite viewer)
# c[35].x = pre-multiplied monochromatic diffuse light color & diffuse material
# c[35].y = pre-multiplied monochromatic ambient light color & diffuse material
# c[36]   = specular color
# c[38].x = specular power
#
# outputs homogenous position and color
#
DP4   o[HPOS].x, c[0], v[OPOS];
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];           # R0 = n' = transformed normal
DP3   R1.x, c[32], R0;               # R1.x = Lpos DOT n'
DP3   R1.y, c[33], R0;               # R1.y = hHat DOT n'
MOV   R1.w, c[38].x;                 # R1.w = specular power
LIT   R2, R1;                        # Compute lighting values
MAD   R3, c[35].x, R2.y, c[35].y;    # diffuse + emissive
MAD   o[COL0].xyz, c[36], R2.z, R3;  # + specular
END
```

Can you perturb transformed vertex positions with a vertex program?

Yes. Here is an example that performs an object-space diffuse lighting computations and perturbs the vertex position based on this lighting result. Do not take this example too seriously.

```
!!VP1.0
#
# c[0-3]  = modelview projection (composite) matrix
# c[32]   = normalized light direction in object-space
# c[35]   = yellow diffuse material, (1.0, 1.0, 0.0, 1.0)
# c[64].x = 0.0
# c[64].z = 0.125, a scaling factor
#
# outputs diffuse illumination for color and perturbed position
#
DP3   R0, c[32], v[NRML];     # light direction DOT normal
MUL   o[COL0].xyz, R0, c[35];
MAX   R0, c[64].x, R0;
MUL   R0, R0, v[NRML];
```

```
    MUL   R0, R0, c[64].z;
    ADD   R1, v[OPOS], -R0;       # perturb object space position
    DP4   o[HPOS].x, c[0], R1;
    DP4   o[HPOS].y, c[1], R1;
    DP4   o[HPOS].z, c[2], R1;
    DP4   o[HPOS].w, c[3], R1;
    END
```

What if more exponential precision is needed than provided by the builtin EXP instruction?

A sequence of vertex program instructions can be used refine the initial EXP approximation. The pseudo-macro below shows an example of how to refine the EXP approximation.

The psuedo-macro requires 10 instructions, 1 temp register, and 2 constant locations.

```
CE0 = { 9.61597636e-03, -1.32823968e-03, 1.47491097e-04, -1.08635004e-05 };
CE1 = { 1.00000000e+00, -6.93147182e-01, 2.40226462e-01, -5.55036440e-02 };

/* Rt != Ro && Rt != Ri */
EXP_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   EXP Rt, Ri.x;                  /* Use appropriate component of Ri */
   MAD Rt.w, c[CE0].w, Rt.y, c[CE0].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE0].x;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].w;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].z;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].y;
   MAD Rt.w, Rt.w,Rt.y, c[CE1].x;
   RCP Rt.w, Rt.w;
   MUL Ro, Rt.w, Rt.x;            /* Apply user write mask to Ro */
}
```

Simulation gives |max abs error| < 3.77e-07 over the range (0.0 <= x < 1.0). Actual vertex program precision may be slightly less accurate than this.

What if more exponential precision is needed than provided by the builtin LOG instruction?

The pseudo-macro requires 10 instructions, 1 temp register, and 3 constant locations.

```
CL0 = { 2.41873696e-01, -1.37531206e-01, 5.20646796e-02, -9.31049418e-03 };
CL1 = { 1.44268966e+00, -7.21165776e-01, 4.78684813e-01, -3.47305417e-01 };
CL2 = { 1.0, NA, NA, NA };

/* Rt != Ro && Rt != Ri */
LOG_MACRO(Ro:vector, Ri:scalar, Rt:vector) {
   LOG Rt, Ri.x;                  /* Use appropriate component of Ri */
   ADD Rt.y, Rt.y, -c[CL2].x;
   MAD Rt.w, c[CL0].w, Rt.y, c[CL0].z;
   MAD Rt.w, Rt.w, Rt.y,c[CL0].y;
   MAD Rt.w, Rt.w, Rt.y,c[CL0].x;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].w;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].z;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].y;
   MAD Rt.w, Rt.w, Rt.y,c[CL1].x;
   MAD Ro, Rt.w, Rt.y, Rt.x;      /* Apply user write mask to Ro */
}
```

```
        Simulation gives |max abs error| < 1.79e-07 over the range (1.0
        <= x < 2.0).  Actual vertex program precision may be slightly
        less accurate than this.

New Procedures and Functions

    void BindProgramNV(enum target, uint id);

    void DeleteProgramsNV(sizei n, const uint *ids);

    void ExecuteProgramNV(enum target, uint id, const float *params);

    void GenProgramsNV(sizei n, uint *ids);

    boolean AreProgramsResidentNV(sizei n, const uint *ids,
                                  boolean *residences);

    void RequestResidentProgramsNV(sizei n, uint *ids);

    void GetProgramParameterfvNV(enum target, uint index,
                                 enum pname, float *params);
    void GetProgramParameterdvNV(enum target, uint index,
                                 enum pname, double *params);

    void GetProgramivNV(uint id, enum pname, int *params);

    void GetProgramStringNV(uint id, enum pname, ubyte *program);

    void GetTrackMatrixivNV(enum target, uint address,
                            enum pname, int *params);

    void GetVertexAttribdvNV(uint index, enum pname, double *params);
    void GetVertexAttribfvNV(uint index, enum pname, float *params);
    void GetVertexAttribivNV(uint index, enum pname, int *params);

    void GetVertexAttribPointervNV(uint index, enum pname, void **pointer);

    boolean IsProgramNV(uint id);

    void LoadProgramNV(enum target, uint id, sizei len,
                       const ubyte *program);

    void ProgramParameter4fNV(enum target, uint index,
                              float x, float y, float z, float w)
    void ProgramParameter4dNV(enum target, uint index,
                              double x, double y, double z, double w)

    void ProgramParameter4dvNV(enum target, uint index,
                               const double *params);
    void ProgramParameter4fvNV(enum target, uint index,
                               const float *params);

    void ProgramParameters4dvNV(enum target, uint index,
                                uint num, const double *params);
    void ProgramParameters4fvNV(enum target, uint index,
                                uint num, const float *params);

    void TrackMatrixNV(enum target, uint address,
                       enum matrix, enum transform);

    void VertexAttribPointerNV(uint index, int size, enum type, sizei stride,
                               const void *pointer);
```

```
    void VertexAttrib1sNV(uint index, short x);
    void VertexAttrib1fNV(uint index, float x);
    void VertexAttrib1dNV(uint index, double x);
    void VertexAttrib2sNV(uint index, short x, short y);
    void VertexAttrib2fNV(uint index, float x, float y);
    void VertexAttrib2dNV(uint index, double x, double y);
    void VertexAttrib3sNV(uint index, short x, short y, short z);
    void VertexAttrib3fNV(uint index, float x, float y, float z);
    void VertexAttrib3dNV(uint index, double x, double y, double z);
    void VertexAttrib4sNV(uint index, short x, short y, short z, short w);
    void VertexAttrib4fNV(uint index, float x, float y, float z, float w);
    void VertexAttrib4dNV(uint index, double x, double y, double z, double w);
    void VertexAttrib4ubNV(uint index, ubyte x, ubyte y, ubyte z, ubyte w);

    void VertexAttrib1svNV(uint index, const short *v);
    void VertexAttrib1fvNV(uint index, const float *v);
    void VertexAttrib1dvNV(uint index, const double *v);
    void VertexAttrib2svNV(uint index, const short *v);
    void VertexAttrib2fvNV(uint index, const float *v);
    void VertexAttrib2dvNV(uint index, const double *v);
    void VertexAttrib3svNV(uint index, const short *v);
    void VertexAttrib3fvNV(uint index, const float *v);
    void VertexAttrib3dvNV(uint index, const double *v);
    void VertexAttrib4svNV(uint index, const short *v);
    void VertexAttrib4fvNV(uint index, const float *v);
    void VertexAttrib4dvNV(uint index, const double *v);
    void VertexAttrib4ubvNV(uint index, const ubyte *v);

    void VertexAttribs1svNV(uint index, sizei n, const short *v);
    void VertexAttribs1fvNV(uint index, sizei n, const float *v);
    void VertexAttribs1dvNV(uint index, sizei n, const double *v);
    void VertexAttribs2svNV(uint index, sizei n, const short *v);
    void VertexAttribs2fvNV(uint index, sizei n, const float *v);
    void VertexAttribs2dvNV(uint index, sizei n, const double *v);
    void VertexAttribs3svNV(uint index, sizei n, const short *v);
    void VertexAttribs3fvNV(uint index, sizei n, const float *v);
    void VertexAttribs3dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4svNV(uint index, sizei n, const short *v);
    void VertexAttribs4fvNV(uint index, sizei n, const float *v);
    void VertexAttribs4dvNV(uint index, sizei n, const double *v);
    void VertexAttribs4ubvNV(uint index, sizei n, const ubyte *v);

New Tokens

    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev, and by the <target> parameter of BindProgramNV,
    ExecuteProgramNV, GetProgramParameter[df]vNV, GetTrackMatrixivNV,
    LoadProgramNV, ProgramParameter[s]4[df][v]NV, and TrackMatrixNV:

        VERTEX_PROGRAM_NV                               0x8620

    Accepted by the <cap> parameter of Disable, Enable, and IsEnabled,
    and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv,
    and GetDoublev:

        VERTEX_PROGRAM_POINT_SIZE_NV                    0x8642
        VERTEX_PROGRAM_TWO_SIDE_NV                      0x8643

    Accepted by the <target> parameter of ExecuteProgramNV and
    LoadProgramNV:

        VERTEX_STATE_PROGRAM_NV                         0x8621
```

Accepted by the <pname> parameter of GetVertexAttrib[dfi]vNV:

ATTRIB_ARRAY_SIZE_NV	0x8623
ATTRIB_ARRAY_STRIDE_NV	0x8624
ATTRIB_ARRAY_TYPE_NV	0x8625
CURRENT_ATTRIB_NV	0x8626

Accepted by the <pname> parameter of GetProgramParameterfvNV
and GetProgramParameterdvNV:

PROGRAM_PARAMETER_NV	0x8644

Accepted by the <pname> parameter of GetVertexAttribPointervNV:

ATTRIB_ARRAY_POINTER_NV	0x8645

Accepted by the <pname> parameter of GetProgramivNV:

PROGRAM_TARGET_NV	0x8646
PROGRAM_LENGTH_NV	0x8627
PROGRAM_RESIDENT_NV	0x8647

Accepted by the <pname> parameter of GetProgramStringNV:

PROGRAM_STRING_NV	0x8628

Accepted by the <pname> parameter of GetTrackMatrixivNV:

TRACK_MATRIX_NV	0x8648
TRACK_MATRIX_TRANSFORM_NV	0x8649

Accepted by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:

MAX_TRACK_MATRIX_STACK_DEPTH_NV	0x862E
MAX_TRACK_MATRICES_NV	0x862F
CURRENT_MATRIX_STACK_DEPTH_NV	0x8640
CURRENT_MATRIX_NV	0x8641
VERTEX_PROGRAM_BINDING_NV	0x864A
PROGRAM_ERROR_POSITION_NV	0x864B

Accepted by the <matrix> parameter of TrackMatrixNV:

NONE	
MODELVIEW	
PROJECTION	
TEXTURE	
COLOR (if ARB_imaging is supported)	
MODELVIEW_PROJECTION_NV	0x8629
TEXTUREi_ARB	

where i is between 0 and n-1 where n is the number of texture units
supported.

Accepted by the <matrix> parameter of TrackMatrixNV and by the
<mode> parameter of MatrixMode:

MATRIX0_NV	0x8630
MATRIX1_NV	0x8631
MATRIX2_NV	0x8632
MATRIX3_NV	0x8633
MATRIX4_NV	0x8634

```
    MATRIX5_NV                                  0x8635
    MATRIX6_NV                                  0x8636
    MATRIX7_NV                                  0x8637

    (Enumerants 0x8638 through 0x863F are reserved for further matrix
    enumerants 8 through 15.)

