Name

EXT_gpu_shader4

Name Strings

GL_EXT_gpu_shader4

Contact

Barthold Lichtenbelt, NVIDIA (blichtenbelt 'at' nvidia.com)
Pat Brown, NVIDIA (pbrown 'at' nvidia.com)

Status

Multi vendor extension

Shipping for GeForce 8 Series (November 2006)

Version

Last Modified Date: 12/14/2009
Author revision: 16

Number

326

Dependencies

OpenGL 2.0 is required.

This extension is written against the OpenGL 2.0 specification and version 1.10.59 of the OpenGL Shading Language specification.

This extension trivially interacts with ARB_texture_rectangle.

This extension trivially interacts with GL_EXT_texture_array.

This extension trivially interacts with GL_EXT_texture_integer.

This extension trivially interacts with GL_EXT_geometry_shader4

This extension trivially interacts with GL_EXT_texture_buffer_object.

NV_primitive_restart trivially affects the definition of this extension.

ARB_color_buffer_float affects the definition of this extension.
EXT_draw_instanced affects the definition of this extension.

Overview

This extension provides a set of new features to the OpenGL Shading Language and related APIs to support capabilities of new hardware. In particular, this extension provides the following functionality:

* New texture lookup functions are provided that allow shaders to access individual texels using integer coordinates referring to the texel location and level of detail. No filtering is performed. These functions allow applications to use textures as one-, two-, and three-dimensional arrays.

* New texture lookup functions are provided that allow shaders to query the dimensions of a specific level-of-detail image of a texture

object.

* New texture lookup functions variants are provided that allow shaders to pass a constant integer vector used to offset the texel locations used during the lookup to assist in custom texture filtering operations.

* New texture lookup functions are provided that allow shaders to access one- and two-dimensional array textures. The second, or third, coordinate is used to select the layer of the array to access.

* New "Grad" texture lookup functions are provided that allow shaders to explicitely pass in derivative values which are used by the GL to compute the level-of-detail when performing a texture lookup.

* A new texture lookup function is provided to access a buffer texture.

* The existing absolute LOD texture lookup functions are no longer restricted to the vertex shader only.

* The ability to specify and use cubemap textures with a DEPTH_COMPONENT internal format. This also enables shadow mapping on cubemaps. The 'q' coordinate is used as the reference value for comparisons. A set of new texture lookup functions is provided to lookup into shadow cubemaps.

* The ability to specify if varying variables are interpolated in a non-perspective correct manner, if they are flat shaded or, if multi-sampling, if centroid sampling should be performed.

* Full signed integer and unsigned integer support in the OpenGL Shading Language:

    - Integers are defined as 32 bit values using two's complement.

    - Unsigned integers and vectors thereof are added.

    - New texture lookup functions are provided that return integer values. These functions are to be used in conjunction with new texture formats whose components are actual integers, rather than integers that encode a floating-point value. To support these lookup functions, new integer and unsigned-integer sampler types are introduced.

    - Integer bitwise operators are now enabled.

    - Several built-in functions and operators now operate on integers or vectors of integers.

    - New vertex attribute functions are added that load integer attribute data and can be referenced in a vertex shader as integer data.

    - New uniform loading commands are added to load unsigned integer data.

    - Varying variables can now be (unsigned) integers. If declared as such, they have to be flat shaded.

    - Fragment shaders can define their own output variables, and declare them to be of type floating-point, integer or unsigned integer. These variables are bound to a fragment color index with the new API command BindFragDataLocationEXT(), and directed to buffers using the existing DrawBuffer or DrawBuffers API commands.

* Added new built-in functions truncate() and round() to the shading language.

* A new built-in variable accessible from within vertex shaders that holds the index <i> implicitly passed to ArrayElement to specify the vertex. This is called the vertex ID.

* A new built-in variable accessible from within fragment and geometry shaders that hold the index of the currently processed primitive. This is called the primitive ID.

This extension also briefly mentions a new shader type, called a geometry shader. A geometry shader is run after vertices are transformed, but before clipping. A geometry shader begins with a single primitive (point, line, triangle. It can read the attributes of any of the vertices in the primitive and use them to generate new primitives. A geometry shader has a fixed output primitive type (point, line strip, or triangle strip) and emits vertices to define a new primitive. Geometry shaders are discussed in detail in the GL_EXT_geometry_shader4 specification.

New Procedures and Functions

```
void VertexAttribI1iEXT(uint index, int x);
void VertexAttribI2iEXT(uint index, int x, int y);
void VertexAttribI3iEXT(uint index, int x, int y, int z);
void VertexAttribI4iEXT(uint index, int x, int y, int z, int w);

void VertexAttribI1uiEXT(uint index, uint x);
void VertexAttribI2uiEXT(uint index, uint x, uint y);
void VertexAttribI3uiEXT(uint index, uint x, uint y, uint z);
void VertexAttribI4uiEXT(uint index, uint x, uint y, uint z,
                         uint w);

void VertexAttribI1ivEXT(uint index, const int *v);
void VertexAttribI2ivEXT(uint index, const int *v);
void VertexAttribI3ivEXT(uint index, const int *v);
void VertexAttribI4ivEXT(uint index, const int *v);

void VertexAttribI1uivEXT(uint index, const uint *v);
void VertexAttribI2uivEXT(uint index, const uint *v);
void VertexAttribI3uivEXT(uint index, const uint *v);
void VertexAttribI4uivEXT(uint index, const uint *v);

void VertexAttribI4bvEXT(uint index, const byte *v);
void VertexAttribI4svEXT(uint index, const short *v);
void VertexAttribI4ubvEXT(uint index, const ubyte *v);
void VertexAttribI4usvEXT(uint index, const ushort *v);

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

void GetVertexAttribIivEXT(uint index, enum pname, int *params);
void GetVertexAttribIuivEXT(uint index, enum pname,
                            uint *params);

void Uniform1uiEXT(int location, uint v0);
void Uniform2uiEXT(int location, uint v0, uint v1);
void Uniform3uiEXT(int location, uint v0, uint v1, uint v2);
void Uniform4uiEXT(int location, uint v0, uint v1, uint v2,
                   uint v3);

void Uniform1uivEXT(int location, sizei count, const uint *value);
void Uniform2uivEXT(int location, sizei count, const uint *value);
void Uniform3uivEXT(int location, sizei count, const uint *value);
```

```
    void Uniform4uivEXT(int location, sizei count, const uint *value);

    void GetUniformuivEXT(uint program, int location, uint *params);

    void BindFragDataLocationEXT(uint program, uint colorNumber,
                                 const char *name);
    int GetFragDataLocationEXT(uint program, const char *name);
```

New Tokens

Accepted by the <pname> parameters of GetVertexAttribdv, GetVertexAttribfv, GetVertexAttribiv, GetVertexAttribIuivEXT and GetVertexAttribIivEXT:

VERTEX_ATTRIB_ARRAY_INTEGER_EXT	0x88FD

Returned by the <type> parameter of GetActiveUniform:

SAMPLER_1D_ARRAY_EXT	0x8DC0
SAMPLER_2D_ARRAY_EXT	0x8DC1
SAMPLER_BUFFER_EXT	0x8DC2
SAMPLER_1D_ARRAY_SHADOW_EXT	0x8DC3
SAMPLER_2D_ARRAY_SHADOW_EXT	0x8DC4
SAMPLER_CUBE_SHADOW_EXT	0x8DC5
UNSIGNED_INT	0x1405
UNSIGNED_INT_VEC2_EXT	0x8DC6
UNSIGNED_INT_VEC3_EXT	0x8DC7
UNSIGNED_INT_VEC4_EXT	0x8DC8
INT_SAMPLER_1D_EXT	0x8DC9
INT_SAMPLER_2D_EXT	0x8DCA
INT_SAMPLER_3D_EXT	0x8DCB
INT_SAMPLER_CUBE_EXT	0x8DCC
INT_SAMPLER_2D_RECT_EXT	0x8DCD
INT_SAMPLER_1D_ARRAY_EXT	0x8DCE
INT_SAMPLER_2D_ARRAY_EXT	0x8DCF
INT_SAMPLER_BUFFER_EXT	0x8DD0
UNSIGNED_INT_SAMPLER_1D_EXT	0x8DD1
UNSIGNED_INT_SAMPLER_2D_EXT	0x8DD2
UNSIGNED_INT_SAMPLER_3D_EXT	0x8DD3
UNSIGNED_INT_SAMPLER_CUBE_EXT	0x8DD4
UNSIGNED_INT_SAMPLER_2D_RECT_EXT	0x8DD5
UNSIGNED_INT_SAMPLER_1D_ARRAY_EXT	0x8DD6
UNSIGNED_INT_SAMPLER_2D_ARRAY_EXT	0x8DD7
UNSIGNED_INT_SAMPLER_BUFFER_EXT	0x8DD8

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

MIN_PROGRAM_TEXEL_OFFSET_EXT	0x8904
MAX_PROGRAM_TEXEL_OFFSET_EXT	0x8905

Additions to Chapter 2 of the OpenGL 2.0 Specification (OpenGL Operation)

Modify Section 2.7 "Vertex Specification", p.20

Insert before last paragraph, p.22:

The VertexAttrib* commands described so far should not be used to load data for vertex attributes declared as signed or unsigned integers or vectors thereof in a vertex shader. If they are used to load signed or unsigned integer vertex attributes, the value in those attributes are undefined. Instead use the commands

```
  void VertexAttribI[1234]{i,ui}EXT(uint index, T values);
```

```
  void VertexAttribI[1234]{i,ui}vEXT(uint index, T values);
  void VertexAttribI4{b,s,ub,us}vEXT(uint index, T values);
```

to specify fixed-point attributes that are not converted to floating-point. These attributes can be accessed in vertex shaders that declare attributes as signed or unsigned integers or vectors. The VertexAttribI4* commands extend the data passed in to a full signed or unsigned integer. If a VertexAttribI* command is used that does not match the type of the attribute declared in a vertex shader, the values in the attributes are undefined. This means that the unsigned versions of the VertexAttribI* commands need to be used to load data for unsigned integer vertex attributes or vectors, and the signed versions of the VertexAttribI* commands for signed integer vertex attributes or vectors. Note that this also means that the VertexAttribI* commands should not be used to load data for a vertex attribute declared as a float, float vector or matrix, otherwise their values are undefined.

Insert at end of function list, p.24:

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

(modify last paragraph, p.24) The <index> parameter in the VertexAttribPointer and VertexAttribIPointerEXT commands identify the generic vertex attribute array being described. The error INVALID_VALUE is generated if <index> is greater than or equal to MAX_VERTEX_ATTRIBS. Generic attribute arrays with integer <type> arguments can be handled in one of three ways: converted to float by normalizing to [0,1] or [-1,1] as specified in table 2.9, converted directly to float, or left as integers. Data for an array specified by VertexAttribPointer will be converted to floating-point by normalizing if the <normalized> parameter is TRUE, and converted directly to floating-point otherwise. Data for an array specified by VertexAttribIPointerEXT will always be left as integer values.

