NVIDIA OpenGL
Extension Specifications

NVIDIA Corporation

July 31, 2000

Other OpenGL extension specifications can be found at:

http://oss.sgi.com/projects/ogl-sample/registry/

Table of Contents

Table of NVIDIA OpenGL Extension Support

Extension	RIVA 128 family	RIVA TNT family	GeForce family	OpenGL 1.2 functionality
ARB_multitexture		X	X	
ARB_texture_compression			X	
ARB_transpose_matrix		X	X	
EXT_abgr	X	X	X	
EXT_bgra	X	X	X	Y
EXT_blend_color			X	ARB_imaging
EXT_blend_minmax			X	ARB_imaging
EXT_blend_subtract			X	ARB_imaging
EXT_compiled_vertex_array		X	X	
EXT_filter_anisotropic			X	
EXT_fog_coord		X	X	
EXT_light_max_exponent		X	X	
EXT_packed_pixels	X	X	X	Y
EXT_paletted_texture			X	
EXT_pointer_parameters	X	X	X	
EXT_rescale_normal			X	Y
EXT_secondary_color		X	X	
EXT_separate_specular_color		X	X	Y
EXT_shared_texture_palette			X	
EXT_stencil_wrap	X	X	X	
EXT_texture_compression_s3tc			X	
EXT_texture_cube_map			X	
EXT_texture_edge_clamp		X	X	Y
EXT_texture_env_add		X	X	
EXT_texture_env_combine		X	X	
EXT_texture_lod_bias			X	
EXT_texture_object	X	X	X	
EXT_vertex_array	X	X	X	
EXT_vertex_weighting		X	X	
KTX_buffer_region	X	X	X	
NV_blend_square		X	X	
NV_fence			X	
NV_fog_distance		X	X	
NV_register_combiners			X	
NV_texgen_emboss			X	
NV_texgen_reflection	X	X	X	
NV_texture_env_combine4		X	X	
NV_vertex_array_range			X	
SGIS_multitexture		X	X	
SGIS_texture_lod			X	Y
WGL_EXT_swap_control		X	X	
WIN_swap_hint	X	X	X	

Warning: The extension support columns are based on the latest & greatest NVIDIA driver release. Check your GL_EXTENSIONS string with glGetString at run-time to determine the specific supported extensions for a particular driver version.

Name

ARB_multitexture

Name Strings

GL_ARB_multitexture

Status

Complete. Approved by ARB on 9/15/1998

NOTE: This extension no longer has its own specification document, since it has been included in the OpenGL 1.2.1 Specification (downloadable from www.opengl.org). Please refer to the 1.2.1 Specification for more information.

Name

ARB_texture_compression

Name Strings

GL_ARB_texture_compression

Contact

Pat Brown, Intel Corporation (patrick.r.brown ’at’ intel.com)

Status

FINAL VERSION -- APPROVED BY OPENGL ARB, 3/16/2000.

Version

```
Final 1.03, 23 May 2000 (supersedes Final 1.0, 24 March 2000 -
                         contains a few minor fixes documented in
                         the Revision History below).
```

Number

ARB Extension #12

Dependencies

OpenGL 1.1 is required.

This extension is written against the OpenGL 1.2.1 Specification.

This extension is written against the GLX Extensions for OpenGL Specification (Version 1.3).

Depends on GL_ARB_texture_cube_map, as cube maps may be stored in compressed form.

Overview

Compressing texture images can reduce texture memory utilization and improve performance when rendering textured primitives. This extension allows OpenGL applications to use compressed texture images by providing:

(1) A framework upon which extensions providing specific compressed image formats can be built.

(2) A set of generic compressed internal formats that allow applications to specify that texture images should be stored in compressed form without needing to code for specific compression formats.

An application can define compressed texture images by providing a texture image stored in a specific compressed image format. This extension does not define any specific compressed image formats, but it does provide the mechanisms necessary to enable other extensions that do.

An application can also define compressed texture images by providing an uncompressed texture image but specifying a compressed internal format. In this case, the GL will automatically compress the texture image using the appropriate image format. Compressed internal formats can either be specific (as above) or generic. Generic compressed internal formats are not actual image formats, but are instead mapped into one of the specific compressed formats provided by the GL (or to an uncompressed base internal format if no appropriate compressed format is available). Generic compressed internal formats allow applications to use texture compression without needing to code to any particular compression algorithm. Generic compressed formats allow the use of texture compression across a wide range of platforms with differing compression algorithms and also allow future GL implementations to substitute improved compression methods transparently.

Compressed texture images can be obtained from the GL in uncompressed form by calling GetTexImage and in compressed form by calling GetCompressedTexImageARB. Queried compressed images can be saved and later reused by calling CompressedTexImage[123]DARB. Pre-compressed texture images do not need to be processed by the GL and should significantly improve texture loading performance relative to uncompressed images.

This extension does not define specific compressed image formats (e.g., S3TC, FXT1), nor does it provide means to encode or decode such images. To support images in a specific compressed format, a hardware vendor would:

(1) Provide a new extension defininig specific compressed <internalformat> and <format> tokens for TexImage[123]D, TexSubImage[123]D, CopyTexImage[12]D, CompressedTexImage[123]DARB, CompressedTexSubImage[123]DARB, and GetCompressedTexImageARB calls.

(2) Specify the encoding of compressed images of that specific format.

(3) Specify a method for deriving the size of compressed images of that specific format, using the <internalformat>, <width>, <height>, <depth> parameters, and (if necessary) the compressed image itself.

IP Status

No known intellectual property issues on this general extension.