Accepted by the <transform> parameter of TrackMatrixNV:

    IDENTITY_NV                                 0x862A
    INVERSE_NV                                  0x862B
    TRANSPOSE_NV                                0x862C
    INVERSE_TRANSPOSE_NV                        0x862D

Accepted by the <array> parameter of EnableClientState and
DisableClientState, by the <cap> parameter of IsEnabled, and by
the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and
GetDoublev:

    VERTEX_ATTRIB_ARRAY0_NV                     0x8650
    VERTEX_ATTRIB_ARRAY1_NV                     0x8651
    VERTEX_ATTRIB_ARRAY2_NV                     0x8652
    VERTEX_ATTRIB_ARRAY3_NV                     0x8653
    VERTEX_ATTRIB_ARRAY4_NV                     0x8654
    VERTEX_ATTRIB_ARRAY5_NV                     0x8655
    VERTEX_ATTRIB_ARRAY6_NV                     0x8656
    VERTEX_ATTRIB_ARRAY7_NV                     0x8657
    VERTEX_ATTRIB_ARRAY8_NV                     0x8658
    VERTEX_ATTRIB_ARRAY9_NV                     0x8659
    VERTEX_ATTRIB_ARRAY10_NV                    0x865A
    VERTEX_ATTRIB_ARRAY11_NV                    0x865B
    VERTEX_ATTRIB_ARRAY12_NV                    0x865C
    VERTEX_ATTRIB_ARRAY13_NV                    0x865D
    VERTEX_ATTRIB_ARRAY14_NV                    0x865E
    VERTEX_ATTRIB_ARRAY15_NV                    0x865F

Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv,
Map1d and Map1f and by the <cap> parameter of Enable, Disable, and
IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:

    MAP1_VERTEX_ATTRIB0_4_NV                    0x8660
    MAP1_VERTEX_ATTRIB1_4_NV                    0x8661
    MAP1_VERTEX_ATTRIB2_4_NV                    0x8662
    MAP1_VERTEX_ATTRIB3_4_NV                    0x8663
    MAP1_VERTEX_ATTRIB4_4_NV                    0x8664
    MAP1_VERTEX_ATTRIB5_4_NV                    0x8665
    MAP1_VERTEX_ATTRIB6_4_NV                    0x8666
    MAP1_VERTEX_ATTRIB7_4_NV                    0x8667
    MAP1_VERTEX_ATTRIB8_4_NV                    0x8668
    MAP1_VERTEX_ATTRIB9_4_NV                    0x8669
    MAP1_VERTEX_ATTRIB10_4_NV                   0x866A
    MAP1_VERTEX_ATTRIB11_4_NV                   0x866B
    MAP1_VERTEX_ATTRIB12_4_NV                   0x866C
    MAP1_VERTEX_ATTRIB13_4_NV                   0x866D
    MAP1_VERTEX_ATTRIB14_4_NV                   0x866E
    MAP1_VERTEX_ATTRIB15_4_NV                   0x866F

Accepted by the <target> parameter of GetMapdv, GetMapfv, GetMapiv,
Map2d and Map2f and by the <cap> parameter of Enable, Disable, and
IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv,
GetFloatv, and GetDoublev:
```

MAP2_VERTEX_ATTRIB0_4_NV	0x8670
MAP2_VERTEX_ATTRIB1_4_NV	0x8671
MAP2_VERTEX_ATTRIB2_4_NV	0x8672
MAP2_VERTEX_ATTRIB3_4_NV	0x8673
MAP2_VERTEX_ATTRIB4_4_NV	0x8674
MAP2_VERTEX_ATTRIB5_4_NV	0x8675
MAP2_VERTEX_ATTRIB6_4_NV	0x8676
MAP2_VERTEX_ATTRIB7_4_NV	0x8677
MAP2_VERTEX_ATTRIB8_4_NV	0x8678
MAP2_VERTEX_ATTRIB9_4_NV	0x8679
MAP2_VERTEX_ATTRIB10_4_NV	0x867A
MAP2_VERTEX_ATTRIB11_4_NV	0x867B
MAP2_VERTEX_ATTRIB12_4_NV	0x867C
MAP2_VERTEX_ATTRIB13_4_NV	0x867D
MAP2_VERTEX_ATTRIB14_4_NV	0x867E
MAP2_VERTEX_ATTRIB15_4_NV	0x867F

Additions to Chapter 2 of the OpenGL 1.2.1 Specification (OpenGL Operation)

-- Section 2.10 "Coordinate Transformations"

Add this initial discussion:

"Per-vertex parameters are transformed before the transformation results are used to generate primitives for rasterization, establish a raster position, or generate vertices for selection or feedback.

Each vertex's per-vertex parameters are transformed by one of two vertex transformation modes. The first vertex transformation mode is GL's conventional vertex transformation model. The second mode, known as 'vertex program' mode, transforms the vertex's per-vertex parameters by an application-supplied vertex program.

Vertex program mode is enabled and disabled, respectively, by

```
void Enable(enum target);
```

and

```
void Disable(enum target);
```

with target equal to VERTEX_PROGRAM_NV. When vertex program mode is enabled, vertices are transformed by the currently bound vertex program as discussed in section 2.14."

Update the original initial paragraph in the section to read:

"When vertex program mode is disabled, vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how the transformation is controlled in the case when vertex program mode is disabled. The discussion that continues through section 2.13 applies when vertex program mode is disabled."

-- Section 2.10.2 "Matrices"

Change the first paragraph to read:

"The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
  void MatrixMode(enum mode);
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, PROJECTION, or MATRIXi_NV as the argument. In the case of MATRIXi_NV, i is an integer between 0 and n-1 indicating one of n tracking matrices where n is the value of the implementation defined constant MAX_TRACK_MATRICES_NV. TEXTURE is described later in section 2.10.2, and COLOR is described in section 3.6.3. The tracking matrices of the form MATRIXi_NV are described in section 2.14.5. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix."

Change the last paragraph to read:

"The state required to implement transformations consists of a n-value integer indicating the current matrix mode (where n is 4 + the number of tracking matrices supported), a stack of at least two 4x4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, n stacks (where n is at least 8) of at least one 4x4 matrix for each MATRIXi_NV with associated stack pointers, and a stack of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW."

-- NEW Section 2.14 "Vertex Programs"

"The conventional GL vertex transformation model described in sections 2.10 through 2.13 is a configurable but essentially hard-wired sequence of per-vertex computations based on a canonical set of per-vertex parameters and vertex transformation related state such as transformation matrices, lighting parameters, and texture coordinate generation parameters.

The general success and utility of the conventional GL vertex transformation model reflects its basic correspondence to the typical vertex transformation requirements of 3D applications.

However when the conventional GL vertex transformation model is not sufficient, the vertex program mode provides a substantially more flexible model for vertex transformation. The vertex program mode permits applications to define their own vertex programs.

2.14.1 The Vertex Program Execution Model

A vertex program is a sequence of floating-point 4-component vector operations that operate on per-vertex attributes and program parameters. Vertex programs execute on a per-vertex basis and operate on each vertex completely independently from the processing of other vertices. Vertex programs execute a finite fixed sequence of instructions with no branching or looping. Vertex programs execute without data hazards so results computed in one operation can be used immediately afterwards. The result of a vertex program is a set of vertex result vectors that becomes the transformed vertex parameters used by primitive assembly.

Vertex programs use a specific well-defined instruction set, register set, and operational model defined in the following sections.

The vertex program register set consists of five types of registers described in the following five sections.

2.14.1.1 The Vertex Attribute Registers

The Vertex Attribute Registers are sixteen 4-component vector floating-point registers containing the current vertex's per-vertex attributes. These registers are numbered 0 through 15. These registers are private to each vertex program invocation and are initialized at each vertex program invocation by the current vertex attribute state specified with VertexAttribNV commands. These registers are read-only during vertex program execution. The VertexAttribNV commands used to update the vertex attribute registers can be issued both outside and inside of Begin/End pairs. Vertex program execution is provoked by updating vertex attribute zero. Updating vertex attribute zero outside of a Begin/End pair is ignored without generating any error (identical to the Vertex command operation).

The commands

```
void VertexAttrib{1234}{sfd}NV(uint index, T coords);
void VertexAttrib{1234}{sfd}vNV(uint index, T coords);
void VertexAttrib4ubNV(uint index, T coords);
void VertexAttrib4ubvNV(uint index, T coords);
```

specify the particular current vertex attribute indicated by index. The coordinates for each vertex attribute are named x, y, z, and w. The VertexAttrib1NV family of commands sets the x coordinate to the provided single argument while setting y and z to 0 and w to 1. Similarly, VertexAttrib2NV sets x and y to the specified values, z to 0 and w to 1; VertexAttrib3NV sets x, y, and z, with w set to 1, and VertexAttrib4NV sets all four coordinates. The error INVALID_VALUE is generated if index is greater than 15.

No conversions are applied to the vertex attributes specified as type short, float, or double. However, vertex attributes specified as type ubyte are converted as described by Table 2.6.

The commands

```
void VertexAttribs{1234}{sfd}vNV(uint index, sizei n, T coords[]);
void VertexAttribs4ubvNV(uint index, sizei n, GLubyte coords[]);
```

specify a contiguous set of n vertex attributes. The effect of

```
VertexAttribs{1234}{sfd}vNV(index, n, coords)
```

is the same as the command sequence

```
#define NUM k  /* where k is 1, 2, 3, or 4 components */
int i;
for (i=n-1; i>=0; i--) {
  VertexAttrib{NUM}{sfd}vNV(i+index, &coords[i*NUM]);
}
```

VertexAttribs4ubvNV behaves similarly.

The VertexAttribNV calls equivalent to VertexAttribsNV are issued in reverse order so that vertex program execution is provoked when index is zero only after all the other vertex attributes have first been specified.

2.14.1.2 The Program Parameter Registers

The Program Parameter Registers are ninety-six 4-component floating-point vector registers containing the vertex program parameters. These registers are numbered 0 through 95. This

relatively large set of registers is intended to hold parameters such as matrices, lighting parameters, and constants required by vertex programs. Vertex program parameter registers can be updated in one of two ways: by the ProgramParameterNV commands outside of a Begin/End pair or by a vertex state program executed outside of a Begin/End pair (vertex state programs are discussed in section 2.14.3).

The commands

```
  void ProgramParameter4fNV(enum target, uint index,
                            float x, float y, float z, float w)
  void ProgramParameter4dNV(enum target, uint index,
                           double x, double y, double z, double w)
```

specify the particular program parameter indicated by index. The coordinates values x, y, z, and w are assigned to the respective components of the particular program parameter. target must be VERTEX_PROGRAM_NV.

The commands

```
  void ProgramParameter4dvNV(enum target, uint index, double *params);
  void ProgramParameter4fvNV(enum target, uint index, float *params);
```

operate identically to ProgramParameter4fNV and ProgramParameter4dNV respectively except that the program parameters are passed as an array of four components.

The commands

```
  void ProgramParameters4dvNV(enum target, uint index,
                              uint num, double *params);
  void ProgramParameters4fvNV(enum target, uint index,
                              uint num, float *params);
```

specify a contiguous set of num program parameters. The effect is the same as