(modify Table 2.4, p. 25)

Command	Sizes	Integer Handling	Types
VertexPointer	2,3,4	cast	...
NormalPointer	3	normalize	...
ColorPointer	3,4	normalize	...
SecondaryColorPointer	3	normalize	...
IndexPointer	1	cast	...
FogCoordPointer	1	n/a	...
TexCoordPointer	1,2,3,4	cast	...
EdgeFlagPointer	1	integer	...
VertexAttribPointer	1,2,3,4	flag	...
VertexAttribIPointerEXT	1,2,3,4	integer	byte, ubyte, short, ushort, int, uint

Table 2.4: Vertex array sizes (values per vertex) and data types. The "integer handling" column indicates how fixed-point data types are handled: "cast" means that they converted to floating-point directly, "normalize" means that they are converted to floating-point by normalizing to [0,1] (for unsigned types) or [-1,1] (for signed types), "integer" means that they remain as integer values, and "flag" means that either "cast" or "normalized" applies, depending on the setting of the <normalized> flag in VertexAttribPointer.

(modify end of pseudo-code, pp. 27-28)

```
  for (j = 1; j < genericAttributes; j++) {
    if (generic vertex attribute j array enabled) {
```

```
      if (generic vertex attribute j array is a pure integer array)
      {
        VertexAttribI[size][type]vEXT(j, generic vertex attribute j
                                      array element i);
      } else if (generic vertex attribute j array normalization
                 flag is set and <type> is not FLOAT or DOUBLE) {
        VertexAttrib[size]N[type]v(j, generic verex attribute j
                                   array element i);
      } else {
        VertexAttrib[size][type]v(j, generic verex attribute j
                                  array element i);
      }
    }
  }

  if (generic vertex attribute 0 array enabled) {
    if (generic vertex attribute 0 array is a pure integer array) {
      VertexAttribI[size][type]vEXT(0, generic verex attribute 0
                                   array element i);
    } else if (generic vertex attribute 0 array normalization flag
               is set and <type> is not FLOAT or DOUBLE) {
     VertexAttrib[size]N[type]v(0, generic verex attribute 0
                                array element i);
    } else {
      VertexAttrib[size][type]v(0, generic verex attribute 0
                                array element i);
    }
  }
```

Modify section 2.14.7, "Flatshading", p. 69

Add a new paragraph at the end of the section on p. 70 as follows:

If a vertex or geometry shader is active, the flat shading control described so far applies to the built-in varying variables gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and gl_BackSecondaryColor. Through the OpenGL Shading Language varying qualifier flat any vertex attribute can be flagged to be flat-shaded. See the OpenGL Shading Language Specification section 4.3.6 for more information.

Modify section 2.14.8, "Color and Associated Data Clipping", p. 71

Add to the end of this section:

For vertex shader varying variables specified to be interpolated without perspective correction (using the noperspective keyword), the value of t used to obtain the varying value associated with P will be adjusted to produce results that vary linearly in screen space.

Modify section 2.15.3, "Shader Variables", page 75

Add the following new return types to the description of GetActiveUniform on p. 81.

```
  SAMPLER_1D_ARRAY_EXT,
  SAMPLER_2D_ARRAY_EXT,
  SAMPLER_1D_ARRAY_SHADOW_EXT,
  SAMPLER_2D_ARRAY_SHADOW_EXT,
  SAMPLER_CUBE_SHADOW_EXT,
  SAMPLER_BUFFER_EXT,

  INT_SAMPLER_1D_EXT,
  INT_SAMPLER_2D_EXT,
  INT_SAMPLER_3D_EXT,
  INT_SAMPLER_CUBE_EXT,
```

```
  INT_SAMPLER_2D_RECT_EXT,
  INT_SAMPLER_1D_ARRAY_EXT,
  INT_SAMPLER_2D_ARRAY_EXT,
  INT_SAMPLER_BUFFER_EXT,

  UNSIGNED_INT,
  UNSIGNED_INT_VEC2_EXT,
  UNSIGNED_INT_VEC3_EXT,
  UNSIGNED_INT_VEC4_EXT,
  UNSIGNED_INT_SAMPLER_1D_EXT,
  UNSIGNED_INT_SAMPLER_2D_EXT,
  UNSIGNED_INT_SAMPLER_3D_EXT,
  UNSIGNED_INT_SAMPLER_CUBE_EXT,
  UNSIGNED_INT_SAMPLER_2D_RECT_EXT,
  UNSIGNED_INT_SAMPLER_1D_ARRAY_EXT,
  UNSIGNED_INT_SAMPLER_2D_ARRAY_EXT,
  UNSIGNED_INT_SAMPLER_BUFFER_EXT.
```

Add the following uniform loading command prototypes on p. 81 as follows:

```
  void Uniform{1234}uiEXT(int location, T value);
  void Uniform{1234}uivEXT(int location, sizei count, T value);
```

(add the following paragraph to the description of the above commands)

The Uniform*ui{v} commands will load count sets of one to four unsigned integer values into a uniform location defined as a unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

(change the first sentence of the last paragraph as follows)

When loading values for a uniform declared as a Boolean, the Uniform*i{v}, Uniform*ui{v} and Uniform*f{v} set of commands can be used to load boolean values.

Modify section 2.15.4 Shader execution, p. 84.

Add a new section "2.15.4.1 Shader Only Texturing" before the sub-section "Texture Access" on p. 85

This section describes texture functionality that is only accessible through vertex, geometry or fragment shaders. Also refer to the OpenGL Shading Language Specification, section 8.7 and Section 3.8 of the OpenGL 2.0 specification.

Note: For unextended OpenGL 2.0 and the OpenGL Shading Language version 1.20, all supported texture internal formats store unsigned integer values but return floating-point results in the range [0, 1] and are considered unsigned "normalized" integers. The ARB_texture_float extension introduces floating-point internal format where components are both stored and returned as floating-point values, and are not clamped. The EXT_texture_integer extension introduces formats that store either signed or unsigned integer values.

This extension defines additional OpenGL Shading Language texture lookup functions, see section 8.7 of the OpenGL Shading Language, that return either signed or unsigned integer values if the internal format of the texture is signed or unsigned, respectively.

Texel Fetches

The OpenGL Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer

coordinates passed to the texel fetch functions are used directly as the texel coordinates (i, j, k) into the texture image. This in turn means the texture image is point-sampled (no filtering is performed).

The level of detail accessed is computed by adding the specified level-of-detail parameter <lod> to the base level of the texture, level_base.

The texel fetch functions can not perform depth comparisons or access cube maps. Unlike filtered texel accesses, texel fetches do not support LOD clamping or any texture wrap mode, and require a mipmapped minification filter to access any level of detail other than the base level.

The results of the texel fetch are undefined:

* if the computed LOD is less than the texture's base level (level_base) or greater than the maximum level (level_max),

* if the computed LOD is not the texture's base level and the texture's minification filter is NEAREST or LINEAR,

* if the layer specified for array textures is negative or greater than the number of layers in the array texture,

* if the texel at (i,j,k) coordinates refer to a border texel outside the defined extents of the specified LOD, where

  i < -b_s, j < -b_s, k < -b_s,
  i >= w_s - b_s, j >= h_s - b_s, or k >= d_s - b_s,

  where the size parameters (w_s, h_s, d_s, and b_s) refer to the width, height, depth, and border size of the image, as in equations 3.15, 3.16, and 3.17, or

. if the texture being accessed is not complete (or cube complete for cubemaps).

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value <lod> passed in as an argument to the texture size functions is added to the level_base of the texture to determine a texture image level. The dimensions of that image level, excluding a possible border, are then returned. If the computed texture image level is outside the range [level_base, level_max], the results are undefined. When querying the size of an array texture, both the dimensions and the layer count are returned. Note that buffer textures do not support mipmapping, therefore the previous lod discussion does not apply to buffer textures

Make the section "Texture Access" a subsection of 2.15.4.1

Modify the first paragraph on p. 86 as follows:

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE.

```
    * The sampler used in a texture lookup function is one of the shadow
      sampler types, and the texture object's internal format is DEPTH
      COMPONENT, and the TEXTURE COMPARE MODE is NONE.

    * The sampler used in a texture lookup function is one of the shadow
      sampler types, and the texture object's internal format is not DEPTH
      COMPONENT.

  Add a new section "2.15.4.2 Shader Inputs" before "Position
  Invariance" on p. 86

  Besides having access to vertex attributes and uniform variables,
  vertex shaders can access the read-only built-in variables
  gl_VertexID and gl_InstanceID. The gl_VertexID variable holds the
  integer index <i> implicitly passed to ArrayElement() to specify
  the vertex. The variable gl_InstanceID holds the integer index of
  the current primitive in an instanced draw call.  See also section
  7.1 of the OpenGL Shading Language Specification.

  Add a new section "2.15.4.3 Shader Outputs"

  A vertex shader can write to built-in as well as user-defined varying
  variables. These values are expected to be interpolated across the
  primitive it outputs, unless they are specified to be flat shaded. Refer
  to section 2.15.3 and the OpenGL Shading Language specification sections
  4.3.6, 7.1 and 7.6 for more detail.

  The built-in output variables gl_FrontColor, gl_BackColor,
  gl_FrontSecondaryColor, and gl_BackSecondaryColor hold the front and back
  colors for the primary and secondary colors for the current vertex.

  The built-in output variable gl_TexCoord[] is an array and holds the set
  of texture coordinates for the current vertex.

  The built-in output variable gl_FogFragCoord is used as the "c" value, as
  described in section 3.10 "Fog" of the OpenGL 2.0 specification.

  The built-in special variable gl_Position is intended to hold the
  homogeneous vertex position. Writing gl_Position is optional.

  The built-in special variable gl_ClipVertex holds the vertex coordinate
  used in the clipping stage, as described in section 2.12 "Clipping" of the
  OpenGL 2.0 specification.

  The built in special variable gl_PointSize, if written, holds the size of
  the point to be rasterized, measured in pixels.

  Number section "Position Invariance", "Validation" and "Undefined
  Behavior" as sections 2.15.4.4, 2.15.4.5, and 2.15.4.6 respectively.

Additions to Chapter 3 of the OpenGL 2.0 Specification (Rasterization)

  Modify Section 3.8.1, Texture Image Specification, p. 150

  (modify 4th paragraph, p. 151 -- add cubemaps to the list of texture
  targets that can be used with DEPTH_COMPONENT textures)

  Textures with a base internal format of DEPTH_COMPONENT are supported by
  texture image specification commands only if <target> is TEXTURE_1D,
  TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_RECTANGLE_ARB, PROXY_TEXTURE_1D,
  PROXY_TEXTURE_2D, PROXY_TEXTURE_CUBE_MAP, or
  PROXY_TEXTURE_RECTANGLE_ARB. Using this format in conjunction with any
  other target will result in an INVALID_OPERATION error.
```

Delete Section 3.8.7, Texture Wrap Modes. (The language in this section is folded into updates to the following section, and is no longer needed here.)

Modify Section 3.8.8, Texture Minification:

(replace the last paragraph, p. 171): Let s(x,y) be the function that associates an s texture coordinate with each set of window coordinates (x,y) that lie within a primitive; define t(x,y) and r(x,y) analogously. Let

```
  u(x,y) = w_t * s(x,y) + offsetu_shader,
  v(x,y) = h_t * t(x,y) + offsetv_shader,
  w(x,y) = d_t * r(x,y) + offsetw_shader, and
```

where w_t, h_t, and d_t are as defined by equations 3.15, 3.16, and 3.17 with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is level_base. (offsetu_shader, offsetv_shader, offsetw_shader) is the texel offset specified in the OpenGL Shading Language texture lookup functions that support offsets. If the texture function used does not support offsets, or for fixed-function texture accesses, all three shader offsets are taken to be zero. For fixed-function texture accesses, all three shader offsets are taken to be zero. For a one-dimensional texture, define v(x,y) == 0 and w(x,y) === 0; for two-dimensional textures, define w(x,y) == 0.