Specific compression algorithms used to implement this extension (and any other specific texture compression extensions) may be protected and require licensing agreements.

Issues

(1) Should we define additional internal formats that strongly tie an underlying compression algorithm to the format?

RESOLVED: Not here. Explicit compressed formats will be provided by other extensions built on top of this one.

(2) Should we provide additional compression state that gives more control on the level/quality of compression? If so, how?

RESOLVED: Yes, as a hint. Could have also been implemented as a [0.0, 1.0] floating-point TexParameter "quality" state variable (such as the JPEG quality scale found in many apps). This control will affect only the speed (and quality) with which a driver compresses incoming images, but will not affect the compressed image format selected by the driver.

As the spec is currently formulated, the requirement that quality control not affect compression format selection could have been relaxed by loosening the invariance requirements (so that the quality control can affect the choice of internal format). The risk was the potential for subtle mipmap consistency issues if the hint changes.

(3) Most current compression algorithms handle primarily RGB and RGBA images. Does it make sense having generic compressed formats for alpha, intensity, luminance, and luminance-alpha?

RESOLVED: Yes. It is conceivable that some or all of these formats may be compressed. Implementations not having compression algorithms for these formats can simply choose not to compress and use the appropriate base internal format instead.

(4) Full GetTexImage support requires that the renderer decompress the whole image. Should this extra implementation burden be imposed on the renderer?

RESOLVED: Yes, returning the uncompressed image is a useful feature for evaluating the quality of the compressed image. A decompression engine may also be required for a number of other areas, including software rasterization.

(5) Full TexSubImage support may require that the renderer decompress portions of the image (or perhaps the whole image), do a merge, and then recompress. Even if this were done, portions of the image outside the "modified" area may also be modified due to lossy compression. Should this extra implementation burden be imposed on the renderer?

RESOLVED: No. To avoid the complications involved with modifying a compressed texture image, only the lower-left corner may be modified by TexSubImage. In addition, after calling TexSubImage, the "unmodified" portion of the image is left undefined. An INVALID_OPERATION error results from any other TexSubImage calls.

This behavior allows for the use of compressed images whose dimensions are not powers of two, which TexImage will not accept. The recommended sequence of calls for defining such images is to first call TexImage with a NULL <data> pointer and the image size parameters padded out to the next power of two, and then call CompressedTexSubImageARB or TexSubImage with <xoffset>, <yoffset>, and <zoffset> parameters of zero and the compressed data pointed to by <data>. This behavior also allows TexSubImage to be used as a light-weight replacement of TexImage, where only the image contents are modified.

Certain compressed formats may allow a wider variety of edits -- their specifications will document the restrictions under which these edits are permitted. it is impossible to document such restrictions for unknown generic formats. It is desirable to keep the behavior of generic formats and the specific formats they map to as consistent as possible.

(6) What do the return values of the component sizes (RED_BITS, GREEN_BITS, ...) give for compressed textures? Compressed proxy textures?

RESOLVED: Some behavior has to be defined. For both normal and proxy textures, we return the bit depths of an uncompressed sized image that would most closely match the quality of the compression algorithm for an "average" texture image. Since compressed image quality is highly data dependent, the actual compressed image quality may be better or worse than the renderer’s best guess at the best matching sized internal format. To implement this feature in a driver, it is expected that an error analysis would be done on a set of representative images, and the resultant "equivalent bit depths" would be hardwired constants.

(7) What should GetTexLevelParameter with TEXTURE_COMPRESSED_ IMAGE_SIZE_ARB return for existing uncompressed formats? For proxy textures?

RESOLVED: For both, an INVALID_OPERATION error results. The actual image to be compressed is not available for proxies, so actually compressing the specified image is not an option.

For uncompressed internal formats, we could return the actual amount of memory taken by the texture image. Such a mechanism might be useful as a metric of "how much space does this texture image take". It’s not particularly useful for an application based texture management scheme, since there is no information available indicating the amount of available memory. In addition, because of implementation-dependent hardware constraints, the amount of texture memory consumed by a texture object is not necessarily equal to the sum of the memory consumed by each of its mipmaps. The OpenGL ARB decided against adopting this behavior when this specification was approved.

(8) What about texture borders?

RESOLVED: Not a problem for generic compressed formats since a base internal format can be used if borders are not supported in the compressed image format. Borders may pose problems for specific compression extensions, and compressed textures with borders might well be disallowed by those extensions.

(9) Should certain pixel operations be disallowed for compressed texture internal formats (e.g., PixelStorage, PixelTransfer)? What about byte swapping?

RESOLVED: For uncompressed source images, all pixel storage and pixel transfer modes will be applied prior to compression. For compressed source images, all pixel storage and transfer modes will be ignored. The encoding of compressed images should be specified as a byte stream that matches the disk file format defined for the corresponding image type.

(10) Should functionality be provided to allow applications to save compressed images to disk and reuse them in subsequent runs without programming to specific formats? If so, how?

RESOLVED: Yes. This can be done without knowledge of specific compression formats in the following manner:

* Call TexImage with an uncompressed image and a generic compressed internal format. The texture image will be compressed by the GL, if possible.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_COMPRESSED_ARB to determine if the GL was able to store the image in compressed form.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_INTERNAL_FORMAT to determine the specific compressed image format in which the image is stored.

* Call GetTexLevelParameteriv with a <value> of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB to determine the size (in ubytes) of the compressed image that will be returned by the GL. Allocate a buffer of at least this size.

* Call GetCompressedTexImageARB. The GL will write the compressed texture image into the allocated buffer.