```
  for (i=index; i<index+num; i++) {
    ProgramParameter4{fd}vNV(i, params + i*4);
  }
```

The program parameter registers are shared to all vertex program invocations within a rendering context. ProgramParameterNV command updates and vertex state program executions are serialized with respect to vertex program invocations and other vertex state program executions.

Writes to the program parameter registers during vertex state program execution can be maskable on a per-component basis.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 95.

The initial value of all ninety-six program parameter registers is (0,0,0,0).

2.14.1.3 The Address Register

The Address Register is a single 4-component vector signed 32-bit integer register though only the x component of the vector is accessible. The register is private to each vertex program invocation

and is initialized to (0,0,0,0) at every vertex program invocation. This register can be written during vertex program execution (but not read) and its value can be used for as a relative offset for reading vertex program parameter registers. Only the vertex program parameter registers can be read using relative addressing (writes using relative addressing are not supported).

See the discussion of relative addressing of program parameters in section 2.14.1.9 and the discussion of the ARL instruction in section 2.14.1.10.1.

2.14.1.4 The Temporary Registers

The Temporary Registers are twelve 4-component floating-point vector registers used to hold temporary results during vertex program execution. These registers are numbered 0 through 11. These registers are private to each vertex program invocation and initialized to (0,0,0,0) at every vertex program invocation. These registers can be read and written during vertex program execution. Writes to these registers can be maskable on a per-component basis.

2.14.1.5 The Vertex Result Register Set

The Vertex Result Registers are fifteen 4-component floating-point vector registers used to write the results of a vertex program. Each register value is initialized to (0,0,0,1) at the invocation of each vertex program. Writes to the vertex result registers can be maskable on a per-component basis. These registers are named in Table X.1 and further discussed below.

Vertex Result Register Name	Description	Component Interpretation
HPOS	Homogeneous clip space position	(x,y,z,w)
COL0	Primary color (front-facing)	(r,g,b,a)
COL1	Secondary color (front-facing)	(r,g,b,a)
BFC0	Back-facing primary color	(r,g,b,a)
BFC1	Back-facing secondary color	(r,g,b,a)
FOGC	Fog coordinate	(f,*,*,*)
PSIZ	Point size	(p,*,*,*)
TEX0	Texture coordinate set 0	(s,t,r,q)
TEX1	Texture coordinate set 1	(s,t,r,q)
TEX2	Texture coordinate set 2	(s,t,r,q)
TEX3	Texture coordinate set 3	(s,t,r,q)
TEX4	Texture coordinate set 4	(s,t,r,q)
TEX5	Texture coordinate set 5	(s,t,r,q)
TEX6	Texture coordinate set 6	(s,t,r,q)
TEX7	Texture coordinate set 7	(s,t,r,q)

Table X.1: Vertex Result Registers.

HPOS is the transformed vertex's homogeneous clip space position. The vertex's homogeneous clip space position is converted to normalized device coordinates and transformed to window coordinates as described at the end of section 2.10 and in section 2.11. Further processing (subsequent to vertex program termination) is responsible for clipping primitives assembled from vertex program-generated vertices as described in section 2.10 but all client-defined clip planes are treated as if they are disabled when vertex program mode is enabled.

Four distinct color results can be generated for each vertex.

COL0 is the transformed vertex's front-facing primary color.
COL1 is the transformed vertex's front-facing secondary color.
BFC0 is the transformed vertex's back-facing primary color. BFC1 is the transformed vertex's back-facing secondary color.

Primitive coloring may operate in two-sided color mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_TWO_SIDE_NV. The selection between the back-facing colors and the front-facing colors depends on the primitive of which the vertex is a part. If the primitive is a point or a line segment, the front-facing colors are always selected. If the primitive is a polygon and two-sided color mode is disabled, the front-facing colors are selected. If it is a polygon and two-sided color mode is enabled, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. This facingness determination is identical to the two-sided lighting facingness determination described in section 2.13.1.

The selected primary and secondary colors for each primitive are clamped to the range [0,1] and then interpolated across the assembled primitive during rasterization with at least 8-bit accuracy for each color component.

FOGC is the transformed vertex's fog coordinate. The register's first floating-point component is interpolated across the assembled primitive during rasterization and used as the fog distance to compute per-fragment the fog factor when fog is enabled. However, if both fog and vertex program mode are enabled, but the FOGC vertex result register is not written, the fog factor is overridden to 1.0. The register's other three components are ignored.

Point size determination may operate in program-specified point size mode. This behavior is enabled and disabled by calling Enable or Disable with the symbolic value VERTEX_PROGRAM_POINT_SIZE_NV. If the vertex is for a point primitive and the mode is enabled and the PSIZ vertex result is written, the point primitive's size is determined by the clamped x component of the PSIZ register. Otherwise (because vertex program mode is disabled, program-specified point size mode is disabled, or because the vertex program did not write PSIZ), the point primitive's size is determined by the point size state (the state specified using the PointSize command).

The PSIZ register's x component is clamped to the range zero through either the hi value of ALIASED_POINT_SIZE_RANGE if point smoothing is disabled or the hi value of the SMOOTH_POINT_SIZE_RANGE if point smoothing is enabled. The register's other three components are ignored.

If the vertex is not for a point primitive, the value of the PSIZ vertex result register is ignored.

TEX0 through TEX7 are the transformed vertex's texture coordinate sets for texture units 0 through 7. These floating-point coordinates are interpolated across the assembled primitive during rasterization and used for accessing textures. If the number of texture units supported is less than eight, the values of vertex result registers that do not correspond to existent texture units are ignored.

2.14.1.6 Semantic Meaning for Vertex Attributes and Program Parameters

One important distinction between the conventional GL vertex transformation mode and the vertex program mode is that per-vertex

parameters and other state parameters in vertex program mode do not have dedicated semantic interpretations the way that they do with the conventional GL vertex transformation mode.

For example, in the conventional GL vertex transformation mode, the Normal command specifies a per-vertex normal. The semantic that the Normal command supplies a normal for lighting is established because that is how the per-vertex attribute supplied by the Normal command is used by the conventional GL vertex transformation mode. Similarly, other state parameters such as a light source position have semantic interpretations based on how the conventional GL vertex transformation model uses each particular parameter.

In contrast, vertex attributes and program parameters for vertex programs have no pre-defined semantic meanings. The meaning of a vertex attribute or program parameter in vertex program mode is defined by how the vertex attribute or program parameter is used by the current vertex program to compute and write values to the Vertex Result Registers. This is the reason that per-vertex attributes and program parameters for vertex programs are numbered instead of named.

For convenience however, the existing per-vertex parameters for the conventional GL vertex transformation mode (vertices, normals, colors, fog coordinates, vertex weights, and texture coordinates) are aliased to numbered vertex attributes. This aliasing is specified in Table X.2. The table includes how the various conventional components map to the 4-component vertex attribute components.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	vertex position	Vertex	x,y,z,w
1	vertex weights	VertexWeightEXT	w,0,0,1
2	normal	Normal	x,y,z,1
3	primary color	Color	r,g,b,a
4	secondary color	SecondaryColorEXT	r,g,b,1
5	fog coordinate	FogCoordEXT	fc,0,0,1
6	-	-	-
7	-	-	-
8	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s,t,r,q
9	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s,t,r,q
10	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s,t,r,q
11	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s,t,r,q
12	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s,t,r,q
13	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s,t,r,q
14	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s,t,r,q
15	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s,t,r,q

Table X.2: Aliasing of vertex attributes with conventional per-vertex parameters.

Only vertex attribute zero is treated specially because it is the attribute that provokes the execution of the vertex program; this is the attribute that aliases to the Vertex command's vertex coordinates.

The result of a vertex program is the set of post-transformation vertex parameters written to the Vertex Result Registers. All vertex programs must write a homogeneous clip space position, but the other Vertex Result Registers can be optionally written.

Clipping and culling are not the responsibility of vertex programs because these operations assume the assembly of multiple vertices into a primitive. View frustum clipping is performed subsequent to vertex program execution. Clip planes are not supported in vertex program mode.

2.14.1.7 Vertex Program Specification

Vertex programs are specified as an array of ubytes. The array is a string of ASCII characters encoding the program.

The command

```
LoadProgramNV(enum target, uint id, sizei len,
              const ubyte *program);
```

loads a vertex program when the target parameter is VERTEX_PROGRAM_NV. Multiple programs can be loaded with different names. id names the program to load. The name space for programs is the positive integers (zero is reserved). The error INVALID_VALUE occurs if a program is loaded with an id of zero. The error INVALID_OPERATION is generated if a program is loaded for an id that is currently loaded with a program of a different program target. Managing the program name space and binding to vertex programs is discussed later in section 2.14.1.8.

program is a pointer to an array of ubytes that represents the program being loaded. The length of the array is indicated by len.

A second program target type known as vertex state programs is discussed in 2.14.4.

At program load time, the program is parsed into a set of tokens possibly separated by white space. Spaces, tabs, newlines, carriage returns, and comments are considered whitespace. Comments begin with the character "#" and are terminated by a newline, a carriage return, or the end of the program array.

The Backus-Naur Form (BNF) grammar below specifies the syntactically valid sequences for vertex programs. The set of valid tokens can be inferred from the grammar. The token "" represents an empty string and is used to indicate optional rules. A program is invalid if it contains any undefined tokens or characters.

```
<program>              ::= "!!VP1.0" <instructionSequence> "END"

<instructionSequence>  ::= <instructionSequence> <instructionLine>
                         | <instructionLine>

<instructionLine>      ::= <instruction> ";"

<instruction>          ::= <ARL-instruction>
                         | <VECTORop-instruction>
                         | <SCALARop-instruction>
                         | <BINop-instruction>
                         | <TRIop-instruction>

<ARL-instruction>      ::= "ARL" <addrReg> "," <scalarSrcReg>

<VECTORop-instruction> ::= <VECTORop> <maskedDstReg> "," <swizzleSrcReg>

<SCALARop-instruction> ::= <SCALARop> <maskedDstReg> "," <scalarSrcReg>
```

```
<BINop-instruction>    ::= <BINop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg>

<TRIop-instruction>    ::= <TRIop> <maskedDstReg> ","
                           <swizzleSrcReg> "," <swizzleSrcReg> ","
                           <swizzleSrcReg>

<VECTORop>             ::= "MOV"
                         | "LIT"

<SCALARop>             ::= "RCP"
                         | "RSQ"
                         | "EXP"
                         | "LOG"

<BINop>                ::= "MUL"
                         | "ADD"
                         | "DP3"
                         | "DP4"
                         | "DST"
                         | "MIN"
                         | "MAX"
                         | "SLT"
                         | "SGE"

<TRIop>                ::= "MAD"

<scalarSrcReg>         ::= <optionalSign> <srcReg> <scalarSuffix>

<swizzleSrcReg>        ::= <optionalSign> <srcReg> <swizzleSuffix>

<maskedDstReg>         ::= <dstReg> <optionalMask>

<optionalMask>         ::= ""
                         | "." "x"
                         | "."     "y"
                         | "." "x" "y"
                         | "."         "z"
                         | "." "x"     "z"
                         | "."     "y" "z"
                         | "." "x" "y" "z"
                         | "."             "w"
                         | "." "x"         "w"
                         | "."     "y"     "w"
                         | "." "x" "y"     "w"
                         | "."         "z" "w"
                         | "." "x"     "z" "w"
                         | "."     "y" "z" "w"
                         | "." "x" "y" "z" "w"

<optionalSign>         ::= "-"
                         | ""

<srcReg>               ::= <vertexAttribReg>
                         | <progParamReg>
                         | <temporaryReg>

<dstReg>               ::= <temporaryReg>
                         | <vertexResultReg>

<vertexAttribReg>      ::= "v" "[" vertexAttribRegNum "]"

<vertexAttribRegNum>   ::= decimal integer from 0 to 15 inclusive
```

```
                          | "OPOS"
                          | "WGHT"
                          | "NRML"
                          | "COL0"
                          | "COL1"
                          | "FOGC"
                          | "TEX0"
                          | "TEX1"
                          | "TEX2"
                          | "TEX3"
                          | "TEX4"
                          | "TEX5"
                          | "TEX6"
                          | "TEX7"

<progParamReg>          ::= <absProgParamReg>
                          | <relProgParamReg>

<absProgParamReg>       ::= "c" "[" <progParamRegNum> "]"

<progParamRegNum>       ::= decimal integer from 0 to 95 inclusive

<relProgParamReg>       ::= "c" "[" <addrReg> "]"
                          | "c" "[" <addrReg> "+" <progParamPosOffset> "]"
                          | "c" "[" <addrReg> "-" <progParamNegOffset> "]"

<progParamPosOffset>    ::= decimal integer from 0 to 63 inclusive

<progParamNegOffset>    ::= decimal integer from 0 to 64 inclusive

<addrReg>               ::= "A0" "." "x"

<temporaryReg>          ::= "R0"
                          | "R1"
                          | "R2"
                          | "R3"
                          | "R4"
                          | "R5"
                          | "R6"
                          | "R7"
                          | "R8"
                          | "R9"
                          | "R10"
                          | "R11"

<vertexResultReg>       ::= "o" "[" vertexResultRegName "]"

<vertexResultRegName>   ::= "HPOS"
                          | "COL0"
                          | "COL1"
                          | "BFC0"
                          | "BFC1"
                          | "FOGC"
                          | "PSIZ"
                          | "TEX0"
                          | "TEX1"
                          | "TEX2"
                          | "TEX3"
                          | "TEX4"
                          | "TEX5"
                          | "TEX6"
                          | "TEX7"
```

```
<scalarSuffix>          ::= "." <component>

<swizzleSuffix>         ::= ""
                          | "." <component>
                          | "." <component> <component>
                                <component> <component>

<component>             ::= "x"
                          | "y"
                          | "z"
                          | "w"
```

The <vertexAttribRegNum> rule matches both register numbers 0 through 15 and a set of mnemonics that abbreviate the aliasing of conventional the per-vertex parameters to vertex attribute register numbers. Table X.3 shows the mapping from mnemonic to vertex attribute register number and what the mnemonic abbreviates.

Mnemonic	Vertex Attribute Register Number	Meaning
"OPOS"	0	object position
"WGHT"	1	vertex weight
"NRML"	2	normal
"COL0"	3	primary color
"COL1"	4	secondary color
"FOGC"	5	fog coordinate
"TEX0"	8	texture coordinate 0
"TEX1"	9	texture coordinate 1
"TEX2"	10	texture coordinate 2
"TEX3"	11	texture coordinate 3
"TEX4"	12	texture coordinate 4
"TEX5"	13	texture coordinate 5
"TEX6"	14	texture coordinate 6
"TEX7"	15	texture coordinate 7

Table X.3: The mapping between vertex attribute register numbers, mnemonics, and meanings.

A vertex programs fails to load if it does not write at least one component of the HPOS register.

A vertex program fails to load if it contains more than 128 instructions.

A vertex program fails to load if any instruction sources more than one unique program parameter register.

A vertex program fails to load if any instruction sources more than one unique vertex attribute register.

The error INVALID_OPERATION is generated if a vertex program fails to load because it is not syntactically correct or for one of the semantic restrictions listed above.

The error INVALID_OPERATION is generated if a program is loaded for id when id is currently loaded with a program of a different target.

A successfully loaded vertex program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

A successfully loaded program replaces the program previously assigned to the name specified by id. If the OUT_OF_MEMORY error is generated by LoadProgramNV, no change is made to the previous contents of the named program.

Querying the value of PROGRAM_ERROR_POSITION_NV returns a ubyte offset into the last loaded program string indicating where the first error in the program. If the program fails to load because of a semantic restriction that cannot be determined until the program is fully scanned, the error position will be len, the length of the program. If the program loads successfully, the value of PROGRAM_ERROR_POSITION_NV is assigned the value negative one.

2.14.1.8 Vertex Program Binding and Program Management

The current vertex program is invoked whenever vertex attribute zero is updated (whether by a VertexAttributeNV or Vertex command). The current vertex program is updated by

```
BindProgramNV(enum target, uint id);
```

where target must be VERTEX_PROGRAM_NV. This binds the vertex program named by id as the current vertex program. The error INVALID_OPERATION is generated if id names a program that is not a vertex program (for example, if id names a vertex state program as described in section 2.14.4).

Binding to a nonexistent program id does not generate an error. In particular, binding to program id zero does not generate an error. However, because program zero cannot be loaded, program zero is always nonexistent. If a program id is successfully loaded with a new vertex program and id is also the currently bound vertex program, the new program is considered the currently bound vertex program.

The INVALID_OPERATION error is generated when both vertex program mode is enabled and Begin is called (or when a command that performs an implicit Begin is called) if the current vertex program is nonexistent or not valid. A vertex program may not be valid for reasons explained in section 2.14.5.

Programs are deleted by calling