After u(x,y), v(x,y), and w(x,y) are generated, they are clamped if the corresponding texture wrap modes are CLAMP or MIRROR_CLAMP_EXT. Let

```
  u'(x,y) = clamp(u(x,y), 0, w_t),       if TEXTURE_WRAP_S is CLAMP
            clamp(u(x,y), -w_t, w_t),    if TEXTURE_WRAP_S is
                                           MIRROR_CLAMP_EXT, or
            u(x,y),                      otherwise
  v'(x,y) = clamp(v(x,y), 0, w_t),       if TEXTURE_WRAP_T is CLAMP
            clamp(v(x,y), -w_t, w_t),    if TEXTURE_WRAP_T is
                                           MIRROR_CLAMP_EXT, or
            v(x,y),                      otherwise
  w'(x,y) = clamp(w(x,y), 0, w_t),       if TEXTURE_WRAP_R is CLAMP
            clamp(w(x,y), -w_t, w_t),    if TEXTURE_WRAP_R is
                                           MIRROR_CLAMP_EXT, or
            w(x,y),                      otherwise,
```

where clamp(<a>,<b>,<c>) returns <b> if <a> is less than <b>, <c> if a is greater than <c>, and <a> otherwise.

(start a new paragraph with "For a polygon, rho is given at a fragment with window coordinates...", and then continue with the original spec text.)

(replace text starting with the last paragraph on p. 172, continuing to the end of p. 174)

When lambda indicates minification, the value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When TEXTURE_MIN_FILTER is NEAREST, the texel in the image array of level level_base that is nearest (in Manhattan distance) to that specified by (s,t,r) is obtained. Let i, j, and k be integers such that:

```
  i = apply_wrap(floor(u'(x,y))),
  j = apply_wrap(floor(v'(x,y))), and
  k = apply_wrap(floor(w'(x,y))),
```

where the coordinate returned by apply_wrap() is as defined by Table X.19. The values of i, j, and k are then modified according to the texture wrap modes, as described in Table 3.19, to produce new values (i', j', and k'). For a three-dimensional texture, the texel at location (i,j,k) becomes the texture value. For a two-dimensional texture, k is irrelevant, and the texel at location (i,j) becomes the texture value. For a one-dimensional texture, j and k are irrelevant, and the texel at location i becomes the texture value.

Wrap mode	Result
CLAMP_TO_EDGE	clamp(coord, 0, size-1)
CLAMP_TO_BORDER	clamp(coord, -1, size)
CLAMP	{ clamp(coord, 0, size-1), { for NEAREST filtering { clamp(coord, -1, size), { for LINEAR filtering
REPEAT	mod(coord, size)
MIRROR_CLAMP_TO_EDGE_EXT	clamp(mirror(coord), 0, size-1)
MIRROR_CLAMP_TO_BORDER_EXT	clamp(mirror(size), 0, size)
MIRROR_CLAMP_EXT	{ clamp(mirror(coord), 0, size-1), { for NEAREST filtering { clamp(mirror(size), 0, size), { for LINEAR filtering
MIRRORED_REPEAT	(size-1) - mirror(mod(coord, 2*size)-size)

Table X.19: Texel location wrap mode application. mod(<a>,<b>) is defined to return <a>-<b>*floor(<a>/<b>), and mirror(<a>) is defined to return <a> if <a> is greater than or equal to zero or -(1+<a>) otherwise. The values of "wrap mode" and size are TEXTURE_WRAP_S and w_t, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t, for i, j, and k coordinates, respectively. The coordinate clamp and MIRROR_CLAMP_EXT depends on the filtering mode (NEAREST or LINEAR).

If the selected (i,j,k), (i,j), or i location refers to a border texel that satisfies any of the following conditions:

```
i < -b_s,
j < -b_s,
k < -b_s,
i >= w_t + b_s,
j >= h_t + b_s, or
j >= d_t + b_s,
```

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.15. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When TEXTURE_MIN_FILTER is LINEAR, a 2x2x2 cube of texels in the image array of level level_base is selected. Let:

```
i_0   = apply_wrap(floor(u' - 0.5)),
j_0   = apply_wrap(floor(v' - 0.5)),
k_0   = apply_wrap(floor(w' - 0.5)),
i_1   = apply_wrap(floor(u' - 0.5) + 1),
j_1   = apply_wrap(floor(v' - 0.5) + 1),
k_1   = apply_wrap(floor(w' - 0.5) + 1),
alpha = frac(u' - 0.5),
beta  = frac(v' - 0.5),
gamma = frac(w' - 0.5),
```

where frac(<x>) denotes the fractional part of <x>.

For a three-dimensional texture, the texture value tau is found as...

(replace last paragraph, p.174) For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

modify the last paragraph of section 3.8.8, p. 175, as follows:

The rules for NEAREST or LINEAR filtering are then applied to the selected array. Specifically, the coordinate (u,v,w) is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd'.

Modify the second paragraph on p. 176

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture valutes Tau1 and Tau2. Specifically, for level d1, the coordinate (u,v,w) is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd1'. For level d2 the coordinate (u', v', w') is computed as in equation 3.20a, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is 'd2'.

Modify the first paragraph of section 3.8.9 "Texture Magnification" as follows:

When lambda indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER and LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER, as described in the previous section, including the wrapping calculations. The level-of-detail level_base texture array is always used for magnification.

Modify Section 3.8.14, Texture Comparison Modes (p. 185)

(modify 2nd paragraph, p. 188, indicating that the Q texture coordinate is used for depth comparisons on cubemap textures)

Let D_t be the depth texture value, in the range [0, 1]. For fixed-function texture lookups, let R be the interpolated <r> texture coordinate, clamped to the range [0, 1]. For texture lookups generated by an OpenGL Shading Language lookup function, let R be the reference value for depth comparisons provided in the lookup function, also clamped to [0, 1]. Then the effective texture value L_t, I_t, or A_t is computed as follows:

Modify section 3.11, Fragment Shaders, p. 193

Modify the third paragraph on p. 194 as follows:

Additionally, when a vertex shader is active, it may define one or more varying variables (see section 2.15.3 and the OpenGL Shading Language Specification). These values are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

Add the following paragraph to the end of section 3.11.1, p. 194

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations.

Varying out variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. The subsection 'Shader Outputs' of the next section describes API how to direct these values to buffers.

Add a new paragraph at the beginning of the section "Texture Access", p. 194

Section 2.15.4.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

Modify the second paragraph on p. 195 as follows:

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the R value (see section 3.8.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler and in the texture using the TEXTURE COMPARE MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

* The sampler used in a texture lookup function is not one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is not NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is DEPTH COMPONENT, and the TEXTURE COMPARE MODE is NONE.

* The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not DEPTH COMPONENT.

Add the following paragraph to the section Shader Inputs, p. 196

If a geometry shader is active, the built-in variable gl_PrimitiveID contains the ID value emitted by the geometry shader for the provoking vertex. If no geometry shader is active, gl_PrimitiveID is filled with the number of primitives processed by the rasterizer since the last time Begin was called (directly or indirectly via vertex array functions). The first primitive generated after a Begin is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed. For POLYGON primitives, the primitive ID counter is undefined. The primitive ID is undefined for fragments generated by DrawPixels or Bitmap. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Modify the first paragraph of the section Shader Outputs, p. 196 as follows

The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These outputs are split into two categories. User-defined varying out variables and built-in variables. The built-in variables are gl_FragColor, gl_FragData[n], and gl_FragDepth. If fragment clamping is enabled, the final fragment color values or the final fragment data values or the final varying out variable values written by a fragment shader are clamped to the range [0,1] and then may be converted to fixed-point as described in section 2.14.9. Only user-defined varying out variables declared as a floating-point type are clamped and may be

converted. If fragment clamping is disabled, the final fragment color values or the final fragment data values or the final varying output variable values are not modified. The final fragment depth written...

Modify the second paragraph of the section Shader Outputs, p. 196 as follows

...A fragment shader may not statically assign values to more than one of gl_FragColor, gl_FragData or any user-defined varying output variable. In this case, a compile or link error will result. A shader statically...

Add the following to the end of the section Shader Outputs, p. 197

The values of user-defined varying out variables are directed to a color buffer in a two step process. First the varying out variable is bound to a fragment color by using its number. The GL will assign a number to each varying out variable, unless overridden by the command BindFragDataLocationEXT(). The number of the fragment color assigned for each user-defined varying out variable can be queried with GetFragDataLocationEXT(). Next, the DrawBuffer or DrawBuffers commands (see section 4.2.1) direct each fragment color to a particular buffer.

The binding of a user-defined varying out variable to a fragment color number can be specified explicitly. The command

```
void BindFragDataLocationEXT(uint program, uint colorNumber,
                             const char *name);
```

specifies that the varying out variable name in program should be bound to fragment color colorNumber when the program is next linked. If name was bound previously, its assigned binding is replaced with colorNumber. name must be a null terminated string. The error INVALID_VALUE is generated if colorNumber is equal or greater than MAX_DRAW_BUFFERS. BindFragDataLocationEXT has no effect until the program is linked. In particular, it doesn't modify the bindings of varying out variables in a program that has already been linked. The error INVALID OPERATION is generated if name starts with the reserved "gl_" prefix.

When a program is linked, any varying out variables without a binding specified through BindFragDataLocationEXT will automatically be bound to fragment colors by the GL. Such bindings can be queried using the command GetFragDataLocationEXT. LinkProgram will fail if the assigned binding of a varying out variable would cause the GL to reference a non-existant fragment color number (one greater than or equal to MAX DRAW_BUFFERS). LinkProgram will also fail if more than one varying out variable is bound to the same number. This type of aliasing is not allowed.

BindFragDataLocationEXT may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with "gl_") to a color number, including a name that is never used as a varying out variable in any fragment shader object. Assigned bindings for variables that do not exist are ignored.

After a program object has been linked successfully, the bindings of varying out variable names to color numbers can be queried. The command

```
int GetFragDataLocationEXT(uint program, const char *name);
```

returns the number of the fragment color that the varying out variable name was bound to when the program object program was last linked. name must be a null terminated string. If program has not been successfully linked, the error INVALID OPERATION is generated. If name is not a varying out variable, or if an error occurs, -1 will be returned.

Additions to Chapter 4 of the OpenGL 2.0 Specification (Per-Fragment

Operations and the Frame Buffer)

Modify Section 4.2.1, Selecting a Buffer for Writing (p. 212)

(modify next-to-last paragraph, p. 213) If a fragment shader writes to gl_FragColor, DrawBuffers specifies a set of draw buffers into which the single fragment color defined by gl_FragColor is written. If a fragment shader writes to gl_FragData or a user-defined varying out variable, DrawBuffers specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to neither gl_FragColor, nor gl FragData, nor any user-defined varying out variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

Additions to Chapter 5 of the OpenGL 2.0 Specification (Special Functions)

Change section 5.4 Display Lists, p. 237

Add the commands VertexAttribIPointerEXT and BindFragDataLocationEXT to the list of commands that are not compiled into a display list, but executed immediately, under "Program and Shader Objects", p. 241

Additions to Chapter 6 of the OpenGL 2.0 Specification (State and State Requests)

Modify section 6.1.14 "Shader and Program Queries", p. 256

Modify 2nd paragraph, p.259:

Add the following to the list of GetVertexAttrib* commands:

```
  void GetVertexAttribIivEXT(uint index, enum pname, int *params);
  void GetVertexAttribIuivEXT(uint index, enum pname, uint *params);
```

obtain the... <pname> must be one of VERTEX_ATTRIB_ARRAY_ENABLED ,., VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER_EXT, or CURRENT_VERTEX_ATTRIB. ...