* Save the returned compressed image to disk, along with the associated width, height, depth, border parameters and the returned values of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB and TEXTURE_INTERNAL_FORMAT.

* Load the compressed image and its parameters, and call CompressedTexImage_[123]DARB to use the compressed image. The value of TEXTURE_INTERNAL_FORMAT should be used as <internalFormat> and the value of TEXTURE_COMPRESSED_IMAGE_SIZE_ARB should be used as <imageSize>.

The saved images will be valid as long as they are used on a device supporting the returned <internalFormat> parameter. If the saved images are used on a device that does not support the compressed internal format, an INVALID_ENUM error would be generated by the call to CompressedTexImage_[123]D because of the unknown format.

Note also that to reliably determine if the GL will compress an image without actually compressing it, an application need only define a proxy texture image and query TEXTURE_COMPRESSED_ARB as above.

(11) Without knowing of the compressed image format, there is no convenient way for the client-side GLX library or tracing tools to ascertain the size of a compressed texture image when sending a TexImage1D, TexImage2D, or TexImage3D packet or interpret pixel storage modes. To complicate matters further, it is possible to create both indirect (that might not understand an image format) and direct rendering contexts (that might understand an image format) on the same renderer. How should this be solved?

RESOLVED: A separate set of CompressedTexImage and CompressedTexSubImage calls has been created that allows libraries to pass compressed images along to the renderer without needing to understand their specific image formats or how to interpret pixel storage modes.

(12) Are the CompressedTexImage[123]DARB entry points really needed?

RESOLVED: Yes. To robustly support images of unknown format, specific compressed entry points are required. While the extension does not support images in a completely unspecified format (early drafts did), having a separate call means that GLX and tools such as GLS (stream encoder) do not need intimate knowledge of every compressed image format. Having separate calls also cleanly solves the problem where pixel storage and pixel transfer operations apply if and only if the source image is uncompressed.

(13) Is variable-ratio compression supported?

RESOLVED: Yes. Fixed-ratio compression is currently the predominant texture compression format, but this spec should not preclude the use of other compression schemes.

(14) Should the <imageSize> parameter be validated on CompressedTexImage calls?

RESOLVED: Yes. Enforcement overhead is generally trivial. Without enforcement, an application could specify incorrect image sizes but notice them only when run on an indirect renderer, causing portability problems. There is also a reliability issue with respect to the GLX environment -- if the compressed image size provided by the user is less than the required image size, the GLX server may run off the end of the image and access invalid memory. A size check may thus be desirable to prevent server crashes (even though that could be considered an "undefined" result).

While enforcing correct <imageSize> parameters is trivial for current compressed internal formats, it might not be reasonable on others (particular variable-ratio compression formats). For such formats, this restriction should be overridden in the spec defining the formats. The <imageSize> check was made mandatory only in the final draft approved at the March 2000 OpenGL ARB meeting.

(15) Should TexImage calls fall back to uncompressed image formats when <internalformat> is a specific compressed format but its use in combination with other parameter values passed is not supported by the renderer?

RESOLVED: Yes. Advantages: Works in exactly the same way as generic formats, meaning no extra code/error checking. Inherent limitations of TexImage on specific formats should be documented in their specs and observed by their users. One simple query can detect fallback cases. Disadvantages: Silent fallback to a format not requested by the user.

(16) Should the texture format invariance requirements disallow scanning of the image data to select a compression method? What about for a base (uncompressed) internal format?

RESOLVED: The primary issue is mipmap consistency. The 1.2.1 spec defines a set of mipmaps as consistent if all are specified using the same internal format. However, it doesn’t require that all mipmaps are allocated using the same format -- the renderer is responsible for ensuring mipmap consistency if it selects different formats for different images. There is no reason to disallow scanning for base internal formats; the renderer is responsible for doing the right thing.

The selection of a specific compressed internal format is different. It must be independent of the the image data because the GL treats the texture image as though it were specified using the specific compressed internal format chosen by the renderer.

(17) Should functionality be provided to enumerate the specific compressed formats supported by the renderer? If so, how and what will it accomplish?

RESOLVED: Yes. A glGet* query is added to return the number of compressed internal formats supported by the renderer and the <internalformat> tokens for each. These tokens can subsequently be used as <internalformat> parameters for normal TexImage calls and the new CompressedTexImage calls.

Providing an internal format enumeration allows applications to weigh the suitability of the various compression methods provided to it by the renderer without needing specific knowledge of the formats. Applications can query the component sizes (see issue 6) to determine the base format and approximate precision. Applications can directly evaluate image compression quality by having the renderer generate compressed texture images (using the returned <internalformat> values) and return them in uncompressed form using GetTexImage. Applications should also be aware that the use of the internal formats returned by this query is subject to the restrictions imposed by the specification defining them. The use of proxy textures allows the application to determine if a specific set of TexImage parameters is supported for a given internal format.

The renderer should enumerate all supported compression formats EXCEPT those that operate fundamentally differently from a normal uncompressed format. For example, the DirectX DXT1 compression format is fundamentally an RGB format, but it has a "transparent" encoding where the red, green, and blue component values are forced to zero, regardless of their original (uncompressed) values. Since such formats may have caveats that must be understood before being used, they should not be enumerated by this query.

This allows for forward compatibility -- an application can exploit compression techniques provided by future renderers.

(18) Should the separate GetCompressedTexImageARB function exist, or is GetTexImage with special <format> and/or <type> parameters sufficient?