```
void DeleteProgramsNV(sizei n, const uint *ids);
```

ids contains n names of programs to be deleted. After a program is deleted, it becomes nonexistent, and its name is again unused. If a program that is currently bound is deleted, it is as though BindProgramNV has been executed with the same target as the deleted program and program zero. Unused names in ids are silently ignored, as is the value zero.

The command

```
void GenProgramsNV(sizei n, uint *ids);
```

returns n previously unused program names in ids. These names are marked as used, for the purposes of GenProgramsNV only, but they become existent programs only when the are first loaded using LoadProgramNV. The error INVALID_VALUE is generated if n is negative.

An implementation may choose to establish a working set of programs on which binding and ExecuteProgramNV operations (execute programs are

explained in section 2.14.4) are performed with higher performance. A program that is currently part of this working set is said to be resident.

The command

```
boolean AreProgramsResidentNV(sizei n, const uint *ids,
                              boolean *residences);
```

returns TRUE if all of the n programs named in ids are resident, or if the implementation does not distinguish a working set. If at least one of the programs named in ids is not resident, then FALSE is returned, and the residence of each program is returned in residences. Otherwise the contents of residences are not changed. If any of the names in ids are nonexistent or zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single named program can also be queried by calling GetProgramivNV with id set to the name of the program and pname set to PROGRAM_RESIDENT_NV.

AreProgramsResidentNV indicates only whether a program is currently resident, not whether it could not be made resident. An implementation may choose to make a program resident only on first use, for example. The client may guide the GL implementation in determining which programs should be resident by requesting a set of programs to make resident.

The command

```
void RequestResidentProgramsNV(sizei n, const uint *ids);
```

requests that the n programs named in ids should be made resident. While all the programs are not guaranteed to become resident, the implementation should make a best effort to make as many of the programs resident as possible. As a result of making the requested programs resident, program names not among the requested programs may become non-resident. Higher priority for residency should be given to programs listed earlier in the ids array. RequestResidentProgramsNV silently ignores attempts to make resident nonexistent program names or zero. AreProgramsResidentNV can be called after RequestResidentProgramsNV to determine which programs actually became resident.

2.14.1.9 Vertex Program Register Accesses

There are 17 vertex program instructions. The instructions and their respective input and output parameters are summarized in Table X.4.

Opcode	Inputs (scalar or vector)	Output (vector or replicated scalar)	Operation
ARL	s	address register	address register load
MOV	v	v	move
MUL	v,v	v	multiply
ADD	v,v	v	add
MAD	v,v,v	v	multiply and add
RCP	s	ssss	reciprocal
RSQ	s	ssss	reciprocal square root
DP3	v,v	ssss	3-component dot product
DP4	v,v	ssss	4-component dot product
DST	v,v	v	distance vector
MIN	v,v	v	minimum

MAX	v,v	v	maximum
SLT	v,v	v	set on less than
SGE	v,v	v	set on greater equal than
EXP	s	v	exponential base 2
LOG	s	v	logarithm base 2
LIT	v	v	light coefficients

Table X.4: Summary of vertex program instructions. "v" indicates a vector input or output, "s" indicates a scalar input, and "ssss" indicates a scalar output replicated across a 4-component vector.

Instructions use either scalar source values or swizzled source values, indicated in the grammar (see section 2.14.1.7) by the rules <scalarSrcReg> and <swizzleSrcReg> respectively. Either type of source value is negated when the <optionalSign> rule matches "-".

Scalar source register values select one of the source register's four components based on the <component> of the <scalarSuffix> rule. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. The indicated component is used as a scalar for the particular source value.

Swizzled source register values may arbitrarily swizzle the source register's components based on the <swizzleSuffix> rule. In the case where the <swizzleSuffix> matches (ignoring whitespace) the pattern ".????" where each question mark is one of "x", "y", "z", or "w", this indicates the ith component of the source register value should come from the component named by the ith component in the sequence. For example, if the swizzle suffix is ".yzzx" and the source register contains [2.0, 8.0, 9.0, 0.0] the swizzled source register value used by the instruction is [8.0, 9.0, 9.0, 2.0].

If the <swizzleSuffix> rule matches "", this is treated the same as ".xyzw". If the <swizzleSuffix> rule matches (ignoring whitespace) ".x", ".y", ".z", or ".w", these are treated the same as ".xxxx", ".yyyy", ".zzzz", and ".wwww" respectively.

The register sourced for either a scalar source register value or a swizzled source register value is indicated in the grammar by the rule <srcReg>. The <vertexAttribReg>, <progParamReg>, and <temporaryReg> sub-rules correspond to one of the vertex attribute registers, program parameter registers, or temporary register respectively.

The vertex attribute and temporary registers are accessed absolutely based on the numbered register. In the case of vertex attribute registers, if the <vertexAttribRegNum> corresponds to a mnemonic, the corresponding register number from Table X.3 is used.

Either absolute or relative addressing can be used to access the program parameter registers. Absolute addressing is indicated by the grammar by the <absProgParamReg> rule. Absolute addressing accesses the numbered program parameter register indicated by the <progParamRegNum> rule. Relative addressing accesses the numbered program parameter register plus an offset. The offset is the positive value of <progParamPosOffset> if the <progParamPosOffset> rule is matched, or the offset is the negative value of <progParamNegOffset> if the <progParamNegOffset> rule is matched, or otherwise the offset is zero. Relative addressing is available only for program parameter registers and only for reads (not writes). Relative addressing reads outside of the 0 to 95 inclusive range always read the value (0,0,0,0).

The result of all instructions except ARL is written back to a

masked destination register, indicated in the grammar by the rule <maskedDstReg>.

Writes to each component of the destination register can be masked, indicated in the grammar by the <optionalMask> rule. If the optional mask is "", all components are written. Otherwise, the optional mask names particular components to write. The characters "x", "y", "z", and "w" match the x, y, z, and w components respectively. For example, an optional mask of ".xzw" indicates that the x, z, and w components should be written but not the y component. The grammar requires that the destination register mask components must be listed in "xyzw" order.

The actual destination register is indicated in the grammar by the rule <dstReg>. The <temporaryReg> and <vertexResultReg> sub-rules correspond to either the temporary registers or vertex result registers. The temporary registers are determined and accessed as described earlier.

The vertex result registers are accessed absolutely based on the named register. The <vertexResultRegName> rule corresponds to registers named in Table X.1.

2.14.1.10 Vertex Program Instruction Set Operations

The operation of the 17 vertex program instructions are described in this section. After the textual description of each instruction's operation, a register transfer level description is also presented.

The following conventions are used in each instruction's register transfer level description. The 4-component vector variables "t", "u", and "v" are assigned intermediate results. The destination register is called "destination". The three possible source registers are called "source0", "source1", and "source2" respectively.

The x, y, z, and w vector components are referred to with the suffixes ".x", ".y", ".z", and ".w" respectively. The suffix ".c" is used for scalar source register values and c represents the particular source register's selected scalar component. Swizzling of components is indicated with the suffixes ".c***", ".*c**", ".**c*", and ".***c" where c is meant to indicate the x, y, z, or w component selected for the particular source operand swizzle configuration. For example:

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
```

This example indicates that t should be assigned the swizzled version of the source0 operand based on the source0 operand's swizzle configuration.

The variables "negate0", "negate1", and "negate2" are booleans that are true when the respective source value should be negated. The variables "xmask", "ymask", "zmask", and "wmask" are booleans that are true when the destination write mask for the respective component is enabled for writing.

Otherwise, the register transfer level descriptions mimic ANSI C syntax.

The idiom "IEEE(expression)" represents the s23e8 single-precision result of the expression if evaluated using IEEE single-precision

floating point operations. The IEEE idiom is used to specify the maximum allowed deviation from IEEE single-precision floating-point arithmetic results.

The following abbreviations are also used:

+Inf	floating-point representation of positive infinity
-Inf	floating-point representation of negative infinity
+NaN	floating-point representation of positive not a number
-NaN	floating-point representation of negative not a number
NA	not applicable or not used

2.14.1.10.1 ARL: Address Register Load

The ARL instruction moves value of the source scalar into the address register. Conceptually, the address register load instruction is a 4-component vector signed integer register, but the only valid address register component for writing and indexing is the x component. The only use for A0.x is as a base address for program parameter reads. The source value is a float that is truncated towards negative infinity into a signed integer.

```
t.x = source0.c;
if (negate0) t.x = -t.x;
A0.x = floor(t.x);
```

2.14.1.10.2 MOV: Move

The MOV instruction moves the value of the source vector into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
if (xmask) destination.x = t.x;
if (ymask) destination.y = t.y;
if (zmask) destination.z = t.z;
if (wmask) destination.w = t.w;
```

2.14.1.10.3 MUL: Multiply

The MUL instruction multiplies the values of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
```

```
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x * u.x;
    if (ymask) destination.y = t.y * u.y;
    if (zmask) destination.z = t.z * u.z;
    if (wmask) destination.w = t.w * u.w;
```

2.14.1.10.4 ADD: Add

The ADD instruction adds the values of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = t.x + u.x;
    if (ymask) destination.y = t.y + u.y;
    if (zmask) destination.z = t.z + u.z;
    if (wmask) destination.w = t.w + u.w;
```

2.14.1.10.5 MAD: Multiply and Add

The MAD instruction adds the value of the third source vector to the product of the values of the first and second two source vectors, writing the result to the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
```

```
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    v.x = source2.c***;
    v.y = source2.*c**;
    v.z = source2.**c*;
    v.w = source2.***c;
    if (negate2) {
      v.x = -v.x;
      v.y = -v.y;
      v.z = -v.z;
      v.w = -v.w;
    }
    if (xmask) destination.x = t.x * u.x + v.x;
    if (ymask) destination.y = t.y * u.y + v.y;
    if (zmask) destination.z = t.z * u.z + v.z;
    if (wmask) destination.w = t.w * u.w + v.w;
```

2.14.1.10.6 RCP: Reciprocal

The RCP instruction inverts the value of the source scalar into the destination register. The reciprocal of exactly 1.0 must be exactly 1.0.

Additionally the reciprocal of negative infinity gives [-0.0, -0.0, -0.0, -0.0]; the reciprocal of negative zero gives [-Inf, -Inf, -Inf, -Inf]; the reciprocal of positive zero gives [+Inf, +Inf, +Inf, +Inf]; and the reciprocal of positive infinity gives [0.0, 0.0, 0.0, 0.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (t.x == 1.0f) {
      u.x = 1.0f;
    } else {
      u.x = 1.0f / t.x;
    }
    if (xmask) destination.x = u.x;
    if (ymask) destination.y = u.x;
    if (zmask) destination.z = u.x;
    if (wmask) destination.w = u.x;
```

where

```
    | u.x - IEEE(1.0f/t.x) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 2.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.7 RSQ: Reciprocal Square Root

The RSQ instruction assigns the inverse square root of the absolute value of the source scalar into the destination register.

Additionally, RSQ(0.0) gives [+Inf, +Inf, +Inf, +Inf]; and both RSQ(+Inf) and RSQ(-Inf) give [0.0, 0.0, 0.0, 0.0];

```
    t.x = source0.c;
    if (negate0) {
```

```
      t.x = -t.x;
    }
    u.x = 1.0f / sqrt(fabs(t.x));
    if (xmask) destination.x = u.x;
    if (ymask) destination.y = u.x;
    if (zmask) destination.z = u.x;
    if (wmask) destination.w = u.x;
```

where

```
    | u.x - IEEE(1.0f/sqrt(fabs(t.x))) | < 1.0f/(2^22)
```

for 1.0f <= t.x <= 4.0f. The intent of this precision requirement is that this amount of relative precision apply over all values of t.x.

2.14.1.10.8 DP3: Three-Component Dot Product

The DP3 instruction assigns the three-component dot product of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
    }
    v.x = t.x * u.x + t.y * u.y + t.z * u.z;
    if (xmask) destination.x = v.x;
    if (ymask) destination.y = v.x;
    if (zmask) destination.z = v.x;
    if (wmask) destination.w = v.x;
```

2.14.1.10.9 DP4: Four-Component Dot Product

The DP4 instruction assigns the four-component dot product of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
```

```
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
v.x = t.x * u.x + t.y * u.y + t.z * u.z + t.w * u.w;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

2.14.1.10.10 DST: Distance Vector

The DST instructions calculates a distance vector for the values of two source vectors. The first vector is assumed to be [NA, d*d, d*d, NA] and the second source vector is assumed to be [NA, 1.0/d, NA, 1.0/d], where the value of a component labeled NA is undefined. The destination vector is then assigned [1,d,d*d,1.0/d].