Split 3rd paragraph, p.259

... The size, stride, type, normalized flag, and unconverted integer flag are set by the commands VertexAttribPointer and VertexAttribIPointerEXT. The normalized flag is always set to FALSE by by VertexAttribIPointerEXT. The unconverted integer flag is always set to FALSE by VertexAttribPointer and TRUE by VertexAttribIPointerEXT.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute <index>. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to integer values; GetVertexAttribIivEXT reads and returns them as integers; GetVertexAttribIuivEXT reads and returns them as unsigned integers. The results of the query are undefined if the current attribute values are read using one data type but were specified using a different one. The error INVALID_OPERATION is generated if <index> is zero.

Change the prototypes in the first paragraph on page 260 as follows:

```
  void GetUniformfv(uint program, int location, float *params);
  void GetUniformiv(uint program, int location, int *params);
  void GetUniformuivEXT(uint program, int location, uint *params);
```

Additions to Appendix A of the OpenGL 2.0 Specification (Invariance)

None.

Additions to the AGL/GLX/WGL Specifications

None.

Interactions with GL_ARB_color_buffer_float

If the GL_ARB_color_buffer_float extension is not supported then any reference to fragment clamping in section 3.11.2 "Shader Execution" needs to be deleted.

Interactions with GL_ARB_texture_rectangle

If the GL_ARB_texture_rectangle extension is not supported then all references to texture lookup functions with 'Rect' in the name need to be deleted.

Interactions with GL_EXT_texture_array

If the GL_EXT_texture_array extension is not supported, all references to one- and two-dimensional array texture sampler types (e.g., sampler1DArray, sampler2DArray) and the texture lookup functions that use them need to be deleted.

Interactions with GL_EXT_geometry_shader4

If the GL_EXT_geometry_shader4 extension is not supported, all references to a geometry shader need to be deleted.

Interactions with GL_NV_primitive_restart

The spec describes the behavior that primitive restart does not affect the primitive ID counter, including for POLYGON primitives (where one could argue that the restart index starts a new primitive without a new Begin to reset the count). If NV_primitive_restart is not supported, references to that extension in the discussion of the primitive ID counter should be removed.

If NV_primitive_restart is supported, index values causing a primitive restart are not considered as specifying an End command, followed by another Begin. Primitive restart is therefore not guaranteed to immediately update material properties when a vertex shader is active. The spec language on p.64 of the OpenGL 2.0 specification says "changes are not guaranteed to update material parameters, defined in table 2.11, until the following End command."

Interactions with EXT_texture_integer

If the EXT_texture_integer spec is not supported, the discussion about this spec in section 2.15.4.1 needs to be removed. All texture lookup functions that return integers or unsigned integers, as discussed in section 8.7 of the OpenGL Shading Language specification, also need to be removed.

Interactions with EXT_texture_buffer_object

If EXT_texture_buffer_object is not supported, references to buffer textures, as well as the texelFetchBuffer and texelSizeBuffer lookup functions and samplerBuffer types, need to be removed.

Interactions with EXT_draw_instanced

If EXT_draw_instanced is not supported, the value of gl_InstanceID

is always zero.

GLX Protocol

The following rendering commands are sent to the server as part of a glXRender request:

Uniform1uiEXT

2	12	rendering command length
2	269	rendering command opcode
4	INT32	location
4	CARD32	v0

Uniform2uiEXT

2	16	rendering command length
2	270	rendering command opcode
4	INT32	location
4	CARD32	v0
4	CARD32	v1

Uniform3uiEXT

2	20	rendering command length
2	271	rendering command opcode
4	INT32	location
4	CARD32	v0
4	CARD32	v1
4	CARD32	v2

Uniform4uiEXT

2	24	rendering command length
2	272	rendering command opcode
4	INT32	location
4	CARD32	v0
4	CARD32	v1
4	CARD32	v2
4	CARD32	v3

BindFragDataLocationEXT

2	12+n+p	rendering command length
2	273	rendering command opcode
4	CARD32	program
4	CARD32	color
n	LISTofBYTE	name, n = strlen(name) + 1
p		padding, p=pad(n)

The following rendering commands are sent sent to the server as part of a glXRender request or as a glXRenderLarge request.

Uniform1uivEXT

2	12+count*4	rendering command length
2	274	rendering command opcode
4	INT32	location
4	CARD32	count
4*count	LISTofCARD32	value

If the command is encoded in a glXRenderLarge request, the command opcode and command length fields above are expanded to 4 bytes each:

```
    4           16+count*4      rendering command length
    4           274             rendering command opcode

Uniform2uivEXT

    2           12+count*4*2    rendering command length
    2           275             rendering command opcode
    4           INT32           location
    4           CARD32          count
    2*4*count   LISTofCARD32    value

If the command is encoded in a glXRenderLarge request, the
command opcode and command length fields above are expanded to
4 bytes each:

    4           16+count*4*2    rendering command length
    4           275             rendering command opcode

Uniform3uivEXT

    2           12+count*4*3    rendering command length
    2           276             rendering command opcode
    4           INT32           location
    4           CARD32          count
    3*4*count   LISTofCARD32    value

If the command is encoded in a glXRenderLarge request, the
command opcode and command length fields above are expanded to
4 bytes each:

    4           16+count*4      rendering command length
    4           276             rendering command opcode

Uniform4uivEXT

    2           12+count*4*4    rendering command length
    2           277             rendering command opcode
    4           INT32           location
    4           CARD32          count
    4*4*count   LISTofCARD32    value

If the command is encoded in a glXRenderLarge request, the
command opcode and command length fields above are expanded to
4 bytes each:

    4           16+count*4*4    rendering command length
    4           277             rendering command opcode

The following non-rendering commands are added.

GetUniformuivEXT

    1           CARD8           opcode (X assigned)
    1           182             GLX opcode
    2           4               request length
    4           GLX_CONTEXT_TAG context tag
    4           CARD32          program
    4           INT32           location
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n == 1 ? 0 : n)
    4           CARD32          unused
    4           CARD32          n
```

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

        4           CARD32          params
        12                          unused

        otherwise this follows:

        16          CARD32          unused
        4*n         CARD32          params

    Note that n may be zero, indicating that a GL error occured.

    GetFragDataLocationEXT

        1           CARD8           opcode (X assigned)
        1           183             GLX opcode
        2           3+(n+p)/4       request length
        4           GLX_CONTEXT_TAG context tag
        4           CARD32          program
        n           LISTofBYTE      name, n = strlen(name) + 1
        p                           padding, p=pad(n)
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           0               reply length
        4           CARD32          retval
        20                          unused

    GLX protocol for following commands is defined in the
    NV_vertex_program4 extension:

        VertexAttribI1iEXT, VertexAttribI2iEXT, VertexAttribI3iEXT,
        VertexAttribI4iEXT, VertexAttribI1uiEXT, VertexAttribI2uiEXT,
        VertexAttribI3uiEXT, VertexAttribI4uiEXT, VertexAttribI1ivEXT,
        VertexAttribI2ivEXT, VertexAttribI3ivEXT, VertexAttribI4ivEXT,
        VertexAttribI1uivEXT, VertexAttribI2uivEXT, VertexAttribI3uivEXT,
        VertexAttribI4uivEXT, VertexAttribI4bvEXT, VertexAttribI4svEXT,
        VertexAttribI4ubvEXT, VertexAttribI4usvEXT, GetVertexAttribIivEXT,
        GetVertexAttribIuivEXT

    VertexAttribIPointerEXT is an entirely client-side command.

Errors

    The error INVALID_VALUE is generated by BindFragDataLocationEXT() if
    colorNumber is equal or greater than MAX_DRAW_BUFFERS.

    The error INVALID OPERATION is generated by BindFragDataLocationEXT() if
    name starts with the reserved "gl_" prefix.

    The error INVALID_OPERATION is generated by BindFragDataLocationEXT() or
    GetFragDataLocationEXT if program is not the name of a program object.

    The error INVALID_OPERATION is generated by GetFragDataLocationEXT() if
    program has not been successfully linked.

New State

    (add to table 6.7, p. 268)
                                                Initial
    Get Value            Type    Get Command    Value   Description          Sec.
Attribute
    ---------            ----    -------------- ------- -------------------- ----
```

```
---------
    VERTEX_ATTRIB_ARRAY  16+xB   GetVertexAttrib FALSE   vertex attrib array  2.8
vertex-array
      INTEGER_EXT                                        has unconverted ints

New Implementation Dependent State

                                                                    Minimum
    Get Value                           Type  Get Command           Value
Description           Sec.   Attrib
    --------------------------------  ----  ---------------  -------
--------------------- ------ ------
    MIN_PROGRAM_TEXEL_OFFSET_EXT        Z     GetIntegerv         -8     minimum texel
offset  2.x.4.4  -
                                                                        allowed in lookup
    MAX_PROGRAM_TEXEL_OFFSET_EXT        Z     GetIntegerv         +7     maximum texel
offset  2.x.4.4  -
                                                                        allowed in lookup

Modifications to The OpenGL Shading Language Specification, Version 1.10.59

    Including the following line in a shader can be used to control the
    language features described in this extension:

      #extension GL_EXT_gpu_shader4 : <behavior>

    where <behavior> is as specified in section 3.3.

    A new preprocessor #define is added to the OpenGL Shading Language:

      #define GL_EXT_gpu_shader4 1

    Add to section 3.6 "Keywords"

    Add the following keywords:

      noperspective, flat, centroid

    Remove the unsigned keyword from the list of keywords reserved for future
    use, and add it to the list of keywords.

    The following new vector types are added:

      uvec2, uvec3, uvec4

    The following new sampler types are added:

      sampler1DArray, sampler2DArray, sampler1DArrayShadow,
      sampler2DArrayShadow, samplerCubeShadow

      isampler1D, isampler2D, isampler3D, isamplerCube, isampler2DRect,
      isampler1DArray, isampler2DArray

      usampler1D, usampler2D, usampler3D, usamplerCube, usampler2DRect,
      usampler1DArray, usampler2DArray

      samplerBuffer, isamplerBuffer, usamplerBuffer

    Add to section 4.1 "Basic Types"

    Break the table in this section up in several tables. The first table
    4.1.1 is named "scalar, vector and matrix data types". It includes the
    first row through the 'mat4" row.