RESOLVED: Provide a separate GetCompressedTexImageARB function. The primary rationale is for GLX indirect rendering. The client GetTexImage would require information to determine if an image is uncompressed (and should be decoded using pixel storage state) or compressed (pixel storage ignored). In addition, if the image is compressed, the actual image size would be required, but the only image size that could be inferred from the GLX protocol is padded out to a multiple of four bytes. A separate call is the cleanest solution to both issues.

New Procedures and Functions

```
    void CompressedTexImage3DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 sizei height, sizei depth,
                                 int border, sizei imageSize,
                                 const void *data);
    void CompressedTexImage2DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 sizei height, int border,
                                 sizei imageSize, const void *data);
    void CompressedTexImage1DARB(enum target, int level,
                                 enum internalformat, sizei width,
                                 int border, sizei imageSize,
                                 const void *data);
    void CompressedTexSubImage3DARB(enum target, int level,
                                    int xoffset, int yoffset,
                                    int zoffset, sizei width,
                                    sizei height, sizei depth,
                                    enum format, sizei imageSize,
                                    const void *data);
    void CompressedTexSubImage2DARB(enum target, int level,
                                    int xoffset, int yoffset,
                                    sizei width, sizei height,
                                    enum format, sizei imageSize,
                                    const void *data);
    void CompressedTexSubImage1DARB(enum target, int level,
                                    int xoffset, sizei width,
                                    enum format, sizei imageSize,
                                    const void *data);
    void GetCompressedTexImageARB(enum target, int lod,
                                  void *img);
```

New Tokens

Accepted by the <internalformat> parameter of TexImage1D, TexImage2D, TexImage3D, CopyTexImage1D, and CopyTexImage2D:

```
        COMPRESSED_ALPHA_ARB                            0x84E9
        COMPRESSED_LUMINANCE_ARB                        0x84EA
        COMPRESSED_LUMINANCE_ALPHA_ARB                  0x84EB
        COMPRESSED_INTENSITY_ARB                        0x84EC
        COMPRESSED_RGB_ARB                              0x84ED
        COMPRESSED_RGBA_ARB                             0x84EE
```

Accepted by the <target> parameter of Hint and the <value> parameter of GetIntegerv, GetBooleanv, GetFloatv, and GetDoublev:

```
        TEXTURE_COMPRESSION_HINT_ARB                    0x84EF
```

Accepted by the <value> parameter of GetTexLevelParameter:

```
TEXTURE_COMPRESSED_IMAGE_SIZE_ARB               0x86A0
TEXTURE_COMPRESSED_ARB                          0x86A1
```

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

```
NUM_COMPRESSED_TEXTURE_FORMATS_ARB              0x86A2
COMPRESSED_TEXTURE_FORMATS_ARB                  0x86A3
```

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

None.

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

Modify **Section 3.8.1, Texture Image Specification (p.113)**

(p.113, modify 3rd paragraph) <internalformat> may be specified as one of the six base internal format symbolic constants listed in table 3.15, as one of the sized internal format symbolic constants listed in table 3.16, as one of the specific compressed internal format symbolic constants listed in table 3.16.1, or as one of the six generic compressed internal format symbolic constants listed in table 3.16.2.

(p.113, add after 3rd paragraph)

The ARB_texture_compression specification provides no specific compressed internal formats but does provide a mechanism to obtain the enums for such formats provided by other specifications. If the ARB_texture_compression extension is supported, the number of specific compressed internal format symbolic constants supported by the renderer can be obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS_ARB. The set of specific compressed internal format symbolic constants supported by the renderer can be obtained by querying the value of COMPRESSED_TEXTURE_FORMATS_ARB. The only symbolic constants returned by this query are those suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

Generic compressed internal formats are never used directly as the internal formats of texture images. If <internalformat> is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL’s choosing with the same base internal format. If no specific compressed format is available, <internalformat> is instead replaced by the corresponding base internal format. If <internalformat> is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), <internalformat> is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

(p.113, modify 4th paragraph) ... If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format’s components, as specified in table 3.15. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

(p.113, 5th paragraph) A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any TexImage3D, TexImage2D, or TexImage1D (see below) parameter (except <target>, but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed image format may not be affected by the <data> parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.8.7.

Add Table 3.16.1: Specific Compressed Internal Formats

Compressed Internal Format	Base Internal Format
none provided here -- defined by dependent extensions	

Add Table 3.16.2: Generic Compressed Internal Formats

Generic Compressed Internal Format	Base Internal Format
COMPRESSED_ALPHA_ARB	ALPHA
COMPRESSED_LUMINANCE_ARB	LUMINANCE
COMPRESSED_LUMINANCE_ALPHA_ARB	LUMINANCE_ALPHA
COMPRESSED_INTENSITY_ARB	INTENSITY
COMPRESSED_RGB_ARB	RGB
COMPRESSED_RGBA_ARB	RGBA

Modify **Section 3.8.2, Alternate Image Specification**

(add to end of TexSubImage discussion, p.123)

Texture images with compressed internal formats may be stored in such a way that it is not possible to edit an image with subimage commands without having to decompress and recompress the texture image being edited. Even if the image were edited in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary edits to texture images with compressed internal formats. Calling TexSubImage3D, CopyTexSubImage3D, TexSubImage2D, CopyTexSubImage2D, TexSubImage1D, or CopyTexSubImage1D will result in an INVALID_OPERATION error if <xoffset>, <yoffset>, or <zoffset> is not equal to -b_s (border). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily edited.

(add new subsection at end of section, p.123)

Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format. The ARB_texture_compression extension defines no such formats, but provides the mechanisms for other extensions that do.