```
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0) {
  t.y = -t.y;
  t.z = -t.z;
}
u.y = source1.*c**;
u.w = source1.***c;
if (negate1) {
  u.y = -u.y;
  u.w = -u.w;
}
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.y*u.y;
if (zmask) destination.z = t.z;
if (wmask) destination.w = u.w;
```

2.14.1.10.11 MIN: Minimum

The MIN instruction assigns the component-wise minimum of the two source vectors into the destination register.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x < u.x) ? t.x : u.x;
if (ymask) destination.y = (t.y < u.y) ? t.y : u.y;
if (zmask) destination.z = (t.z < u.z) ? t.z : u.z;
```

```
    if (wmask) destination.w = (t.w < u.w) ? t.w : u.w;
```

2.14.1.10.12 MAX: Maximum

The MAX instruction assigns the component-wise maximum of the two source vectors into the destination register.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x >= u.x) ? t.x : u.x;
    if (ymask) destination.y = (t.y >= u.y) ? t.y : u.y;
    if (zmask) destination.z = (t.z >= u.z) ? t.z : u.z;
    if (wmask) destination.w = (t.w >= u.w) ? t.w : u.w;
```

2.14.1.10.13 SLT: Set On Less Than

The SLT instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is less than the value of the second source vector; otherwise, 0.0 is assigned.

```
    t.x = source0.c***;
    t.y = source0.*c**;
    t.z = source0.**c*;
    t.w = source0.***c;
    if (negate0) {
      t.x = -t.x;
      t.y = -t.y;
      t.z = -t.z;
      t.w = -t.w;
    }
    u.x = source1.c***;
    u.y = source1.*c**;
    u.z = source1.**c*;
    u.w = source1.***c;
    if (negate1) {
      u.x = -u.x;
      u.y = -u.y;
      u.z = -u.z;
      u.w = -u.w;
    }
    if (xmask) destination.x = (t.x < u.x) ? 1.0 : 0.0;
    if (ymask) destination.y = (t.y < u.y) ? 1.0 : 0.0;
    if (zmask) destination.z = (t.z < u.z) ? 1.0 : 0.0;
    if (wmask) destination.w = (t.w < u.w) ? 1.0 : 0.0;
```

2.14.1.10.14 SGE: Set On Greater or Equal Than

The SGE instruction performs a component-wise assignment of either 1.0 or 0.0 into the destination register. 1.0 is assigned if the value of the first source vector is greater than or equal the value of the second source vector; otherwise, 0.0 is assigned.

```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.z = -t.z;
  t.w = -t.w;
}
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) {
  u.x = -u.x;
  u.y = -u.y;
  u.z = -u.z;
  u.w = -u.w;
}
if (xmask) destination.x = (t.x >= u.x) ? 1.0 : 0.0;
if (ymask) destination.y = (t.y >= u.y) ? 1.0 : 0.0;
if (zmask) destination.z = (t.z >= u.z) ? 1.0 : 0.0;
if (wmask) destination.w = (t.w >= u.w) ? 1.0 : 0.0;
```

2.14.1.10.15 EXP: Exponential Base 2

The EXP instruction generates an approximation of the exponential base 2 for the value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The exponential base 2 of the source scalar can be better approximated by destination.x*FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) to 2^destination.y where 0.0 <= destination.y < 1.0.

Additionally, EXP(-Inf) or if the exponential result underflows gives [0.0, 0.0, 0.0, 0.0]; and EXP(+Inf) or if the exponential result overflows gives [+Inf, 0.0, +Inf, 1.0].

```
t.x = source0.c;
if (negate0) {
  t.x = -t.x;
}
q.x = 2^floor(t.x);
q.y = t.x - floor(t.x);
q.z = q.x * APPX(q.y);
if (xmask) destination.x = q.x;
if (ymask) destination.y = q.y;
if (zmask) destination.z = q.z;
if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of exponential

base 2 such that

```
    | exp(q.y*log(2.0))-APPX(q.y) | < 1/(2^11)
```

for all 0 <= q.y < 1.0.

The expression "2^floor(t.x)" should overflow to +Inf and underflow to zero.

2.14.1.10.16 LOG: Logarithm Base 2

The LOG instruction generates an approximation of the logarithm base 2 for the absolute value of a source scalar. This approximation is assigned to the z component of the destination register. Additionally, the x and y components of the destination register are assigned values useful for determining a more accurate approximation. The logarithm base 2 of the absolute value of the source scalar can be better approximated by destination.x+FUNC(destination.y) where FUNC is some user approximation (presumably implemented by subsequent instructions in the vertex program) of log2(destination.y) where 1.0 <= destination.y < 2.0.

Additionally, LOG(0.0) gives [-Inf, 1.0, -Inf, 1.0]; and both LOG(+Inf) and LOG(-Inf) give [+Inf, 1.0, +Inf, 1.0].

```
    t.x = source0.c;
    if (negate0) {
      t.x = -t.x;
    }
    if (fabs(t.x) != 0.0f) {
      if (fabs(t.x) == +Inf) {
        q.x = +Inf;
        q.y = 1.0;
        q.z = +Inf;
      } else {
        q.x = Exponent(t.x);
        q.y = Mantissa(t.x);
        q.z = q.x + APPX(q.y);
      }
    } else {
      q.x = -Inf;
      q.y = 1.0;
      q.z = -Inf;
    }
    if (xmask) destination.x = q.x;
    if (ymask) destination.y = q.y;
    if (zmask) destination.z = q.z;
    if (wmask) destination.w = 1.0;
```

where APPX is an implementation dependent approximation of logarithm base 2 such that

```
    | log(q.y)/log(2.0) - APPX(q.y) | < 1/(2^11)
```

for all 1.0 <= q.y < 2.0.

The "Exponent(t.x)" function returns the unbiased exponent between -126 and 127. For example, "Exponent(1.0)" equals 0.0. (Note that the IEEE floating-point representation maintains the exponent as a biased value.) Larger or smaller exponents should generate +Inf or -Inf respectively. The "Mantissa(t.x)" function returns a value in the range [1.0f, 2.0). The intent of these functions is that fabs(t.x) is approximately "Mantissa(t.x)*2^Exponent(t.x)".

2.14.1.10.17 LIT: Light Coefficients

The LIT instruction is intended to compute ambient, diffuse, and specular lighting coefficients from a diffuse dot product, a specular dot product, and a specular power that is clamped to (-128,128) exclusive. The x component of the source vector is assumed to contain a diffuse dot product (unit normal vector dotted with a unit light vector). The y component of the source vector is assumed to contain a Blinn specular dot product (unit normal vector dotted with a unit half-angle vector). The w component is assumed to contain a specular power.

An implementation must support at least 8 fraction bits in the specular power. Note that because 0.0 times anything must be 0.0, taking any base to the power of 0.0 will yield 1.0.

```
t.x = source0.c***;
t.y = source0.*c**;
t.w = source0.***c;
if (negate0) {
  t.x = -t.x;
  t.y = -t.y;
  t.w = -t.w;
}
if (t.w < -(128.0-epsilon)) t.w = -(128.0-epsilon);
else if (t.w > 128-epsilon) t.w = 128-epsilon;
if (t.x < 0.0) t.x = 0.0;
if (t.y < 0.0) t.y = 0.0;
if (xmask) destination.x = 1.0;
if (ymask) destination.y = t.x;
if (zmask) destination.z = (t.x > 0.0) ? EXP(t.w*LOG(t.y)) : 0.0;
if (wmask) destination.w = 1.0;
```

where EXP and LOG are functions that approximate the exponential base 2 and logarithm base 2 with the identical accuracy and special case requirements of the EXP and LOG instructions. epsilon is 1.0/256.0 or approximately 0.0039 which would correspond to representing the specular power with a s8.8 representation.

2.14.1.11 Vertex Program Floating Point Requirements

All vertex program calculations are assumed to use IEEE single precision floating-point math with a format of s1e8m23 (one signed bit, 8 bits of exponent, 23 bits of magnitude) or better and the round-to-zero rounding mode. The only exceptions to this are the RCP, RSQ, LOG, EXP, and LIT instructions.

Note that (positive or negative) 0.0 times anything is (positive) 0.0.

The RCP and RSQ instructions deliver results accurate to 1.0/(2^22) and the approximate output (the z component) of the EXP and LOG instructions only has to be accurate to 1.0/(2^11). The LIT instruction specular output (the z component) is allowed an error equivalent to the combination of the EXP and LOG combination to implement a power function.

The floor operations used by the ARL and EXP instructions must operate identically. Specifically, the EXP instruction's floor(t.x) intermediate result must exactly match the integer stored in the address register by the ARL instruction.

Since distance is calculated as (d^2)*(1/sqrt(d^2)), 0.0 multiplied by anything must be 0.0. This affects the MUL, MAD, DP3, DP4, DST, and LIT instructions.

Because if/then/else conditional evaluation is done by multiplying by 1.0 or 0.0 and adding, the floating point computations require:

```
  0.0 * x = 0.0    for all x (including +Inf, -Inf, +NaN, and -NaN)
  1.0 * x = x      for all x (including +Inf and -Inf)
  0.0 + x = x      for all x (including +Inf and -Inf)
```

Including +Inf, -Inf, +NaN, and -NaN when applying the above three rules is recommended but not required. (The recommended inclusion of +Inf, -Inf, +NaN, and -NaN when applying the first rule is inconsistent with IEEE floating-point requirements.)

For the purpose of comparisons performed by the SGE and SLT instructions, -0.0 is less than +0.0. (This is inconsistent with IEEE floating-point requirements).

No floating-point exceptions or interrupts are generated. Denorms are not supported; if a denorm is input, it is treated as 0.0 (ie, denorms are flushed to zero).

Computations involving +NaN or -NaN generate +NaN, except for the requirement that zero times +NaN or -NaN must always be zero. (This exception is inconsistent with IEEE floating-point requirements).

2.14.2 Vertex Program Update for the Current Raster Position

When vertex programs are enabled, the raster position is determined by the current vertex program. The raster position specified by RasterPos is treated as if they were specified in a Vertex command. The contents of vertex result register set is used to update respective raster position state.

Assuming an existent program, the homogeneous clip-space coordinates are passed to clipping as if they represented a point and assuming no client-defined clip planes are enabled. If the point is not culled, then the projection to window coordinates is computed (section 2.10) and saved as the current raster position and the valid bit is set. If the current vertex program is nonexistent or the "point" is culled, the current raster position and its associated data become indeterminate and the raster position valid bit is cleared.

2.14.3 Vertex Arrays for Vertex Attributes

Data for vertex attributes in vertex program mode may be specified using vertex array commands. The client may specify and enable any of sixteen vertex attribute arrays.

The vertex attribute arrays are ignored when vertex program mode is disabled. When vertex program mode is enabled, vertex attribute arrays are used.

The command

```
  void VertexAttribPointerNV(uint index, int size, enum type,
                             sizei stride, const void *pointer);
```

describes the locations and organizations of the sixteen vertex attribute arrays. index specifies the particular vertex attribute to be described. size indicates the number of values per vertex

that are stored in the array; size must be one of 1, 2, 3, or 4. type specifies the data type of the values stored in the array. type must be one of SHORT, FLOAT, DOUBLE, or UNSIGNED_BYTE and these values correspond to the array types short, int, float, double, and ubyte respectively. The INVALID_OPERATION error is generated if type is UNSIGNED_BYTE and size is not 4. The INVALID_VALUE error is generated if index is greater than 15. The INVALID_VALUE error is generated if stride is negative.

The one, two, three, or four values in an array that correspond to a single vertex attribute comprise an array element. The values within each array element at stored sequentially in memory. If the stride is specified as zero, then array elements are stored sequentially as well. Otherwise points to the ith and (i+1)st elements of an array differ by stride basic machine units (typically unsigned bytes), the pointer to the (i+1)st element being greater. pointer specifies the location in memory of the first value of the first element of the array being specified.

Vertex attribute arrays are enabled with the EnableClientState command and disabled with the DisableClientState command. The value of the argument to either command is VERTEX_ATTRIB_ARRAYi_NV where i is an integer between 0 and 15; specifying a value of i enables or disables the vertex attribute array with index i. The constants obey VERTEX_ATTRIB_ARRAYi_NV = VERTEX_ATTRIB_ARRAY0_NV + i.

When vertex program mode is enabled, the ArrayElement command operates as described in this section in contrast to the behavior described in section 2.8. Likewise, any vertex array transfer commands that are defined in terms of ArrayElement (DrawArrays, DrawElements, and DrawRangeElements) assume the operation of ArrayElement described in this section when vertex program mode is enabled.

When vertex program mode is enabled, the ArrayElement command transfers the ith element of particular enabled vertex arrays as described below. For each enabled vertex attribute array, it is as though the corresponding command from section 2.14.1.1 were called with a pointer to element i. For each vertex attribute, the corresponding command is VertexAttrib[size][type]v, where size is one of [1,2,3,4], and type is one of [s,f,d,ub], corresponding to the array types short, int, float, double, and ubyte respectively.

However, if a given vertex attribute array is disabled, but its corresponding aliased conventional per-vertex parameter's vertex array (as described in section 2.14.1.6) is enabled, then it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. In this case, the corresponding command is determined as described in section 2.8's description of ArrayElement.

If the vertex attribute array 0 is enabled, it is as though VertexAttrib[size][type]v(0, ...) is executed last, after the executions of other corresponding commands. If the vertex attribute array 0 is disabled but the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of other corresponding commands.