    Add the following to the first section of this table:
```

```
  unsigned int           An unsigned integer
  uvec2                  A two-component unsigned integer vector
  uvec3                  A three-component unsigned integer vector
  uvec4                  A four-component unsigned integer vector

Break out the sampler types in a separate table, and name that table 4.1.2
"default sampler types". Add the following sampler types to this new
table:

  sampler1DArray         handle for accessing a 1D array texture
  sampler2DArray         handle for accessing a 2D array texture
  sampler1DArrayShadow   handle for accessing a 1D array depth texture
                         with comparison
  sampler2DArrayShadow   handle for accessing a 2D array depth texture
                         with comparison
  samplerBuffer          handle for accessing a buffer texture

Add a table 4.1.3 called "integer sampler types":

  isampler1D             handle for accessing an integer 1D texture
  isampler2D             handle for accessing an integer 2D texture
  isampler3D             handle for accessing an integer 3D texture
  isamplerCube           handle for accessing an integer cube map texture
  isampler2DRect         handle for accessing an integer rectangle texture
  isampler1DArray        handle for accessing an integer 1D array texture
  isampler2DArray        handle for accessing an integer 2D array texture
  isamplerBuffer         handle for accessing an integer buffer texture

Add a table 4.1.4 called "unsigned integer sampler types":

  usampler1D             handle for accessing an unsigned integer
                         1D texture
  usampler2D             handle for accessing an unsigned integer
                         2D texture
  usampler3D             handle for accessing an unsigned integer
                         3D texture
  usamplerCube           handle for accessing an unsigned integer
                         cube map texture
  usampler2DRect         handle for accessing an unsigned integer
                         rectangle texture
  usampler1DArray        handle for accessing an unsigned integer 1D
                         array texture
  usampler2DArray        handle for accessing an unsigned integer 2D
                         array texture
  usamplerBuffer         handle for accessing an unsigned integer
                         buffer texture

Change section 4.1.3 "Integers"

Remove the first two paragraphs and replace with the following:

Signed, as well as unsigned integers, are fully supported.  Integers hold
whole numbers. Integers have at least 32 bits of precision, including a
sign bit. Signed integers are stored using a two's complement
representation.

Integers are declared and optionally initialized with integer expressions
as in the following example:

    int i, j = 42;
    unsigned int k = 3u;

Literal integer constants can be expressed in decimal (base 10), octal
(base 8), or hexadecimal (base 16) as follows.
```

```
      integer-constant:
            decimal-constant integer-suffix_opt
            octal-constant integer-suffix_opt
            hexadecimal-constant integer-suffix_opt

      integer-suffix:  one of
            u U
```

Change section 4.3 "Type Qualifiers"

Change the "varying" and "out" qualifier as follows:

varying - linkage between a vertex shader and fragment shader, or between a fragment shader and the back end of the OpenGL pipeline.

out - for function parameters passed back out of a function, but not initialized for use when passed in. Also for output varying variables (fragment only).

In the qualifier table, add the following sub-qualifiers under the varying qualifier:

```
  flat varying
  noperspective varying
  centroid varying
```

Change section 4.3.4 "Attribute"

Change the sentence:

The attribute qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4.

To:

The attribute qualifier can be used only with the data types int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3, uvec4, float, vec2, vec3, vec4, mat2, mat3, and mat4.

Change the fourth paragraph to:

It is expected that graphics hardware will have a small number of fixed locations for passing vertex attributes. Therefore, the OpenGL Shading language defines each non-matrix attribute variable as having space for up to four integer or floating-point values (i.e., a vec4, ivec4 or uvec4). There is an implementation dependent limit on the number of attribute variables that can be used and if this is exceeded it will cause a link error. (Declared attribute variables that are not used do not count against this limit.) A scalar attribute counts the same amount against this limit as a vector of size four, so applications may want to consider packing groups of four unrelated scalar attributes together into a vector to better utilize the capabilities of the underlying hardware. A mat4 attribute will...

Change section 4.3.6 "Varying"

Change the first paragraph to:

Varying variables provide the interface between the vertex shader, the fragment shader, and the fixed functionality between the vertex and fragment shader, as well as the interface from the fragment shader to the back-end of the OpenGL pipeline.

The vertex shader will compute values per vertex (such as color, texture

coordinates, etc.) and write them to variables declared with the varying qualifier. A vertex shader may also read varying variables, getting back the same values it has written. Reading a varying variable in a vertex shader returns undefined values if it is read before being written.

The fragment shader will compute values per fragment and write them to variables declared with the varying out qualifier. A fragment shader may also read varying variables, getting back the same result it has written. Reading a varying variable in a fragment shader returns undefined values if it is read before being written.

Varying variables may be written more than once. If so, the last value assigned is the one used.

Change the second paragraph to:

Varying variables that are set per vertex are interpolated by default in a perspective-correct manner over the primitive being rendered, unless the varying is further qualified with noperspective. Interpolation in a perspective correct manner is specified in equations 3.6 and 3.8 in the OpenGL 2.0 specification. When noperspective is specified, interpolation must be linear in screen space, as described in equation 3.7 and the approximation that follows equation 3.8.

If single-sampling, the value is interpolated to the pixel's center, and the centroid qualifier, if present, is ignored. If multi-sampling, and the varying is not qualified with centroid, then the value must be interpolated to the pixel's center, or anywhere within the pixel, or to one of the pixel's samples. If multi-sampling and the varying is qualified with centroid, then the value must be interpolated to a point that lies in both the pixel and in the primitive being rendered, or to one of the pixel's samples that falls within the primitive.

[NOTE: Language for centroid sampling taken from the GLSL 1.20.4 specification]

Varying variables, set per vertex, can be computed on a per-primitive basis (flat shading), or interpolated over a line or polygon primitive (smooth shading). By default, a varying variable is smooth shaded, unless the varying is further qualified with flat. When smooth shading, the varying is interpolated over the primitive. When flat shading, the varying is constant over the primitive, and is taken from the single provoking vertex of the primitive, as described in Section 2.14.7 of the OpenGL 2.0 specification.

Change the fourth paragraph to:

The type and any qualifications (flat, noperspective, centroid) of varying variables with the same name declared in both the vertex and fragment shaders must match, otherwise the link command will fail. Note that built-in varying variables, which have names starting with "gl_", can not be further qualified with flat, noperspective or centroid. The flat keyword cannot be used together with either the noperspective or centroid keywords to further qualify a single varying variable, otherwise a compile error will occur. When using the keywords centroid, flat or noperspective, it must immediately precede the varying keyword. When using both centroid and noperspective keywords, either one can be specified first. Only those varying variables used (i.e. read) in the fragment shader must be written to by the vertex shader; declaring superfluous varying variables in the vertex shader is permissible. Varying out variables, set per fragment, can not be further qualified with flat, noperspective or centroid.

Fragment shaders output values to the back-end of the OpenGL pipeline using either user-defined varying out variables or built-in variables, as described in section 7.2, unless the discard keyword is executed. If the

back-end of the OpenGL pipeline consumes a user-defined varying out variable and an execution of a fragment shader does not write a value to that variable, then the value consumed is undefined. If the back-end of the OpenGL pipeline consumes a varying out variable and a fragment shader either writes values into less components of the variable, or if the variable is declared to have less components, than needed, the values of the missing component(s) are undefined. The OpenGL specification, section 3.x.x, describes API to route varying output variables to color buffers.

Add the following examples:

```
  noperspective varying float temperature;
  flat varying vec3 myColor;
  centroid varying vec2 myTexCoord;
  centroid noperspective varying vec2 myTexCoord;
  varying out ivec3 foo;
```

Change the third paragraph on p. 25 as follows:

The "varying" qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, and mat4, int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3, uvec4 or arrays of these. Structures cannot be varying. If the varying is declared as one of the integer or unsigned integer data type variants, then it has to also be qualified as being flat shaded, otherwise a compile error will occur.

The "varying out" qualifier can be used only with the data types float, vec2, vec3, vec4, int, ivec2, ivec3, ivec4, unsigned int, uvec2, uvec3 or uvec4. Structures or arrays cannot be declared as varying out.

Change section 5.1 "Operators"

Remove the "reserved" qualifications from the following operator precedence table entries:

Precedence	Operator class
3	(tilde is reserved)
4	(modulus reserved)
6	bit-wise shift (reserved)
9	bit-wise and (reserved)
10	bit-wise exclusive or (reserved)
11	bit-wise inclusive or (reserved)
16	(modulus, shift, and bit-wise are reserved)

Change section 5.8 "Assignments"

Change the first bullet from:

   * The arithmetic assignments add into (+=)..

To:

   * The arithmetic assignments add into (+=), subtract from (-=), multiply into (*=), and divide into (/=) as well as the assignments modulus into (%=), left shift by (<<=), right shift by (>>=), and into (&=), inclusive or into (|=), exclusive or into (^=). The expression

Delete the last bullet in this paragraph.

Remove the second bullet in the section starting with: The assignments modulus into..

Change section 5.9 "Expressions"

Change the bullet: The operator modulus (%) is reserved for future
use to:

* The arithmetic operator % that operates on signed or unsigned integer
  typed expressions (including vectors). The two operands must be of the
  same type, or one can be a signed or unsigned integer scalar and the
  other a signed or unsigned integer vector. If the second operand is
  zero, results are undefined. If one operand is scalar and the other is a
  vector, the scalar is applied component-wise to the vector, resulting in
  the same type as the vector. If both operands are non-negative, then the
  remainder is non-negative. Results are undefined if one, or both,
  operands are negative.

Change the last bullet: "Operators and (&), or (|), exclusive or (^), not
(~), right-shift (>>), left shift (<<). These operators are reserved for
future use." To the following bullets:

* The one's complement operator ~. The operand must be of type signed or
  unsigned integer (including vectors), and the result is the one's
  complement of its operand. If the operand is a vector, the operator is
  applied component-wise to the vector. If the operand is unsigned, the
  result is computed by subtracting the value from the largest unsigned
  integer value. If the operand is signed, the result is computed by
  converting the operand to an unsigned integer, applying ~, and
  converting back to a signed integer.

* The shift operators << and >>. For both operators, the operands must be
  of type signed or unsigned integer (including vectors). If the first
  operand is a scalar, the second operand has to be a scalar as well. The
  result is undefined if the right operand is negative, or greater than or
  equal to the number of bits in the left expression's type. The value of
  E1 << E2 is E1 (interpreted as a bit pattern) left-shifted by E2
  bits. The value of E1 >> E2 is E1 right-shifted by E2 bit positions. If
  E1 is a signed integer, the right-shift will extend the sign bit. If E1
  is an unsigned integer, the right-shift will zero-extend.

* The bitwise AND operator &. The operands must be of type signed or
  unsigned integer (including vectors). The two operands must be of the
  same type, or one can be a signed or unsigned integer scalar and the
  other a signed or unsigned integer vector. If one operand is a scalar
  and the other a vector, the scalar is applied component-wise to the
  vector, resulting in the same type as the vector. The result is the
  bitwise AND function of the operands.

* The bitwise exclusive OR operator ^. The operands must be of type signed
  or unsigned integer (including vectors). The two operands must be of the
  same type, or one can be a signed or unsigned integer scalar and the
  other a signed or unsigned integer vector. If one operand is a scalar
  and the other a vector, the scalar is applied component-wise to the
  vector, resulting in the same type as the vector. The result is the
  bitwise exclusive OR function of the operands.

* The bitwise inclusive OR operator |. The operands must be of type signed
  or unsigned integer (including vectors). The two operands must be of the
  same type, or one can be a signed or unsigned integer scalar and the
  other a signed or unsigned integer vector. If one operand is a scalar
  and the other a vector, the scalar is applied component-wise to the
  vector, resulting in the same type as the vector. The result is the
  bitwise inclusive OR function of the operands.

Change Section 7.1 "Vertex Shader Special Variables"

Add the following definition to the list of built-in variable definitions:

```
      int gl_VertexID   // read-only
      int gl_InstanceID // read-only
```

Add the following paragraph at the end of the section:

The variable gl_VertexID is available as a read-only variable from within vertex shaders and holds the integer index <i> implicitly passed to ArrayElement() to specify the vertex. The value of gl_VertexID is defined if and only if:

  * the vertex comes from a vertex array command that specifies a complete primitive (e.g. DrawArrays, DrawElements),

  * all enabled vertex arrays have non-zero buffer object bindings, and

  * the vertex does not come from a display list, even if the display list was compiled using DrawArrays / DrawElements with data sourced from buffer objects.

The variable gl_InstanceID is availale as a read-only variable from within vertex shaders and holds holds the integer index of the current primitive in an instanced draw call (DrawArraysInstancedEXT, DrawElementsInstancedEXT). If the current primitive does not come from an instanced draw call, the value of gl_InstanceID is zero.

Change Section 7.2 "Fragment Shader Special Variables"

Change the 8th and 9th paragraphs on p. 43 as follows:

If a shader statically assigns a value to gl_FragColor, it may not assign a value to any element of gl_FragData nor to any user-defined varying output variable (section 4.3.6). If a shader statically writes a value to any element of gl_FragData, it may not assign a value to gl_FragColor nor to any user-defined varying output variable. That is, a shader may assign values to either gl_FragColor, gl_FragData, or any user-defined varying output variable, but not to a combination of the three options.

If a shader executes the discard keyword, the fragment is discarded, and the values of gl_FragDepth, gl_FragColor, gl_FragData and any user-defined varying output variables become irrelevant.

Add the following paragraph to the top of p. 44:

The variable gl_PrimitiveID is available as a read-only variable from within fragment shaders and holds the id of the currently processed primitive. Section 3.11, subsection "Shader Inputs" of the OpenGL 2.0 specification describes what value it holds based on the primitive type.

Add the following prototype to the list of built-in variables accessible from a fragment shader:

```
     int gl_PrimitiveID;
```

Change Chapter 8, sixth paragraph on page 50:

Change the sentence:

When the built-in functions are specified below, where the input arguments (and corresponding output)can be float, vec2, vec3, or vec4, genType is used as the argument.

To:

When the built-in functions are specified below, where the input arguments (and corresponding output) can be float, vec2, vec3, or vec4, genType is

used as the argument. Where the input arguments (and corresponding output) can be int, ivec2, ivec3 or ivec4, genIType is used as the argument. Where the input arguments (and corresponding output) can be unsigned int, uvec2, uvec3, or uvec4, genUType is used as the argument.

Add to section 8.3 "Common functions"

Add integer versions of the abs, sign, min, max and clamp functions, as follows:

Syntax:

```
  genIType abs(genIType x)

  genIType sign(genIType x)

  genIType min(genIType x, genIType y)
  genIType min(genIType x, int y)
  genUType min(genUType x, genUType y)
  genUType min(genUType x, unsigned int y)

  genIType max(genIType x, genIType y)
  genIType max(genIType x, int y)
  genUType max(genUType x, genUType y)
  genUType max(genUType x, unsigned int y)

  genIType clamp(genIType x, genIType minval, genIType maxval)
  genIType clamp(genIType x, int minval, int maxval)
  genUType clamp(genUType x, genUType minval, genUType maxval)
  genUType clamp(genUType x, unsigned int minval,
                 unsigned int maxval)
```

Add the following new functions:

Syntax:

```
  genType truncate(genType x)
```

Description:

  Returns a value equal to the integer closest to x whose absolute value is not larger than the absolute value of x.

Syntax:

```
  genType round(genType x)
```

Description:

  Returns a value equal to the closest integer to x. If the fractional portion of the operand is 0.5, the nearest even integer is returned. For example, round (1.0) returns 1.0. round(-1.5) returns -2.0. round(3.5) and round (4.5) both return 4.0.

Add to section 8.6 "Vector Relational Functions"

Change the sentence:

Below, "bvec" is a placeholder for one of bvec2, bvec3, or bvec4, "ivec" is a placeholder for one of ivec2, ivec3, or ivec4, and "vec" is a placeholder for vec2, vec3, or vec4.

To:

Below, "bvec" is a placeholder for one of bvec2, bvec3, or bvec4, "ivec"

is a placeholder for one of ivec2, ivec3, or ivec4, "uvec" is a placeholder for one of uvec2, uvec3 or uvec4 and "vec" is a placeholder for vec2, vec3, or vec4.

Add uvec versions of all but the any, all and not functions to the table in this section, as follows:

```
   bvec lessThan(uvec x, uvec y)
   bvec lessThanEqual(uvec x, uvec y)

   bvec greaterThan(uvec x, uvec y)
   bvec greaterThanEqual(uvec x, uvec y)

   bvec equal(uvec x, uvec y)
   bvec notEqual(uvec x, uvec y)
```

Add to section 8.7 "Texture Lookup Functions"

Remove the first sentence in the last paragraph:

"The built-ins suffixed with "Lod" are allowed only in a vertex shader.".

Add to this section:

Texture data can be stored by the GL as floating point, unsigned normalized integer, unsigned integer or signed integer data. This is determined by the type of the internal format of the texture. Texture lookups on unsigned normalized integer and floating point data return floating point values in the range [0, 1]. See also section 2.15.4.1 of the OpenGL specification.

Texture lookup functions are provided that can return their result as floating point, unsigned integer or signed integer, depending on the sampler type passed to the lookup function. Care must be taken to use the right sampler type for texture access. Table 8.xxx lists the supported combinations of sampler types and texture internal formats.

texture internal format	default (float) sampler	integer sampler	unsigned integer sampler
float	vec4	n/a	n/a
normalized	vec4	n/a	n/a
signed int	n/a	ivec4	n/a
unsigned int	n/a	n/a	uvec4

Table 8.xxx Valid combinations of the type of the internal format of a texture and the type of the sampler used to access the texture. Each cell in the table indicates the type of the return value of a texture lookup. N/a means this combination is not supported. A texture lookup using a n/a combination will return undefined values. The exceptions to this table are the "textureSize" lookup functions, which will return an integer or integer vector, regardless of the sampler type.

If a texture with a signed integer internal format is accessed, one of the signed integer sampler types must be used. If a texture with an unsigned integer internal format is accessed, one of the unsigned integer sampler types must be used. Otherwise, one of the default (float) sampler types must be used. If the types of a sampler and the corresponding texture internal format do not match, the result of a texture lookup is undefined.

If an integer sampler type is used, the result of a texture lookup is an ivec4. If an unsigned integer sampler type is used, the result of a texture lookup is a uvec4. If a default sampler type is used, the result of a texture lookup is a vec4, where each component is in the range [0, 1].

Integer and unsigned integer functions of all the texture lookup functions described in this section are also provided, except for the "shadow" versions, using function overloading. Their prototypes, however, are not listed separately. These overloaded functions use the integer or unsigned-integer versions of the sampler types and will return an ivec4 or an uvec4 respectively, except for the "textureSize" functions, which will always return an integer, or integer vector. Refer also to table 8.xxxx for valid combinations of texture internal formats and sampler types. For example, for the texture1D function, the complete set of prototypes is:

```
   vec4 texture1D(sampler1D sampler, float coord
                  [, float bias])
   ivec4 texture1D(isampler1D sampler, float coord
                   [, float bias])
   uvec4 texture1D(usampler1D sampler, float coord
                   [, float bias])
```

Add the following new texture lookup functions:

Syntax:

```
  vec4 texelFetch1D(sampler1D sampler, int coord, int lod)
  vec4 texelFetch2D(sampler2D sampler, ivec2 coord, int lod)
  vec4 texelFetch3D(sampler3D sampler, ivec3 coord, int lod)
  vec4 texelFetch2DRect(sampler2DRect sampler, ivec2 coord)
  vec4 texelFetch1DArray(sampler1DArray sampler, ivec2 coord, int lod)
  vec4 texelFetch2DArray(sampler2DArray sampler, ivec3 coord, int lod)
```

Description:

Use integer texture coordinate <coord> to lookup a single texel from the level-of-detail <lod> on the texture bound to <sampler> as described in section 2.15.4.1 of the OpenGL specification "Texel Fetches". For the "array" versions, the layer of the texture array to access is either coord.t or coord.p, depending on the use of the 1D or 2D texel fetch lookup, respectively. Note that texelFetch2DRect does not take a level-of-detail input.

Syntax:

```
  vec4 texelFetchBuffer(samplerBuffer sampler, int coord)
```

Description:

Use integer texture coordinate <coord> to lookup into the buffer texture bound to <sampler>.

Syntax:

```
  int textureSizeBuffer(samplerBuffer sampler)
  int textureSize1D(sampler1D sampler, int lod)
  ivec2 textureSize2D(sampler2D sampler, int lod)
  ivec3 textureSize3D(sampler3D sampler, int lod)
  ivec2 textureSizeCube(samplerCube sampler, int lod)
  ivec2 textureSize2DRect(sampler2DRect sampler, int lod)
  ivec2 textureSize1DArray(sampler1DArray sampler, int lod)
  ivec3 textureSize2DArray(sampler2DArray sampler, int lod)
```

Description:

Returns the dimensions, width, height, depth, and number of layers, of level <lod> for the texture bound to <sampler>, as described in section 2.15.4.1 of the OpenGL specification section "Texture Size Query". For the textureSize1DArray function, the first (".x") component of the returned

vector is filled with the width of the texture image and the second component with the number of layers in the texture array. For the textureSize2DArray function, the first two components (".x" and ".y") of the returned vector are filled with the width and height of the texture image respectively. The third component (".z") is filled with the number of layers in the texture array.

Syntax:

```
  vec4 texture1DArray(sampler1DArray sampler, vec2 coord
                      [, float bias])
  vec4 texture1DArrayLod(sampler1DArray sampler, vec2 coord,
                         float lod)
```

Description:

Use the first element (coord.s) of texture coordinate coord to do a texture lookup in the layer indicated by the second coordinate coord.t of the 1D texture array currently bound to sampler. The layer to access is computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the texture array.

Syntax:

```
  vec4 texture2DArray(sampler2DArray sampler, vec3 coord
                      [, float bias])
  vec4 texture2DArrayLod(sampler2DArray sampler, vec3 coord,
                         float lod)
```

Description:

Use the first two elements (coord.s, coord.t) of texture coordinate coord to do a texture lookup in the layer indicated by the third coordinate coord.p of the 2D texture array currently bound to sampler. The layer to access is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth of the texture array.

Syntax:

```
  vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                     [float bias])
  vec4 shadow1DArrayLod(sampler1DArrayShadow sampler,
                        vec3 coord, float lod)
```

Description:

Use texture coordinate coord.s to do a depth comparison lookup on an array layer of the depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The layer to access is indicated by the second coordinate coord.t and is computed by layer = max (0, min(d - 1, floor (coord.t + 0.5)) where 'd' is the depth of the texture array. The third component of coord (coord.p) is used as the R value. The texture bound to sampler must be a depth texture, or results are undefined.

Syntax:

```
  vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
```

Description:

Use texture coordinate (coord.s, coord.t) to do a depth comparison lookup on an array layer of the depth texture bound to sampler, as described in section 3.8.14 of version 2.0 of the OpenGL specification. The layer to access is indicated by the third coordinate coord.p and is computed by layer = max (0, min(d - 1, floor (coord.p + 0.5)) where 'd' is the depth of the texture array. The fourth component of coord (coord.q) is used as

the R value. The texture bound to sampler must be a depth texture, or results are undefined.

Syntax:

```
    vec4 shadowCube(samplerCubeShadow sampler, vec4 coord)
```

Description:

Use texture coordinate (coord.s, coord.t, coord.p) to do a depth comparison lookup on the depth cubemap bound to sampler, as described in section 3.8.14. The direction of the vector (coord.s, coord.t, coord.p) is used to select which face to do a two-dimensional texture lookup in, as described in section 3.8.6 of the OpenGL 2.0 specification. The fourth component of coord (coord.q) is used as the R value. The texture bound to sampler must be a depth cubemap, otherwise results are undefined.

Syntax:

```
  vec4 texture1DGrad(sampler1D sampler, float coord,
                     float ddx, float ddy);
  vec4 texture1DProjGrad(sampler1D sampler, vec2 coord,
                         float ddx, float ddy);
  vec4 texture1DProjGrad(sampler1D sampler, vec4 coord,
                         float ddx, float ddy);
  vec4 texture1DArrayGrad(sampler1DArray sampler, vec2 coord,
                          float ddx, float ddy);

  vec4 texture2DGrad(sampler2D sampler, vec2 coord,
                     vec2 ddx, vec2 ddy);
  vec4 texture2DProjGrad(sampler2D sampler, vec3 coord,
                         vec2 ddx, vec2 ddy);
  vec4 texture2DProjGrad(sampler2D sampler, vec4 coord,
                         vec2 ddx, vec2 ddy);
  vec4 texture2DArrayGrad(sampler2DArray sampler, vec3 coord,
                          vec2 ddx, vec2 ddy);

  vec4 texture3DGrad(sampler3D sampler, vec3 coord,
                     vec3 ddx, vec3 ddy);
  vec4 texture3DProjGrad(sampler3D sampler, vec4 coord,
                         vec3 ddx, vec3 ddy);

  vec4 textureCubeGrad(samplerCube sampler, vec3 coord,
                       vec3 ddx, vec3 ddy);

  vec4 shadow1DGrad(sampler1DShadow sampler, vec3 coord,
                    float ddx, float ddy);
  vec4 shadow1DProjGrad(sampler1DShadow sampler, vec4 coord,
                        float ddx, float ddy);
  vec4 shadow1DArrayGrad(sampler1DArrayShadow sampler, vec3 coord,
                         float ddx, float ddy);

  vec4 shadow2DGrad(sampler2DShadow sampler, vec3 coord,
                    vec2 ddx, vec2 ddy);
  vec4 shadow2DProjGrad(sampler2DShadow sampler, vec4 coord,
                        vec2 ddx, vec2 ddy);
  vec4 shadow2DArrayGrad(sampler2DArrayShadow sampler, vec4 coord,
                         vec2 ddx, vec2 ddy);

  vec4 texture2DRectGrad(sampler2DRect sampler, vec2 coord,
                         vec2 ddx, vec2 ddy);
  vec4 texture2DRectProjGrad(sampler2DRect sampler, vec3 coord,
                             vec2 ddx, vec2 ddy);
  vec4 texture2DRectProjGrad(sampler2DRect sampler, vec4 coord,
                             vec2 ddx, vec2 ddy);
```

```
  vec4 shadow2DRectGrad(sampler2DRectShadow sampler, vec3 coord,
                        vec2 ddx, vec2 ddy);
  vec4 shadow2DRectProjGrad(sampler2DRectShadow sampler, vec4 coord,
                            vec2 ddx, vec2 ddy);

  vec4 shadowCubeGrad(samplerCubeShadow sampler, vec4 coord,
                      vec3 ddx, vec3 ddy);
```

Description:

The "Grad" functions map the partial derivatives ddx and ddy to ds/dx, dt/dx, dr/dx, and ds/dy, dt/dy, dr/dy respectively and use texture coordinate "coord" to do a texture lookup as described for their non "Grad" counterparts. The derivatives ddx and ddy are used as the explicit derivate of "coord" with respect to window x and window y respectively and are used to compute lambda_base(x,y) as in equation 3.18 in the OpenGL 2.0 specification. For the "Proj" versions, it is assumed that the partial derivatives ddx and ddy are already projected. I.e. the GL assumes that ddx and ddy represent d(s/q)/dx, d(t/q)/dx, d(r/q)/dx and d(s/q)/dy, d(t/q)/dy, d(r/q)/dy respectively. For the "Cube" versions, the partial derivatives ddx and ddy are assumed to be in the coordinate system used before texture coordinates are projected onto the appropriate cube face. The partial derivatives of the post-projection texture coordinates, which are used for level-of-detail and anisotropic filtering calculations, are derived from coord, ddx and ddy in an implementation-dependent manner.

NOTE: Except for the "array" and shadowCubeGrad() functions, these functions are taken from the ARB_shader_texture_lod spec and are functionally equivalent.

Syntax:

```
  vec4 texture1DOffset(sampler1D sampler, float coord,
                       int offset [,float bias])
  vec4 texture1DProjOffset(sampler1D sampler, vec2 coord,
                           int offset [,float bias])
  vec4 texture1DProjOffset(sampler1D sampler, vec4 coord,
                           int offset [,float bias])
  vec4 texture1DLodOffset(sampler1D sampler, float coord,
                          float lod, int offset)
  vec4 texture1DProjLodOffset(sampler1D sampler, vec2 coord,
                              float lod, int offset)
  vec4 texture1DProjLodOffset(sampler1D sampler, vec4 coord,
                              float lod, int offset)

  vec4 texture2DOffset(sampler2D sampler, vec2 coord,
                       ivec2 offset [,float bias])
  vec4 texture2DProjOffset(sampler2D sampler, vec3 coord,
                           ivec2 offset [,float bias])
  vec4 texture2DProjOffset(sampler2D sampler, vec4 coord,
                           ivec2 offset [,float bias])
  vec4 texture2DLodOffset(sampler2D sampler, vec2 coord,
                          float lod, ivec2 offset)
  vec4 texture2DProjLodOffset(sampler2D sampler, vec3 coord,
                              float lod, ivec2 offset)
  vec4 texture2DProjLodOffset(sampler2D sampler, vec4 coord,
                              float lod, ivec2 offset)

  vec4 texture3DOffset(sampler3D sampler, vec3 coord,
                       ivec3 offset [,float bias])
  vec4 texture3DProjOffset(sampler3D sampler, vec4 coord,
                           ivec3 offset [,float bias])
  vec4 texture3DLodOffset(sampler3D sampler, vec3 coord,
```

```
                          float lod, ivec3 offset)
vec4 texture3DProjLodOffset(sampler3D sampler, vec4 coord,
                            float lod, ivec3 offset)

vec4 shadow1DOffset(sampler1DShadow sampler, vec3 coord,
                    int offset [,float bias])
vec4 shadow2DOffset(sampler2DShadow sampler, vec3 coord,
                    ivec2 offset [,float bias])
vec4 shadow1DProjOffset(sampler1DShadow sampler, vec4 coord,
                        int offset [,float bias])
vec4 shadow2DProjOffset(sampler2DShadow sampler, vec4 coord,
                        ivec2 offset [,float bias])
vec4 shadow1DLodOffset(sampler1DShadow sampler, vec3 coord,
                       float lod, int offset)
vec4 shadow2DLodOffset(sampler2DShadow sampler, vec3 coord,
                       float lod, ivec2 offset)
vec4 shadow1DProjLodOffset(sampler1DShadow sampler, vec4 coord,
                           float lod, int offset)
vec4 shadow2DProjLodOffset(sampler2DShadow sampler, vec4 coord,
                           float lod, ivec2 offset)

vec4 texture2DRectOffset(sampler2DRect sampler, vec2 coord,
                         ivec2 offset)
vec4 texture2DRectProjOffset(sampler2DRect sampler, vec3 coord,
                             ivec2 offset)
vec4 texture2DRectProjOffset(sampler2DRect sampler, vec4 coord,
                             ivec2 offset)
vec4 shadow2DRectOffset(sampler2DRectShadow sampler, vec3 coord,
                        ivec2 offset)
vec4 shadow2DRectProjOffset(sampler2DRectShadow sampler, vec4 coord,
                            ivec2 offset)

vec4 texelFetch1DOffset(sampler1D sampler, int coord, int lod,
                        int offset)
vec4 texelFetch2DOffset(sampler2D sampler, ivec2 coord, int lod,
                        ivec2 offset)
vec4 texelFetch3DOffset(sampler3D sampler, ivec3 coord, int lod,
                        ivec3 offset)
vec4 texelFetch2DRectOffset(sampler2DRect sampler, ivec2 coord,
                            ivec2 offset)
vec4 texelFetch1DArrayOffset(sampler1DArray sampler, ivec2 coord,
                             int lod, int offset)
vec4 texelFetch2DArrayOffset(sampler2DArray sampler, ivec3 coord,
                             int lod, ivec2 offset)

vec4 texture1DArrayOffset(sampler1DArray sampler, vec2 coord,
                          int offset [, float bias])
vec4 texture1DArrayLodOffset(sampler1DArray sampler, vec2 coord,
                             float lod, int offset)

vec4 texture2DArrayOffset(sampler2DArray sampler, vec3 coord,
                          ivec2 offset [, float bias])
vec4 texture2DArrayLodOffset(sampler2DArray sampler, vec3 coord,
                             float lod, ivec2 offset)

vec4 shadow1DArrayOffset(sampler1DArrayShadow sampler, vec3 coord,
                         int offset, [float bias])
vec4 shadow1DArrayLodOffset(sampler1DArrayShadow sampler, vec3 coord,
                            float lod, int offset)

vec4 shadow2DArrayOffset(sampler2DArrayShadow sampler,
                         vec4 coord, ivec2 offset)

vec4 texture1DGradOffset(sampler1D sampler, float coord,
                         float ddx, float ddy, int offset);
```

```
  vec4 texture1DProjGradOffset(sampler1D sampler, vec2 coord,
                               float ddx, float ddy, int offset);
  vec4 texture1DProjGradOffset(sampler1D sampler, vec4 coord,
                               float ddx, float ddy, int offset);
  vec4 texture1DArrayGradOffset(sampler1DArray sampler, vec2 coord,
                                float ddx, float ddy, int offset);

  vec4 texture2DGradOffset(sampler2D sampler, vec2 coord,
                           vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DProjGradOffset(sampler2D sampler, vec3 coord,
                               vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DProjGradOffset(sampler2D sampler, vec4 coord,
                               vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DArrayGradOffset(sampler2DArray sampler, vec3 coord,
                                vec2 ddx, vec2 ddy, ivec2 offset);

  vec4 texture3DGradOffset(sampler3D sampler, vec3 coord,
                           vec3 ddx, vec3 ddy, ivec3 offset);
  vec4 texture3DProjGradOffset(sampler3D sampler, vec4 coord,
                               vec3 ddx, vec3 ddy, ivec3 offset);

  vec4 shadow1DGradOffset(sampler1DShadow sampler, vec3 coord,
                          float ddx, float ddy, int offset);
  vec4 shadow1DProjGradOffset(sampler1DShadow sampler,
                              vec4 coord, float ddx, float ddy,
                              int offset);
  vec4 shadow1DArrayGradOffset(sampler1DArrayShadow sampler,
                               vec3 coord, float ddx, float ddy,
                               int offset);

  vec4 shadow2DGradOffset(sampler2DShadow sampler, vec3 coord,
                          vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 shadow2DProjGradOffset(sampler2DShadow sampler, vec4 coord,
                              vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 shadow2DArrayGradOffset(sampler2DArrayShadow sampler,
                               vec4 coord, vec2 ddx, vec2 ddy,
                               ivec2 offset);

  vec4 texture2DRectGradOffset(sampler2DRect sampler, vec2 coord,
                               vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DRectProjGradOffset(sampler2DRect sampler, vec3 coord,
                                   vec2 ddx, vec2 ddy, ivec2 offset);
  vec4 texture2DRectProjGradOffset(sampler2DRect sampler, vec4 coord,
                                   vec2 ddx, vec2 ddy, ivec2 offset);

  vec4 shadow2DRectGradOffset(sampler2DRectShadow sampler,
                              vec3 coord, vec2 ddx, vec2 ddy,
                              ivec2 offset);
  vec4 shadow2DRectProjGradOffset(sampler2DRectShadow sampler,
                                  vec4 coord, vec2 ddx, vec2 ddy,
                                  ivec2 offset);
```

Description:

The "offset" version of each function provides an extra parameter <offset> which is added to the (u,v,w) texel coordinates before looking up each texel. The offset value must be a constant expression. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by MIN_PROGRAM_TEXEL_OFFSET_EXT and MAX_PROGRAM_TEXEL_OFFSET_EXT, respectively. Note that <offset> does not apply to the layer coordinate for texture arrays. This is explained in detail in section 3.8.7 of the OpenGL Specification. Note that texel offsets are also not supported for cubemaps or buffer textures.

Add to section 9 "Grammar"