The commands

```
  void CompressedTexImage1DARB(enum target, int level,
                               enum internalformat, sizei width,
                               int border, sizei imageSize,
                               const void *data);
  void CompressedTexImage2DARB(enum target, int level,
                               enum internalformat, sizei width,
                               sizei height, int border,
                               sizei imageSize, const void *data);
  void CompressedTexImage3DARB(enum target, int level,
                               enum internalformat, sizei width,
                               sizei height, sizei depth,
                               int border, sizei imageSize,
                               const void *data);
```

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The <target>, <level>, <internalformat>, <width>, <height>, <depth>, and <border> parameters have the same meaning as in TexImage1D, TexImage2D, and TexImage3D. <data> points to compressed image data stored in the compressed image format corresponding to <internalformat>. Since this extension provides no specific image formats, using any of the six generic compressed internal formats as <internalformat> will result in an INVALID_ENUM error.

For all other compressed internal formats, the compressed image will be decoded according to the specification defining the <internalformat> token. Compressed texture images are treated as an array of <imageSize> ubytes beginning at address <data>. All pixel storage and pixel transfer modes are ignored when decoding a compressed texture image. If the <imageSize> parameter is not consistent with the format, dimensions, and contents of the compressed image, an INVALID_VALUE error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Specific compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might be supported only for 2D textures or may not allow non-zero <border> values. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB will not result in an INVALID_OPERATION error if the following restrictions are satisfied:

- <data> points to a compressed texture image returned by GetCompressedTexImageARB (Section 6.1.4).

- <target>, <level>, and <internalformat> match the <target>, <level> and <format> parameters provided to the GetCompressedTexImageARB call returning <data>.

- <width>, <height>, <depth>, <border>, <internalformat>, and <imageSize> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_BORDER, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE_ARB for image level <level> in effect at the time of the GetCompressedTexImageARB call returning <data>.

This guarantee applies not just to images returned by GetCompressedTexImageARB, but also to any other properly encoded compressed texture image of the same size and format.

The commands

```
void CompressedTexSubImage1DARB(enum target, int level,
                                int xoffset, sizei width,
                                enum format, sizei imageSize,
                                const void *data);
void CompressedTexSubImage2DARB(enum target, int level,
                                int xoffset, int yoffset,
                                sizei width, sizei height,
                                enum format, sizei imageSize,
                                const void *data);
void CompressedTexSubImage3DARB(enum target, int level,
                                int xoffset, int yoffset,
                                int zoffset, sizei width,
                                sizei height, sizei depth,
                                enum format, sizei imageSize,
                                const void *data);
```

respecify only a rectangular region of an existing texture array, with incoming data stored in a known compressed image format. The <target>, <level>, <xoffset>, <yoffset>, <zoffset>, <width>, <height>, and <depth> parameters have the same meaning as in TexSubImage1D, TexSubImage2D, and TexSubImage3D. <data> points to compressed image data stored in the compressed image format corresponding to <format>. Since this extension provides no specific image formats, using any of these six generic compressed internal formats as <format> will result in an INVALID_ENUM error.

The image pointed to by <data> and the <imageSize> parameter are interpreted as though they were provided to CompressedTexImage1DARB, CompressedTexImage2DARB, and CompressedTexImage3DARB. These commands do not provide for image format conversion, so an INVALID_OPERATION error results if <format> does not match the internal format of the texture image being modified. If the <imageSize> parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an INVALID_VALUE error results.

As with CompressedTexImage calls, compressed internal formats may have additional restrictions on the use of the compressed image specification calls or parameters. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, CompressedTexSubImage3DARB will not result in an INVALID_OPERATION error if the following restrictions are satisfied:

* <data> points to a compressed texture image returned by GetCompressedTexImageARB (Section 6.1.4).

* <target>, <level>, and <format> match the <target>, <level> and <format> parameters provided to the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <format>, and <imageSize> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE_ARB for image level <level> in effect at the time of the GetCompressedTexImageARB call returning <data>.

* <width>, <height>, <depth>, <format> match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_INTERNAL_FORMAT currently in effect for image level <level>.

* <xoffset>, <yoffset>, and <zoffset> are all "-<b>", where <b> is the value of TEXTURE_BORDER currently in effect for image level <level>.

This guarantee applies not just to images returned by GetCompressedTexImageARB, but also to any other properly encoded compressed texture image of the same size.

Calling CompressedTexSubImage3D, CompressedTexSubImage2D, or CompressedTexSubImage1D will result in an INVALID_OPERATION error if <xoffset>, <yoffset>, or <zoffset> is not equal to -b_s (border), or if <width>, <height>, and <depth> do not match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, or TEXTURE_DEPTH, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily edited.

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

None.

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

Modify **Section 5.6, Hints** (p.180)

(p.180, modify first paragraph)

...; FOG_HINT, indicating whether fog calculations are done per pixel or per vertex; and TEXTURE_COMPRESSION_HINT_ARB, indicating the desired quality and performance of compressing texture images.

For the texture compression hint, a <hint> of FASTEST indicates that texture images should be compressed as quickly as possible, while NICEST indicates that the texture images be compressed with as little image degradation as possible. FASTEST should be used for one-time texture compression, and NICEST should be used if the compression results are to be retrieved by GetCompressedTexImageARB (Section 6.1.4) for reuse.

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

Modify **Section 6.1.3, Enumerated Queries** (p.183)

(p.183, modify next-to-last paragraph)

For texture images with uncompressed internal formats, queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. For texture images with a compressed internal format, the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation’s compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations. ...