2.14.4 Vertex State Programs

Vertex state programs share the same instruction set as and a similar execution model to vertex programs. While vertex program are executed implicitly when a vertex transformation is provoked, vertex state programs are executed explicitly, independently of any vertices.

Vertex state programs can write program parameter registers, but may not write vertex result registers.

The purpose of a vertex state program is to update program parameter registers by means of an application-defined program. Typically, an application will load a set of program parameters and then execute a vertex state program that reads and updates the program parameter registers. For example, a vertex state program might normalize a set of unnormalized vectors previously loaded as program parameters. The expectation is that subsequently executed vertex programs would use the normalized program parameters.

Vertex state programs are loaded with the same LoadProgramNV command (see section 2.14.1.7) used to load vertex programs except that the target must be VERTEX_STATE_PROGRAM_NV when loading a vertex state program.

Vertex state programs must conform to a more limited grammar than the grammar for vertex programs. The vertex state program grammar for syntactically valid sequences is the same as the grammar defined in section 2.14.1.7 with the following modified rules:

```
<program>              ::= "!!VSP1.0" <instructionSequence> "END"

<dstReg>               ::= <absProgParamReg>
                         | <temporaryReg>

<vertexAttribReg>      ::= "v" "[" "0" "]"
```

A vertex state program fails to load if it does not write at least one program parameter register.

A vertex state program fails to load if it contains more than 128 instructions.

A vertex state program fails to load if any instruction sources more than one unique program parameter register.

A vertex state program fails to load if any instruction sources more than one unique vertex attribute register (this is necessarily true because only vertex attribute 0 is available in vertex state programs).

The error INVALID_OPERATION is generated if a vertex state program fails to load because it is not syntactically correct or for one of the other reasons listed above.

A successfully loaded vertex state program is parsed into a sequence of instructions. Each instruction is identified by its tokenized name. The operation of these instructions when executed is defined in section 2.14.1.10.

Executing vertex state programs is legal only outside a Begin/End pair. A vertex state program may not read any vertex attribute register other than register zero. A vertex state program may not write any vertex result register.

The command

```
ExecuteProgramNV(enum target, uint id, const float *params);
```

executes the vertex state program named by id. The target must be VERTEX_STATE_PROGRAM_NV and the id must be the name of program loaded

with a target type of VERTEX_STATE_PROGRAM_NV. params points to an array of four floating-point values that are loaded into vertex attribute register zero (the only vertex attribute readable from a vertex state program).

The INVALID_OPERATION error is generated if the named program is nonexistent, is invalid, or the program is not a vertex state program. A vertex state program may not be valid for reasons explained in section 2.14.5.

2.14.5 Tracking Matrices

As a convenience to applications, standard GL matrix state can be tracked into program parameter vectors. This permits vertex programs to access matrices specified through GL matrix commands.

In addition to GL's conventional matrices, several additional matrices are available for tracking. These matrices have names of the form MATRIXi_NV where i is between zero and n-1 where n is the value of the MAX_TRACK_MATRICES_NV implementation dependent constant. The MATRIXi_NV constants obey MATRIXi_NV = MATRIX0_NV + i. The value of MAX_TRACK_MATRICES_NV must be at least eight. The maximum stack depth for tracking matrices is defined by the MAX_TRACK_MATRIX_STACK_DEPTH_NV and must be at least 1.

The command

```
TrackMatrixNV(enum target, uint address, enum matrix, enum transform);
```

tracks a given transformed version of a particular matrix into a contiguous sequence of four vertex program parameter registers beginning at address. target must be VERTEX_PROGRAM_NV (though tracked matrices apply to vertex state programs as well because both vertex state programs and vertex programs shared the same program parameter registers). matrix must be one of NONE, MODELVIEW, PROJECTION, TEXTURE, TEXTUREi_ARB (where i is between 0 and n-1 where n is the number of texture units supported), COLOR (if the ARB_imaging subset is supported), MODELVIEW_PROJECTION_NV, or MATRIXi_NV. transform must be one of IDENTITY_NV, INVERSE_NV, TRANSPOSE_NV, or INVERSE_TRANSPOSE_NV. The INVALID_VALUE error is generated if address is not a multiple of four.

The MODELVIEW_PROJECTION_NV matrix represents the concatenation of the current modelview and projection matrices. If M is the current modelview matrix and P is the current projection matrix, then the MODELVIEW_PROJECTION_NV matrix is C and computed as

$$C = P M$$

Matrix tracking for the specified program parameter register and the next consecutive three registers is disabled when NONE is supplied for matrix. When tracking is disabled the previously tracked program parameter registers retain the state of their last tracked values. Otherwise, the specified transformed version of matrix is tracked into the specified program parameter register and the next three registers. Whenever the matrix changes, the transformed version of the matrix is updated in the specified range of program parameter registers. If TEXTURE is specified for matrix, the texture matrix for the current active texture unit is tracked. If TEXTUREi_ARB is specified for matrix, the <i>th texture matrix is tracked.

Matrices are tracked row-wise meaning that the top row of the transformed matrix is loaded into the program parameter address,

the second from the top row of the transformed matrix is loaded into the program parameter address+1, the third from the top row of the transformed matrix is loaded into the program parameter address+2, and the bottom row of the transformed matrix is loaded into the program parameter address+3. The transformed matrix may be identical to the specified matrix, the inverse of the specified matrix, the transpose of the specified matrix, or the inverse transpose of the specified matrix, depending on the value of transform.

When matrix tracking is enabled for a particular program parameter register sequence, updates to the program parameter using ProgramParameterNV commands, a vertex program, or a vertex state program are not possible. The INVALID_OPERATION error is generated if a ProgramParameterNV command is used to update a program parameter register currently tracking a matrix.

The INVALID_OPERATION error is generated by ExecuteProgramNV when the vertex state program requested for execution writes to a program parameter register that is currently tracking a matrix because the program is considered invalid.

2.14.6 Required Vertex Program State

The state required for vertex programs consists of:

- a bit indicating whether or not program mode is enabled;
- a bit indicating whether or not two-sided color mode is enabled;
- a bit indicating whether or not program-specified point size mode is enabled;
- 96 4-component floating-point program parameter registers;
- 16 4-component vertex attribute registers (though this state is aliased with the current normal, primary color, secondary color, fog coordinate, weights, and texture coordinate sets);
- 24 sets of matrix tracking state for each set of four sequential program parameter registers, consisting of a n-valued integer indicated the tracked matrix or GL_NONE (where n is 5 + the number of texture units supported + the number of tracking matrices supported) and a four-valued integer indicating the transformation of the tracked matrix;
- an unsigned integer naming the currently bound vertex program
- and the state must be maintained to indicate which integers are currently in use as program names.

Each existent program object consists of a target, a boolean indicating whether the program is resident, an array of type ubyte containing the program string, and the length of the program string array. Initially, no program objects exist.

Program mode, two-sided color mode, and program-specified point size mode are all initially disabled.

The initial state of all 96 program parameter registers is (0,0,0,0).

The initial state of the 16 vertex attribute registers is (0,0,0,1) except in cases where a vertex attribute register aliases to a conventional GL transform mode vertex parameter in which case

the initial state is the initial state of the respective aliased conventional vertex parameter.

The initial state of the 24 sets of matrix tracking state is NONE for the tracked matrix and IDENTITY_NV for the transformation of the tracked matrix.

The initial currently bound program is zero.

The client state required to implement the 16 vertex attribute arrays consists of 16 boolean values, 16 memory pointers, 16 integer stride values, 16 symbolic constants representing array types, and 16 integers representing values per element. Initially, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four."

Additions to Chapter 3 of the OpenGL 1.2.1 Specification (Rasterization)

-- Section 3.3 "Points"

Change the first paragraph to read:

"When program vertex mode is disabled, the point size for rasterizing points is controlled with

```
void PointSize(float size);
```

size specifies the width or diameter of a point. The initial point size value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE. When vertex program mode is enabled, the point size for rasterizing points is determined as described in section 2.14.1.5."

-- Section 3.9 "Color Sum"

Change the first paragraph to read:

"At the beginning of color sum, a fragment has two RGBA colors: a primary color cpri (which texturing, if enabled, may have modified) and a secondary color csec. If vertex program mode is disabled, csec is defined by the lighting equations in section 2.13.1. If vertex program mode is enabled, csec is the fragment's secondary color, obtained by interpolating the COL1 (or BFC1 if the primitive is a polygon, the vertex program two-sided color mode is enabled, and the polygon is back-facing) vertex result register RGB components for the vertices making up the primitive; the alpha component of csec when program mode is enabled is always zero. The components of these two colors are summed to produce a single post-texturing RGBA color c. The components of c are then clamped to the range [0,1]."

-- Section 3.10 "Fog"

Change the initial sentences in the second paragraph to read:

"This factor f may be computed according to one of three equations:

```
f = exp(-d*c)                                   (3.24)
f = exp(-(d*c)^2)                               (3.25)
f = (e-c)/(e-s)                                 (3.26)
```

If vertex program mode is enabled, then c is the fragment's fog coordinate, obtained by interpolating the FOGC vertex result register values for the vertices making up the primitive. When vertex program

mode is disabled, the c is the eye-coordinate distance from the eye, (0,0,0,1) in eye-coordinates, to the fragment center." ...

Additions to Chapter 4 of the OpenGL 1.2.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the OpenGL 1.2.1 Specification (Special Functions)

-- Section 5.1 "Evaluators"

Add the following lines to the end of table 5.1 (page 165):

target	k	values
MAP1_VERTEX_ATTRIB0_4_NV	4	x, y, z, w vertex attribute 0
MAP1_VERTEX_ATTRIB1_4_NV	4	x, y, z, w vertex attribute 1
MAP1_VERTEX_ATTRIB2_4_NV	4	x, y, z, w vertex attribute 2
MAP1_VERTEX_ATTRIB3_4_NV	4	x, y, z, w vertex attribute 3
MAP1_VERTEX_ATTRIB4_4_NV	4	x, y, z, w vertex attribute 4
MAP1_VERTEX_ATTRIB5_4_NV	4	x, y, z, w vertex attribute 5
MAP1_VERTEX_ATTRIB6_4_NV	4	x, y, z, w vertex attribute 6
MAP1_VERTEX_ATTRIB7_4_NV	4	x, y, z, w vertex attribute 7
MAP1_VERTEX_ATTRIB8_4_NV	4	x, y, z, w vertex attribute 8
MAP1_VERTEX_ATTRIB9_4_NV	4	x, y, z, w vertex attribute 9
MAP1_VERTEX_ATTRIB10_4_NV	4	x, y, z, w vertex attribute 10
MAP1_VERTEX_ATTRIB11_4_NV	4	x, y, z, w vertex attribute 11
MAP1_VERTEX_ATTRIB12_4_NV	4	x, y, z, w vertex attribute 12
MAP1_VERTEX_ATTRIB13_4_NV	4	x, y, z, w vertex attribute 13
MAP1_VERTEX_ATTRIB14_4_NV	4	x, y, z, w vertex attribute 14
MAP1_VERTEX_ATTRIB15_4_NV	4	x, y, z, w vertex attribute 15

Replace the four paragraphs on pages 167-168 that explain the operation of EvalCoord:

"EvalCoord operates differently depending on whether vertex program mode is enabled or not. We first discuss how EvalCoord operates when vertex program mode is disabled.

When one of the EvalCoord commands is issued and vertex program mode is disabled, all currently enabled maps (excluding the maps that correspond to vertex attributes, i.e. maps of the form MAPx_VERTEX_ATTRIBn_4_NV). ..."

Add a paragraph before the initial paragraph discussing AUTO_NORMAL:

"When one of the EvalCoord commands is issued and vertex program mode is enabled, the evaluation and the issuing of per-vertex parameter commands matches the discussion above, except that if any vertex attribute maps are enabled, the corresponding VertexAttribNV call for each enabled vertex attribute map is issued with the map's evaluated coordinates and the corresponding aliased per-vertex parameter map is ignored if it is also enabled, with one important difference. As is the case when vertex program mode is disabled, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise the current values are used). The order of the effective commands is immaterial, except that Vertex or VertexAttribNV(0, ...) (the commands that issue provoke vertex program execution) must be issued last. Use of evaluators has no effect on the current vertex attributes or conventional per-vertex parameters. If a vertex attribute map is disabled, but its corresponding conventional per-vertex parameter map is enabled, the conventional per-vertex

parameter map is evaluated and issued as when vertex program mode is not enabled."

Replace the two paragraphs discussing AUTO_NORMAL with:

"Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled or if both MAP2_VERTEX_ATTRIB0_4_NV and vertex program mode are enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex (when program mode is disabled) or as vertex attribute 2 (when vertex program mode is enabled). Automatic normal generation is controlled with Enable and Disable with the symbolic constant AUTO_NORMAL. If automatic normal generation is disabled and vertex program mode is enabled, then vertex attribute 2 is evaluated as usual. If automatic normal generation and vertex program mode are disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a map corresponding to the normal per-vertex parameter (or vertex attribute 2 in program mode) are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used). For MAP_VERTEX3, let q=p. For MAP_VERTEX_4 or MAP2_VERTEX_ATTRBI0_4_NV, let q = (x/w, y/w, z/w) where (x,y,z,w)=p. Then let

m = (partial q / partial u) cross (partial q / partial v)

Then when vertex program mode is disabled, the generated analytic normal, n, is given by n=m/||m||. However, when vertex program mode is enabled, the generated analytic normal used for vertex attribute 2 is simply (mx,my,mz,1). In vertex program mode, the normalization of the generated analytic normal can be performed by the current vertex program."