```
    type_qualifer:
        CONST
        ATTRIBUTE  // Vertex only
        varying-modifier_opt VARYING
        UNIFORM

    varying-modifier:
        FLAT
        CENTROID
        NOPERSPECTIVE

    type_specifier:
        VOID
        FLOAT
        INT
        UNSIGNED_INT
        BOOL
```

Issues

1. Should we support shorts in GLSL?

   DISCUSSION:

   RESOLUTION: UNRESOLVED

2. Do bitwise shifts, AND, exclusive OR and inclusive OR support all combinations of scalars and vectors for each operand?

   DISCUSSION: It seems sense to support scalar OP scalar, vector OP scalar and vector OP vector. But what about scalar OP vector? Should the scalar be promoted to a vector first?

   RESOLUTION: RESOLVED. Yes, this should work essentially as the '+' operator. The scalar is applied to each component of the vector.

3. Which built-in functions should also operate on integers?

   DISCUSSION: There are several that don't make sense to define to operate on integers at all, but the following can be debated: pow, sqrt, dot (and the functions that use dot), cross.

   RESOLUTION: RESOLVED. Integer versions of the abs, sign, min, max and clamp functions are defined. Note that the modulus operator % has been defined for integer operands.

4. Do we need to support integer matrices?

   DISCUSSION:

   RESOLUTION: RESOLVED No, not at the moment.

5. Which texture array lookup functions do we need to support?

   DISCUSSION: We don't want to support lookup functions that need more than four components passed as parameters. Components can be used for texture coordinates, layer selection, 'R' depth compare and the 'q' coordinate for projection. However, texture projection might be relatively easy to support through code-generation, thus we might be able to support functions that need five components, as long as one of them is 'q' for projective texturing. Specifically, should we support:

```
    vec4 texture2DArrayProjLod(sampler2DArray sampler, vec4 coord,
                               float lod)
```

```
   vec4 shadow1DArray(sampler1DArrayShadow sampler, vec3 coord,
                      [float bias])
   vec4 shadow1DArrayProj(sampler1DArrayShadow sampler, vec4 coord,
                          [float bias])
   vec4 shadow1DArrayLod(sampler1DArrayShadow sampler, vec3 coord,
                         float lod)
   vec4 shadow1DArrayProjLod(sampler1DArrayShadow sampler,
                             vec4 coord, float lod)
   vec4 shadow2DArray(sampler2DArrayShadow sampler, vec4 coord)
   vec4 shadow2DArrayProj(sampler2DArrayShadow sampler, vec4 coord,
                          float refValue)
```

  RESOLUTION: RESOLVED, We'll support all but the "Proj" versions. The
  assembly spec (NV_gpu_program4) doesn't support the equivalent
  functionality, either.

6. How do we handle conversions between integer and unsigned
integers?

  DISCUSSION: Do we allow automatic type conversions between signed and
  unsigned integers?

  RESOLUTION: RESOLVED. We will not add this until GLSL version 1.20 has
  been defined, and the implicit conversion rules have been established
  there. If we do this, we would likely only support implicit conversion
  from int to unsigned int, just like C does.

7. Should varying modifiers (flat, noperspective) apply to built-in
   varying variables also?

  DISCUSSION: There is API to control flat vs smooth shading for colors
  through glShadeModel(). There is also API to hint if colors should be
  interpolated perspective correct, or not, through glHint(). These API
  commands apply to the built-in color varying variables (gl_FrontColor
  etc). If the varying modifiers in a shader also apply to the color
  built-ins, which has precedence?

  RESOLUTION: RESOLVED. It is simplest and cleanest to only allow the
  varying modifiers to apply to user-defined varying variables. The
  behavior of the built-in color varying variables can still be controlled
  through the API.

8. How should perspective-incorrect interpolation (linear in screen space)
   and clipping interact?

  RESOLVED: Primitives with attributes specified to be perspective-
  incorrect should be clipped so that the vertices introduced by clipping
  should have attribute values consistent with the interpolation mode. We
  do not want to have large color shifts introduced by clipping a
  perspective-incorrect attribute. For example, a primitive that
  approaches, but doesn't cross, a frustum clip plane should look pretty
  much identical to a similar primitive that just barely crosses the clip
  plane.

  Clipping perspective-incorrect interpolants that cross the W==0 plane is
  very challenging. The attribute clipping equation provided in the spec
  effectively projects all the original vertices to screen space while
  ignoring the X and Y frustum clip plane. As W approaches zero, the
  projected X/Y window coordinates become extremely large. When clipping
  an edge with one vertex inside the frustum and the other out near
  infinity (after projection, due to W approaching zero), the interpolated
  attribute for the entire visible portion of the edge should almost
  exactly match the attribute value of the visible vertex.

  If an outlying vertex approaches and then goes past W==0, it can be said

to go "to infinity and beyond" in screen space. The correct answer for screen-linear interpolation is no longer obvious, at least to the author of this specification. Rather than trying to figure out what the "right" answer is or if one even exists, the results of clipping such edges is specified as undefined.

9. Do we need to support a non-MRT fragment shader writing to (unsigned) integer outputs?

DISCUSSION: Fragment shaders with only one fragment output are considered non-MRT shaders. This means that the output of the shader gets smeared across all color buffers attached to the framebuffer. Fragment shaders with multiple fragment outputs are MRT shaders. Each output is directed to a color buffer using the DrawBuffers API (for gl_FragData) and a combination of the BindFragDataLocationEXT and DrawBuffers API (for varying out variables). Before this extension, a non-MRT shader would write to gl_Color only. A shader writing to gl_FragData[] is a MRT shader. With the addition of varying out variables in this extension, any shader writing to a variable out variable is a MRT shader. It is not possible to construct a non-MRT shader writing to varying out variables. Varying out variables can be declared to be of type integer or unsigned integer. In order to support a non-MRT shader that can write to (unsigned) integer outputs, we could define two new built-in variables:

```
ivec4 gl_FragColorInt;
uvec4 gl_FragColorUInt;
```

Or we could add a special rule stating that if the program object writes to exactly one varying out variable, it is considered to be non-MRT.

RESOLUTION: NO. We don't care enough to support this.

10. Is section 2.14.8, "Color and Associated Data Clipping" in the core specification still correct?

DISCUSSION: This section is in need of some updating, now that varying variables can be interpolated without perspective correction. Some (not so precise) language has been added in the spec body, suggesting that the interpolation needs to be performed in such a way as to produce results that vary linearly in screen space. However, we could define the exact interpolation method required to achieve this. A suggested updated paragraph follows, but we'll leave updating section 2.14.8 to a future edit of the core specification, not this extension.

Replace Section 2.14.8, and rename it to "Vertex Attribute Clipping"

After lighting, clamping or masking and possible flatshading, vertex attributes, including colors, texture and fog coordinates, shader varying variables, and point sizes computed on a per vertex basis, are clipped. Those attributes associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the attributes assigned to vertices produced by clipping are produced by interpolating attributes along the clipped edge.

Let the attributes assigned to the two vertices P_1 and P_2 of an unclipped edge be a_1 and a_2. The value of t (section 2.12) for a clipped point P is used to obtain the attribute associated with P as

$$a = t * a_1 + (1-t) * a_2$$

unless the attribute is specified to be interpolated without perspective correction in a shader (using the noperspective keyword). In that case, the attribute associated with P is

```
    a = t' * a_1 + (1-t') * a_2
```

where

```
    t' = (t * w_1) / (t * w_1 + (1-t) * w_2)
```

and w_1 and w_2 are the w clip coordinates of P_1 and P_2, respectively. If w_1 or w_2 is either zero or negative, the value of the associated attribute is undefined.

For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For a vector attribute, it means multiplying each vector component by the scalar. Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Attribute clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

11. When and where in the texture filtering process are texel offsets applied?

    DISCUSSION: Texel offsets are applied to the (u,v,w) coordinates of the base level of the texture if the texture filter mode does not indicate mipmapping. Otherwise, texel offsets are applied to the (u,v,w) coordinates of the mipmap level 'd', as found by equation 3.27 or to mipmap levels 'd1' and 'd2' as found by equation 3.28 in the OpenGL 2.0 specification. In other words, texel offsets are applied to the (u,v,w) coordinate of whatever mipmap level is accessed.

12. Why is writing to the built-in output variable "gl_Position" in a vertex shader now optional?

    DISCUSSION: Before this specification, writing to gl_Position in a vertex shader was mandatory. The GL pipeline required a vertex position to be written in order to produce well-defined output. This is still the case if the GL pipeline indeed needs a vertex position. However, with fourth-generation programmable hardware there are now cases where the GL pipeline no longer requires a vertex position in order to produce well-defined results. If a geometry shader is present, the vertex shader does not need to write to gl_Position anymore. Instead, the geometry shader can compute a vertex position and write to its gl_Position output. In case of transform-feedback, where the output of a vertex or geometry shader is streamed to one or more buffer objects, perfectly valid results can be obtained without either the vertex shader nor geometry shader writing to gl_Position. The transform-feedback specification adds a new enable to discard primitives right before rasterization, making it potentially unnecessary to write to gl_Position.

13. How does this extension interact with ARB_shader_texture_lod?

    DISCUSSION: This extension adds "Grad" functions which are functionally equivalent to those defined by ARB_shader_texture_lod, but do not have an ARB suffix.

    RESOLUTION: There is no interaction. If both extensions are supported, both sets of functions are available and can be controlled independently via the #extension mechanism.

Revision History

Rev.	Date	Author	Changes

16	12/14/09	mgodse	Added GLX protocol.
15	04/09/09	pbrown	Fixed a typo in the texture size query spec language - returns a layer count, not "index".
14	07/29/08	pbrown	Discovered additional issues with texture wrap handling, replaced with logic that applies wrap modes per sample.
13	04/04/08	pbrown	Added issue 13, concerning the (non-)interaction with ARB_shader_texture_lod.
12	02/04/08	pbrown	Fix errors in texture wrap mode handling. Added a missing clamp to avoid sampling border in REPEAT mode. Fixed incorrectly specified weights for LINEAR filtering.
11	05/08/07	pbrown	Add VertexAttribIPointerEXT to the list of commands that can't go in display lists.
10	01/23/07	pbrown	Fix prototypes for a variety of functions that were specified with an incorrect sampler type.
9	12/15/06	pbrown	Documented that the '#extension' token for this extension should begin with "GL_", as apparently called for per convention.
8	--		Pre-release revisions.