(p.183, add to end of next-to-last paragraph)

TEXTURE_COMPRESSED_IMAGE_SIZE_ARB returns the size (in ubytes) of the compressed texture image that would be returned by GetCompressedTexImageARB (Section 6.1.4). Querying TEXTURE_COMPRESSED_IMAGE_SIZE_ARB is not allowed on texture images with an uncompressed internal format or on proxy targets and will result in an INVALID_OPERATION error if attempted.

Modify **Section 6.1.4, Texture Queries** (p.184)

(add immediately after the GetTexImage section and before the IsTexture section)

The command

```
  void GetCompressedTexImageARB(enum target, int lod,
                                void *img);
```

is used to obtain texture images stored in compressed form. The parameters <target>, <lod>, and <img> are interpreted in the same manner as in GetTexImage. When called, GetCompressedTexImageARB writes TEXTURE_COMPRESSED_IMAGE_SIZE_ARB ubytes of compressed image data to the memory pointed to by <img>. The compressed image data is formatted according to the specification defining INTERNAL_FORMAT. All pixel storage and pixel transfer modes are ignored when returning a compressed texture image.

Calling GetCompressedTexImageARB with an <lod> value less than zero or greater than the maximum allowable causes an INVALID_VALUE error. Calling GetCompressedTexImageARB with a texture image stored with an uncompressed internal format causes an INVALID_OPERATION error.

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

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

(Add after GetTexImage to Section 2.2.2 of the GLX 1.3 encoding spec, p.74)

```
GetCompressedTexImageARB

    1       CARD8               opcode (X assigned)
    1       160                 GLX opcode
    2       4                   request length
    4       GLX_CONTEXT_TAG     context tag
    4       ENUM                target
    4       INT32               level

 -->

    1       1                   Reply
    1       1                   unused
    2       CARD16              sequence number
    4       n                   reply length
    8                           unused
    4       INT32               compressed image size (in bytes) --
                                  should be between 4n-3 and 4n
    12                          unused
    4*n     LISTofBYTE          teximage
```

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

Since pixel storage modes do not apply to compressed texture images, teximage is simply an array of bytes. The client library will ignore pixel storage modes and should copy only <compressed image size> bytes, regardless of the value of <reply length>.

(Add to end of Section 2.3 of the GLX 1.3 encoding spec, p.147)

CompressedTexImage1DARB

```
    2       32+n+p          rendering command length
    2       214             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       ENUM            internalformat
    4       INT32           width
    4                       unused
    4       INT32           border
    n       LISTofBYTE      image
    4       INT32           imageSize
    p                       unused, p=pad(n)
```

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

```
    4       36+n+p          rendering command length
    4       214             rendering command opcode
```

CompressedTexImage2DARB

```
    2       32+n+p          rendering command length
    2       215             rendering command opcode
    4       ENUM            target
    4       INT32           level
    4       ENUM            internalformat
    4       INT32           width
    4       INT32           height
    4       INT32           border
    4       INT32           imageSize
    n       LISTofBYTE      image
    p                       unused, p=pad(n)
```

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

```
    4       36+n+p          rendering command length
    4       215             rendering command opcode
```

CompressedTexImage3DARB

2	36+n+p	rendering command length
2	216	rendering command opcode
4	ENUM	target
4	INT32	level
4	INT32	internalformat
4	INT32	width
4	INT32	height
4	INT32	depth
4	INT32	border
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

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

4	36+n+p	rendering command length
4	216	rendering command opcode

CompressedTexSubImage1DARB

2	36+n+p	rendering command length
2	217	rendering command opcode
4	ENUM	target
4	INT32	level
4	INT32	xoffset
4		unused
4	INT32	width
4		unused
4	ENUM	format
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

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

4	40+n+p	rendering command length
4	217	rendering command opcode

CompressedTexSubImage2DARB

2	36+n+p	rendering command length
2	218	rendering command opcode
4	ENUM	target
4	INT32	level
4	INT32	xoffset
4	INT32	yoffset
4	INT32	width
4	INT32	height
4	ENUM	format
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

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

4	40+n+p	rendering command length
4	218	rendering command opcode

CompressedTexSubImage3DARB

2	44+n+p	rendering command length
2	219	rendering command opcode
4	ENUM	target
4	INT32	level
4	INT32	xoffset
4	INT32	yoffset
4	INT32	zoffset
4	INT32	width
4	INT32	height
4	INT32	depth
4	ENUM	format
4	INT32	imageSize
n	LISTofBYTE	image
p		unused, p=pad(n)

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

4	48+n+p	rendering command length
4	219	rendering command opcode

Errors

Errors for compressed TexImage and TexSubImage calls specific to compression:

INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D, TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or CopyTexSubImage3D if the internal format of the texture image is compressed and <xoffset>, <yoffset>, or <zoffset> does not equal -b, where b is value of TEXTURE_BORDER.

INVALID_VALUE is generated by CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if the entire texture image is not being edited: if <xoffset>, <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is less than w+b, <yoffset> + <height> is less than h+b, or <zoffset> + <depth> is less than d+b, where b is the value of TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.

INVALID_ENUM is generated by CompressedTexImage1DARB, CompressedTexImage2DARB, or CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, if <internalformat> is any of the six generic compressed internal formats (e.g., COMPRESSED_RGBA_ARB)

INVALID_OPERATION is generated by CompressedTexImage1DARB, CompressedTexImage2DARB, CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, if any parameter combinations are not supported by the specific compressed internal format. Such invalid combinations are documented in the specification defining the internal format.