Change the respective sentences of the last paragraph discussing required evaluator state to read:

"The state required for evaluators potentially consists of 9 conventional one-dimensional map specifications, 16 vertex attribute one-dimensional map specifications, 9 conventional two-dimensional map specifications, and 16 vertex attribute two-dimensional map specifications indicating which are enabled. ... All vertex coordinate maps produce the coordinates (0,0,0,1) (or the appropriate subset); all normal coordinate maps produce (0,0,1); RGBA maps produce (1,1,1,1); color index maps produce 1.0; texture coordinate maps produce (0,0,0,1); and vertex attribute maps produce (0,0,0,1). ... If any evaluation command is issued when none of MAPn_VERTEX_3, MAPn_VERTEX_4, or MAPn_VERTEX_ATTRIB0_NV (where n is the map dimension being evaluated) are enabled, nothing happens."

-- Section 5.4 "Display Lists"

Add to the list of commands not compiled into display lists in the third to the last paragraph:

"AreProgramsResidentNV, IsProgramNV, GenProgramsNV, DeleteProgramsNV, VertexAttribPointerNV"

Additions to Chapter 6 of the OpenGL 1.2.1 Specification (State and State Requests)

-- Section 6.1.12 "Saving and Restoring State"

Only the enables and vertex array state introduced by this extension can be pushed and popped.

See the attribute column in table X.5 for determining what vertex program state can be pushed and popped with PushAttrib, PopAttrib, PushClientAttrib, and PopClientAttrib.

The new evaluator enables in table 6.22 can also be pushed and popped.

-- NEW Section 6.1.13 "Vertex Program Queries"

"The commands

```
  void GetProgramParameterfvNV(enum target, uint index,
                               enum pname, float *params);
  void GetProgramParameterdvNV(enum target, uint index,
                               enum pname, double *params);
```

obtain the current program parameters for the given program target and parameter index into the array params. target must be VERTEX_PROGRAM_NV. pname must be PROGRAM_PARAMETER_NV. The INVALID_VALUE error is generated if index is greater than 95. Each program parameter is an array of four values.

The command

```
  void GetProgramivNV(uint id, enum pname, int *params);
```

obtains program state named by pname for the program named id in the array params. pname must be one of PROGRAM_TARGET_NV, PROGRAM_LENGTH_NV, or PROGRAM_RESIDENT_NV. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetProgramStringNV(uint id, enum pname,
                          ubyte *program);
```

obtains the program string for program id. pname must be PROGRAM_STRING_NV. n ubytes are returned into the array program where n is the length of the program in ubytes. GetProgramivNV with PROGRAM_LENGTH_NV can be used to query the length of a program's string. The INVALID_OPERATION error is generated if the program named id does not exist.

The command

```
  void GetTrackMatrixivNV(enum target, uint address,
                          enum pname, int *params);
```

obtains the matrix tracking state named by pname for the specified address in the array params. target must be VERTEX_PROGRAM_NV. pname must be either TRACK_MATRIX_NV or TRACK_MATRIX_TRANSFORM_NV. If the matrix tracked is a texture matrix, TEXTUREi_ARB is returned (never TEXTURE) where i indicates the texture unit of the particular tracked texture matrix. The INVALID_VALUE error is generated if address is not divisible by four and is not less than 96.

The commands

```
  void GetVertexAttribdvNV(uint index, enum pname, double *params);
```

```
  void GetVertexAttribfvNV(uint index, enum pname, float *params);
  void GetVertexAttribivNV(uint index, enum pname, int *params);
```

obtain the vertex attribute state named by pname for the vertex attribute numbered index. pname must be one of ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, ATTRIB_ARRAY_TYPE_NV, or CURRENT_ATTRIB_NV. Note that all the queries except CURRENT_ATTRIB_NV return client state. The INVALID_VALUE error is generated if index is greater than 15, or if index is zero and pname is CURRENT_ATTRIB_NV.

The command

```
  void GetVertexAttribPointervNV(uint index,
                                 enum pname, void **pointer);
```

obtains the pointer named pname in the array params for vertex attribute numbered index. pname must be ATTRIB_ARRAY_POINTER_NV. The INVALID_VALUE error is generated if index greater than 15.

The command

```
  boolean IsProgramNV(uint id);
```

returns TRUE if program is the name of a program object. If program is zero or is a non-zero value that is not the name of a program object, or if an error condition occurs, IsProgramNV returns FALSE. A name returned by GenProgramsNV but not yet loaded with a program is not the name of a program object."

-- NEW Section 6.1.14 "Querying Current Matrix State"

"Instead of providing distinct symbolic tokens for querying each matrix and matrix stack depth, the symbolic tokens CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV in conjunction with the GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev return the respective state of the current matrix given the current matrix mode.

Querying CURRENT_MATRIX_NV and CURRENT_MATRIX_STACK_DEPTH_NV is the only means for querying the matrix and matrix stack depth of the tracking matrices described in section 2.14.5."

Additions to Appendix A of the OpenGL 1.2.1 Specification (Invariance)

Add the following rule:

"Rule X Vertex program and vertex state program instructions not relevant to the calculation of any result must have no effect on that result.

Rules X+1 Vertex program and vertex state program instructions relevant to the calculation of any result must always produce the identical result. In particular, the same instruction with the same source inputs must produce the identical result whether executed by a vertex program or a vertex state program.

Instructions relevant to the calculation of a result are any instructions in a sequence of instructions that eventually determine the source values for the calculation under consideration.

There is no guaranteed invariance between vertices transformed by conventional GL vertex transform mode and vertices transformed by vertex program mode. Multi-pass rendering algorithms that require rendering invariances to operate correctly should not mix conventional

GL vertex transform mode with vertex program mode for different rendering passes. However such algorithms will operate correctly if the algorithms limit themselves to a single mode of vertex transformation."

Additions to the AGL/GLX/WGL Specifications

Program objects are shared between AGL/GLX/WGL rendering contexts if and only if the rendering contexts share display lists. No change is made to the AGL/GLX/WGL API.

Dependencies on EXT_vertex_weighting

If the EXT_vertex_weighting extension is not supported, there is no aliasing between vertex attribute 1 and the current vertex weight. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Dependencies on EXT_point_parameters

When EXT_point_parameters is supported, the amended discussion of point size determination should be further amended with the language from the EXT_point_parameters specification though the point parameters functionality only applies when vertex program mode is disabled.

Even if the EXT_point_parameters extension is not supported, the PSIZ vertex result register must operate as specified.

Dependencies on ARB_multitexture

ARB_multitexture is required to support NV_vertex_program and the value of MAX_TEXTURE_UNITS_ARB must be at least 2. If more than 8 texture units are supported, only the first 8 texture units can be assigned texture coordinates when vertex program mode is enabled. Texture units beyond 8 are implicitly disabled when vertex program mode is enabled.

Dependencies on EXT_fog_coord

If the EXT_fog_coord extension is not supported, there is no aliasing between vertex attribute 5 and the current fog coordinate. Replace the contents of the last three columns in row 5 of table X.2 with dashes.

Even if the EXT_fog_coord extension is not supported, the FOGC vertex result register must operate as specified. Note that the FOGC vertex result register behaves identically to the EXT_fog_coord extension's FOG_COORDINATE_SOURCE_EXT being FOG_COORDINATE_EXT. This means that the functionality of EXT_fog_coord is required to implement NV_vertex_program even if the EXT_fog_coord extension is not supported.

If the EXT_fog_coord extension is supported, the state of FOG_COORDINATE_SOURCE_EXT only applies when vertex program mode is disabled and the discussion in section 3.10 is further amended by the discussion of FOG_COORDINATE_SOURCE_EXT in the EXT_fog_coord specification.

Dependencies on EXT_secondary_color

If the EXT_secondary_color extension is not supported, there is no aliasing between vertex attribute 4 and the current secondary color.

Replace the contents of the last three columns in row 4 of table X.2 with dashes.

Even if the EXT_secondary_color extension is not supported, the COL1 and BFC1 vertex result registers must operate as specified. These vertex result registers are required to implement OpenGL 1.2's separate specular mode within a vertex program.

GLX Protocol

Forty-five new GL commands are added.

The following thirty-five rendering commands are sent to the sever as part of a glXRender request:

BindProgramNV

2	12		rendering command length
2	4180		rendering command opcode
4	ENUM		target
4	CARD32		id

ExecuteProgramNV

2	12+4*n		rendering command length
2	4181		rendering command opcode
4	ENUM		target
	0x8621	n=4	GL_VERTEX_STATE_PROGRAM_NV
	else	n=0	command is erroneous
4	CARD32		id
4*n	LISTofFLOAT32		params

RequestResidentProgramsNV

2	8+4*n	rendering command length
2	4182	rendering command opcode
4	INT32	n
n*4	CARD32	programs

LoadProgramNV

2	16+n+p	rendering command length
2	4183	rendering command opcode
4	ENUM	target
4	CARD32	id
4	INT32	len
n	LISTofCARD8	n
p		unused, p=pad(n)

ProgramParameter4fvNV

2	32	rendering command length
2	4184	rendering command opcode
4	ENUM	target
4	CARD32	index
4	FLOAT32	params[0]
4	FLOAT32	params[1]
4	FLOAT32	params[2]
4	FLOAT32	params[3]

ProgramParameter4dvNV

2	44	rendering command length
2	4185	rendering command opcode
4	ENUM	target
4	CARD32	index
8	FLOAT64	params[0]
8	FLOAT64	params[1]
8	FLOAT64	params[2]

```
    8           FLOAT64         params[3]

ProgramParameters4fvNV
    2           16+16*n         rendering command length
    2           4186            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           CARD32          n
    16*n        FLOAT32         params

ProgramParameters4dvNV
    2           16+32*n         rendering command length
    2           4187            rendering command opcode
    4           ENUM            target
    4           CARD32          index
    4           CARD32          n
    32*n        FLOAT64         params

TrackMatrixNV
    2           20              rendering command length
    2           4188            rendering command opcode
    4           ENUM            target
    4           CARD32          address
    4           ENUM            matrix
    4           ENUM            transform

VertexAttribPointerNV is an entirely client-side command

VertexAttrib1svNV
    2           12              rendering command length
    2           4189            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2                           unused

VertexAttrib2svNV
    2           12              rendering command length
    2           4190            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]

VertexAttrib3svNV
    2           12              rendering command length
    2           4191            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2                           unused

VertexAttrib4svNV
    2           12              rendering command length
    2           4192            rendering command opcode
    4           CARD32          index
    2           INT16           v[0]
    2           INT16           v[1]
    2           INT16           v[2]
    2           INT16           v[3]

VertexAttrib1fvNV
    2           12              rendering command length
    2           4193            rendering command opcode
```

```
    4           CARD32          index
    4           FLOAT32         v[0]

VertexAttrib2fvNV
    2           16              rendering command length
    2           4194            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]

VertexAttrib3fvNV
    2           20              rendering command length
    2           4195            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]

VertexAttrib4fvNV
    2           24              rendering command length
    2           4196            rendering command opcode
    4           CARD32          index
    4           FLOAT32         v[0]
    4           FLOAT32         v[1]
    4           FLOAT32         v[2]
    4           FLOAT32         v[3]

VertexAttrib1dvNV
    2           16              rendering command length
    2           4197            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]

VertexAttrib2dvNV
    2           24              rendering command length
    2           4198            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]

VertexAttrib3dvNV
    2           32              rendering command length
    2           4199            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]

VertexAttrib4dvNV
    2           40              rendering command length
    2           4200            rendering command opcode
    4           CARD32          index
    8           FLOAT64         v[0]
    8           FLOAT64         v[1]
    8           FLOAT64         v[2]
    8           FLOAT64         v[3]

VertexAttrib4ubvNV
    2           12              rendering command length
    2           4201            rendering command opcode
    4           CARD32          index
    1           CARD8           v[0]
    1           CARD8           v[1]
```

```
    1           CARD8           v[2]
    1           CARD8           v[3]

VertexAttribs1svNV
    2           12+2*n+p        rendering command length
    2           4202            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    2*n         INT16           v
    p                           unused, p=pad(2*n)

VertexAttribs2svNV
    2           12+4*n          rendering command length
    2           4203            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    4*n         INT16           v

VertexAttribs3svNV
    2           12+6*n+p        rendering command length
    2           4204            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    6*n         INT16           v
    p                           unused, p=pad(6*n)

VertexAttribs4svNV
    2           12+8*n          rendering command length
    2           4205            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    8*n         INT16           v

VertexAttribs1fvNV
    2           12+4*n          rendering command length
    2           4206            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    4*n         FLOAT32         v

VertexAttribs2fvNV
    2           12+8*n          rendering command length
    2           4207            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    8*n         FLOAT32         v

VertexAttribs3fvNV
    2           12+12*n         rendering command length
    2           4208            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    12*n        FLOAT32         v

VertexAttribs4fvNV
    2           12+16*n         rendering command length
    2           4209            rendering command opcode
    4           CARD32          index
    4           CARD32          n
    16*n        FLOAT32         v

VertexAttribs1dvNV
    2           12+8*n          rendering command length
```

```
        2           4210            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        8*n         FLOAT64         v

    VertexAttribs2dvNV
        2           12+16*n         rendering command length
        2           4211            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        16*n        FLOAT64         v

    VertexAttribs3dvNV
        2           12+24*n         rendering command length
        2           4212            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        24*n        FLOAT64         v

    VertexAttribs4dvNV
        2           12+32*n         rendering command length
        2           4213            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        32*n        FLOAT64         v

    VertexAttribs4ubvNV
        2           12+4*n          rendering command length
        2           4214            rendering command opcode
        4           CARD32          index
        4           CARD32          n
        4*n         CARD8           v
```

The remaining twelve commands are non-rendering commands. These commands are sent separately (i.e., not as part of a glXRender or glXRenderLarge request), using the glXVendorPrivateWithReply request:

```
    AreProgramsResidentNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           1293            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           (n+p)/4         reply length
        4           BOOL32          return value
        20                          unused
        n           LISTofBOOL      programs
        p                           unused, p=pad(n)

    DeleteProgramsNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           4+n             request length
        4           1294            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           INT32           n
        n*4         LISTofCARD32    programs
```

```
GenProgramsNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           4               request length
    4           1295            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           n
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           n               reply length
    24                          unused
    n*4         LISTofCARD322   programs

GetProgramParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1296            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params

GetProgramParameterdvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1297            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    8           FLOAT64         params
```

```
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params

GetProgramivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1298            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetProgramStringNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1299            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          id
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           (n+p)/4         reply length
    4                           unused
    4           CARD32          n
    16                          unused
    n           STRING          program
    p                           unused, p=pad(n)

GetTrackMatrixivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1300            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            target
    4           CARD32          address
    4           ENUM            pname
  =>
    1           1               reply
```

```
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

Note that ATTRIB_ARRAY_SIZE_NV, ATTRIB_ARRAY_STRIDE_NV, and
ATTRIB_ARRAY_TYPE_NV may be queried by GetVertexAttribNV but
return client-side state.

GetVertexAttribdvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1301            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n*2)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:

    8           FLOAT64         params
    8                           unused

    otherwise this follows:

    16                          unused
    n*8         LISTofFLOAT64   params

GetVertexAttribfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1302            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           INT32           index
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m=(n==1?0:n)
    4                           unused
    4           CARD32          n

    if (n=1) this follows:
```

```
            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params

        GetVertexAttribivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           5               request length
            4           1303            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           index
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m=(n==1?0:n)
            4                           unused
            4           CARD32          n

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetVertexAttribPointervNV is an entirely client-side command

        IsProgramNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           4               request length
            4           1304            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           INT32           n
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           0               reply length
            4           BOOL32          return value
            20                          unused
```

Errors

The error INVALID_VALUE is generated if VertexAttribNV is called where index is greater than 15.

The error INVALID_VALUE is generated if any ProgramParameterNV has an index is greater than 95.

The error INVALID_VALUE is generated if VertexAttribPointerNV is called where index is greater than 15.

The error INVALID_VALUE is generated if VertexAttribPointerNV
is called where size is not one of 1, 2, 3, or 4.

The error INVALID_VALUE is generated if VertexAttribPointerNV
is called where stride is negative.

The error INVALID_OPERATION is generated if VertexAttribPointerNV
is called where type is UNSIGNED_BYTE and size is not 4.

The error INVALID_VALUE is generated if LoadProgramNV is used to load a
program with an id of zero.

The error INVALID_OPERATION is generated if LoadProgramNV is used
to load an id that is currently loaded with a program of a different
program target.

The error INVALID_OPERATION is generated if the program passed to
LoadProgramNV fails to load because it is not syntactically correct
based on the specified target. The value of PROGRAM_ERROR_POSITION_NV
is still updated when this error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a
target of VERTEX_PROGRAM_NV and the specified program fails to
load because it does not write the HPOS register at least once.
The value of PROGRAM_ERROR_POSITION_NV is still updated when this
error is generated.

The error INVALID_OPERATION is generated if LoadProgramNV has a target
of VERTEX_STATE_PROGRAM_NV and the specified program fails to load
because it does not write at least one program parameter register.
The value of PROGRAM_ERROR_POSITION_NV is still updated when this
error is generated.

The error INVALID_OPERATION is generated if the vertex program
or vertex state program passed to LoadProgramNV fails to load
because it contains more than 128 instructions. The value of
PROGRAM_ERROR_POSITION_NV is still updated when this error is
generated.

The error INVALID_OPERATION is generated if a program is loaded with
LoadProgramNV for id when id is currently loaded with a program of
a different target.

The error INVALID_OPERATION is generated if BindProgramNV attempts
to bind to a program name that is not a vertex program (for example,
if the program is a vertex state program).

The error INVALID_VALUE is generated if GenProgramsNV is called
where n is negative.

The error INVALID_VALUE is generated if AreProgramsResidentNV is
called and any of the queried programs are zero or do not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes
a program that does not exist.

The error INVALID_OPERATION is generated if ExecuteProgramNV executes
a program that is not a vertex state program.

The error INVALID_OPERATION is generated if Begin, RasterPos, or a
command that performs an explicit Begin is called when vertex program
mode is enabled and the currently bound vertex program writes program
parameters that are currently being tracked.

The error INVALID_OPERATION is generated if ExecuteProgramNV is called and the vertex state program to execute writes program parameters that are currently being tracked.

The error INVALID_VALUE is generated if TrackMatrixNV has a target of VERTEX_PROGRAM_NV and attempts to track an address is not a multiple of four.

The error INVALID_VALUE is generated if GetProgramParameterNV is called to query an index greater than 95.

The error INVALID_VALUE is generated if GetVertexAttribNV is called to query an <index> greater than 15, or if <index> is zero and <pname> is CURRENT_ATTRIB_NV.

The error INVALID_VALUE is generated if GetVertexAttribPointervNV is called to query an index greater than 15.

The error INVALID_OPERATION is generated if GetProgramivNV is called and the program named id does not exist.

The error INVALID_OPERATION is generated if GetProgramStringNV is called and the program named <program> does not exist.

The error INVALID_VALUE is generated if GetTrackMatrixivNV is called with an <address> that is not divisible by four and not less than 96.

The error INVALID_VALUE is generated if AreProgramsResidentNV, DeleteProgramsNV, GenProgramsNV, or RequestResidentProgramsNV are called where <n> is negative.

The error INVALID_VALUE is generated if LoadProgramNV is called where <len> is negative.

The error INVALID_VALUE is generated if ProgramParameters4dvNV or ProgramParameters4fvNV are called where <count> is negative.

The error INVALID_VALUE is generated if VertexAttribs{1,2,3,4}{d,f,s}vNV is called where <count> is negative.

New State

update table 6.22 (page 212) so that all the "9"s are "25"s because there are 9 conventional map targets and 16 vertex attribute map targets making a total of 25.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_PROGRAM_NV	B	IsEnabled	False	vertex program enable	2.10	enable
VERTEX_PROGRAM_POINT_SIZE_NV	B	IsEnabled	False	program-specified point size mode	2.14.1.5	enable
VERTEX_PROGRAM_TWO_SIDE_NV	B	IsEnabled	False	two-sided color mode	2.14.1.5	enable
PROGRAM_ERROR_POSITION_NV	Z	GetIntegerv	-1	last program error position	2.14.1.7	-
PROGRAM_PARAMETER_NV	96xR4	GetProgramParameterNV	(0,0,0,0)	program parameters	2.14.1.2	-
CURRENT_ATTRIB_NV	16xR4	GetVertexAttribNV but zero cannot be queried, aliased with per-vertex parameters	see 2.14.6	vertex attributes	2.14.1.1	current
TRACK_MATRIX_NV	24xZ8+	GetTrackMatrixivNV	NONE	track matrix	2.14.5	-
TRACK_MATRIX_TRANSFORM_NV	24xZ8+	GetTrackMatrixivNV	IDENTITY_NV	track matrix transform	2.14.5	-
VERTEX_PROGRAM_BINDING_NV	Z+	GetIntegerv	0	bound vertex program	2.14.1.8	-
VERTEX_ATTRIB_ARRAYn_NV	16xB	IsEnabled	False	vertex attrib array enable	2.14.3	vertex-array
ATTRIB_ARRAY_SIZE_NV	16xZ	GetVertexAttribNV	4	vertex attrib array size	2.14.3	vertex-array

ATTRIB_ARRAY_STRIDE_NV	16xZ+	GetVertexAttribNV	0	vertex attrib array stride	2.14.3	vertex-array
ATTRIB_ARRAY_TYPE_NV	16xZ4	GetVertexAttribNV	FLOAT	vertex attrib array type	2.14.3	vertex-array

Table X.5. New State Introduced by NV_vertex_program.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
PROGRAM_TARGET_NV	Z2	GetProgramivNV	0	program target	6.1.13	-
PROGRAM_LENGTH_NV	Z+	GetProgramivNV	0	program length	6.1.13	-
PROGRAM_RESIDENT_NV	Z2	GetProgramivNV	False	program residency	6.1.13	-
PROGRAM_STRING_NV	ubxn	GetProgramStringNV	""	program string	6.1.13	-

Table X.6. Program Object State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
-	12xR4	-	(0,0,0,0)	temporary registers	2.14.1.4	-
-	15xR4	-	(0,0,0,1)	vertex result registers	2.14.1.4	-
	Z4	-	(0,0,0,0)	vertex program address register	2.14.1.3	-

Table X.7. Vertex Program Per-vertex Execution State.

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_MATRIX_STACK_DEPTH_NV	m*Z+	GetIntegerv	1	current stack depth	6.1.14	-
CURRENT_MATRIX_NV	m*n*xM^4	GetFloatv	Identity	current matrix	6.1.14	-

Table X.8. Current matrix state where m is the total number of matrices including texture matrices and tracking matrices and n is the number of matrices on each particular matrix stack. Note that this state is aliased with existing matrix state.

New Implementation Dependent State

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TRACK_MATRIX_STACK_DEPTH_NV	Z+	GetIntegerv	1	maximum tracking matrix stack depth	2.14.5	-
MAX_TRACK_MATRICES_NV	Z+	GetIntegerv	8 (not to exceed 32)	maximum number of tracking matrices	2.14.5	-

Table X.9. New Implementation-Dependent Values Introduced by NV_vertex_program.

Revision History

Version 1.1:

Added normalization example to Issues.

Fix explanation of EXP and ARL floor equivalence.

Clarify that vertex state programs fail if they load more than one vertex attribute (though only one is possible).

Version 1.2

Add GLX protocol for VertexAttrib4ubvNV and VertexAttribs4ubvNV

Add issue about TrackMatrixNV transform behavior with example

Fix the C code specifying VertexAttribsvNV

Version 1.3

Dropped support for INT typed vertex attrib arrays.

Clarify that when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is enabled, the vertex attrib 0 array command is executed last. However when ArrayElement is executed and vertex program mode is enabled and the vertex attrib 0 array is disabled and the vertex array is enabled, the vertex array command is executed last.

Version 1.4

  Allow TEXTUREi_ARB for the track matrix. This allows matrix
  tracking of a particular texture matrix without reference to active
  texture (set by glActiveTextureARB) state.

  Early NVIDIA drivers (prior to October 5, 2001) have a bug
  in their handling of tracking matrices specified with TEXTURE.
  Rather than tracking the particular texture matrix indicated
  by the active texture state when TrackMatrixNV is called, these
  early drivers incorrectly track matrix the active texture's texture
  matrix _at track matrix validation time_. In practice this means,
  every tracked matrix defined with TEXTURE tracks the same matrix
  values; you cannot track distinct texture matrices at the same
  time and the texture matrix you actually track depends on the
  active texture matrix at validation time. This is a driver bug.

  Drivers after October 5, 2001 properly track the texture matrix
  specified by active texture when TrackMatrix is called.

  The new correct drivers can be distinguished from the old drivers
  at run time with the following code:

```
    while (glGetError() != GL_NO_ERROR);  // Clear any pre-existing OpenGL errors.
    glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_TEXTURE0_ARB, GL_IDENTITY_NV);
    if (glGetError() != GL_NO_ERROR) {
      // Old buggy pre-version 1.4 drivers with GL_TEXTURE
      // glTrackMatrixNV bug.
    } else {
      // Correct new version 1.4 drivers (or later) with GL_TEXTURE
      // glTrackMatrixNV bug fixed and GL_TEXTUREi_NV support.

      // Note: you may want to untrack the matrix at this point.
    }
```

Version 1.5

  Earlier versions of this specification claimed for
  GetVertexAttribARB that it is an error to query any vertex attrib
  state for vertex attrib array zero. In fact, it should only be
  an error to query the CURRENT_ATTRIB_ARB state for vertex attrib
  zero; the size, stride, and type of vertex attrib array zero may
  be queried. Version 1.5 specifies the correct behavior.

  Early NVIDIA drivers (prior to January 11, 2002) did not implement
  generate error when querying vertex attrib array zero state (ie,
  did the right thing for size, stride, and type) but not create an
  error when querying the current attribute values for vertex attrib
  array zero either.

Version 1.6

  GLX opcodes and vendorpriv values assigned.