INVALID_VALUE is generated by CompressedTexImage1DARB, CompressedTexImage2DARB, or CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, if <imageSize> is not consistent with the format, dimensions, and contents of the specified image. The appropriate value for the <imageSize> parameter is documented in the specification defining the compressed internal format.

Undefined results (including abnormal program termination) are generated by CompressedTexImage1DARB, CompressedTexImage2DARB, or CompressedTexImage3DARB, CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB, is not encoded in a manner consistent with the specification defining the internal format.

INVALID_OPERATION is generated by CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if <format> does not match the internal format of the texture image being modified.

INVALID_OPERATION is generated by GetTexLevelParameter[if]v if <target> is PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, or PROXY_TEXTURE_3D and <value> is TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.

INVALID_OPERATION is generated by GetTexLevelParameter[if]v if the internal format of the queried texture image is not compressed and <value> is TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.

INVALID_OPERATION is generated by GetCompressedTexImageARB if the internal format of the queried texture image is not compressed.

Errors for compressed TexImage and TexSubImage calls not specific to compression:

```
INVALID_ENUM is generated by CompressedTexImage3DARB or
CompressedTexSubImage3DARB if <target> is not TEXTURE_3D.

INVALID_ENUM is generated by CompressedTexImage2DARB or
CompressedTexSubImage2DARB if <target> is not TEXTURE_2D,
TEXTURE_CUBE_MAP_POSITIVE_X_ARB, TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, or TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.

INVALID_ENUM is generated by CompressedTexImage1DARB or
CompressedTexSubImage1DARB if <target> is not TEXTURE_1D.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, CompressedTexImage3DARB,
CompressedTexSubImage1DARB, CompressedTexSubImage1DARB, or
CompressedTexSubImage3DARB if <level> is negative.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, CompressedTexImage3DARB,
CompressedTexSubImage1DARB, CompressedTexSubImage2DARB, or
CompressedTexSubImage3DARB, if <width>, <height>, or <depth> is negative.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, or CompressedTexImage3DARB if <width>, <height>,
or <depth> can not be represented as 2^k+2 for some integer value k.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, or CompressedTexImage3DARB if <border> is not
zero or one.

INVALID_VALUE is generated by CompressedTexImage1DARB,
CompressedTexImage2DARB, CompressedTexImage3DARB,
CompressedTexSubImage1DARB, CompressedTexSubImage1DARB, or
CompressedTexSubImage3DARB if the call is made between a call to Begin and
the corresponding call to End.

INVALID_VALUE is generated by CompressedTexSubImage1DARB,
CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if <xoffset>,
<yoffset>, or <zoffset> is less than -b, <xoffset> + <width> is greater
than w+b, <yoffset> + <height> is greater than h+b, or <zoffset> + <depth>
is greater than d+b, where b is the value of TEXTURE_BORDER, w is the
value of TEXTURE_WIDTH, h is the value of TEXTURE_HEIGHT, and d is the
value of TEXTURE_DEPTH.

INVALID_VALUE is generated by GetCompressedTexImageARB if <lod> is
negative or greater than the maximum allowable level.
```

New State

(table 6.12, p.202)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSED_IMAGE_SIZE_ARB	n x Z+	GetTexLevelParameter	0	size (in ubytes) of xD compressed texture image i.	3.8	-
TEXTURE_COMPRESSED_ARB	n x B	GetTexLevelParameter	FALSE	True if xD image i has a compressed internal format	3.8	-

(table 6.23, p.213)

Get Value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSION_HINT_ARB	Z_3	GetIntegerv	DONT_CARE	Texture compression quality hint	5.6	hint

(table 6.25, p. 215)

Get Value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
NUM_COMPRESSED_TEXTURE_FORMATS_ARB	Z	GetIntegerv	0	Number of enumerated compressed texture formats	3.8	-
COMPRESSED_TEXTURE_FORMATS_ARB	0* x Z	GetIntegerv	-	Enumerated compressed texture formats	3.8	-

Revision History

1.03, 05/23/00 prbrown1: Removed stray "None." paragraph in modifications to Chapter 5.

1.02, 05/08/00 prbrown1: Fixed prototype of GetCompressedTexImageARB (no "const" qualifiers) in "New Procedures and Functions" section. Changed <internalformat> parameter of CompressedTexImage functions to be an "enum" instead of an "int". "int" was carried over only on TexImage calls as a 1.0 legacy -- the newer CopyTexImage call takes an "enum".

1.01, 04/11/00 prbrown1: Minor bug fixes to the first published version. Fixed prototypes to match extension spec standards (no "GL" type prefixes). Fixed a couple erroneous function names. Added "const" qualifier to prototypes involving image data not modified by the GL. Added text to indicate that compressed formats apply to texture maps supported by GL_ARB_texture_cube_map.

1.0, 03/24/00 prbrown1: Applied changes approved as part of the extension at the March 2000 ARB meeting, as follows:

- CompressedTexSubImage: Only allowed if the

```
                                 entire image is replaced.  Document that this
                                 restriction can be relaxed for specific
                                 compression extensions.
                               * Renamed TEXTURE_IMAGE_SIZE_ARB to
                                 TEXTURE_COMPRESSED_IMAGE_SIZE_ARB.
                               * Querying image size on uncompressed images is
                                 now an INVALID_OPERATION error.
                               * INVALID_VALUE error is generated if <imageSize>
                                 is inconsistent with the image data.  This
                                 restriction may be overridden by specific
                                 extensions only if requiring an image size
                                 check is unreasonable.
                               * Added documentaion of undefined behavior for
                                 CompressedTexImage/SubImage if the image data
                                 is encoded in a manner inconsistent with the
                                 spec defining the compressed image format.
                               * Fixed issue (16).  Text was truncated.
                               * Modified invariance section.  <data> can not
                                 affect the choice of compressed internal
                                 format, but can theoretically affect regular
                                 component resolution.
                               * Add new function GetCompressedTexImage to deal
                                 with subtle GLX issues.
                               * GLX protocol for CompressedTexImage/SubImage
                                 and GetCompressedTexImage holds both a padded
                                 image size (for GLX data transfer) and actual
                                 image size (for packing in user buffers).

                               Minor wording clean-ups.

                               Added enum and GLX opcode values allocated from
                               OpenGL Extensions and GLX registries.

    0.81, 03/07/00 prbrown1: Fixed error documentation for TexSubImage calls
                               of arbitrary alignment (did not document that the
                               internal format had to be compressed).  Removed
                               references to CopyTexImage3D, which doesn’t
                               actually exist.

                               Per Kurt Akeley suggestions: (1) Renamed
                               TexImageCompressed to CompressedTexImage to
                               conform with naming conventions, (2) clarified
                               that the main feature distinguishing
                               CompressedTex[Sub]Image calls from normal
                               Tex[Sub]Image calls is compressed input data, (3)
                               added query to explicitly determine whether the
                               internal format of a texture is compressed.

    0.8,  02/23/00 prbrown1: Marked previously unresolved issues as resolved
                               per the ARB working group.  Added docs for errors
                               not specific to compression for the new
                               CompressedTexImage and CompressedTexSubImage
                               calls.  Added queries to enumerate specific
                               compressed texture formats.
    0.76, 02/16/00 prbrown1: Removed "gl" and "GL_" prefixes.
    0.75, 02/07/00 prbrown1: Incorporated feedback from 12/99 ARB meeting
                               and a number of other revisions.
    0.7,  12/03/99 prbrown1: Incorporated comments from public review of 0.2
                             document.
    0.2,  10/28/99 prbrown1: Renamed to ARB_texture_compression.  Significant
                             functional changes.
    0.11, 10/21/99 prbrown1: Edits suggested by 3dfx.
    0.1,  10/19/99 prbrown1: Initial revision.
```

Name

ARB_transpose_matrix

Name Strings

GL_ARB_transpose_matrix

Contact

David Blythe (blythe ’at’ sgi.com)

Status

Complete. Approved by ARB on 12/8/1999

Version

Last Modified Date: January 3, 2000
Author Revision: 1.3

Number

ARB Extension #3

Dependencies

This extensions is written against the OpenGL 1.2 Specification.
May be implemented in any version of OpenGL.

Overview

New functions and tokens are added allowing application matrices stored in row major order rather than column major order to be transferred to the OpenGL implementation. This allows an application to use standard C-language 2-dimensional arrays (m[row][col]) and have the array indices match the expected matrix row and column indexes. These arrays are referred to as transpose matrices since they are the transpose of the standard matrices passed to OpenGL.

This extension adds an interface for transfering data to and from the OpenGL pipeline, it does not change any OpenGL processing or imply any changes in state representation.

IP Status

No IP is believed to be involved.

Issues

* Why do this?

  It’s very useful for layered libraries that desire to use two dimensional C arrays as matrices. It avoids having the layered library perform the transpose itself before calling OpenGL since most OpenGL implementations can efficiently perform the transpose while reading the matrix from client memory.

* Why not add a mode?

It’s substantially more confusing and complicated to add a mode. Simply adding two new entry points saves considerable confusion and avoids having layered libraries need to query the current mode in order to send a matrix with the correct memory layout.

* Why not a utility routine in GLU

It costs some performance. It is believed that most OpenGL implementations can perform the transpose in place with negligble performance penalty.

* Why use the name transpose?

It’s sure a lot less confusing than trying to ascribe unambiguous meaning to terms like row and column. It could be matrix_transpose rather than transpose_matrix though.

* Short Transpose to Trans?

New Procedures and Functions

```
void LoadTransposeMatrix{fd}ARB(T m[16]);
void MultTransposeMatrix{fd}ARB(T m[16]);
```

New Tokens

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

```
TRANSPOSE_MODELVIEW_MATRIX_ARB    0x84E3
TRANSPOSE_PROJECTION_MATRIX_ARB   0x84E4
TRANSPOSE_TEXTURE_MATRIX_ARB      0x84E5
TRANSPOSE_COLOR_MATRIX_ARB        0x84E6
```

Additions to Chapter 2 of the 1.2 OpenGL Specification (OpenGL Operation)

Add to Section 2.10.2 Matrices <before LoadIdentity>

LoadTransposeMatrixARB takes a 4x4 matrix stored in row-major order as

Let transpose(m,n) be defined as

```
n[0] = m[0];
n[1] = m[4];
n[2] = m[8];
n[3] = m[12];
n[4] = m[1];
n[5] = m[5];
n[6] = m[9];
n[7] = m[13];
n[8] = m[2];
n[9] = m[6];
n[10] = m[10];
n[11] = m[14];
n[12] = m[3];
n[13] = m[7];
n[14] = m[11];
n[15] = m[15];
```

The effect of LoadTransposeMatrixARB(m) is then the same as the effect of the command sequence

```
float n[16];
transpose(m,n)
LoadMatrix(n);
```

The effect of MultTransposeMatrixARB(m) is then the same as the effect of the command sequence

```
float n[16];
transpose(m,n);
MultMatrix(n);
```

Additions to Chapter 3 of the 1.2 OpenGL Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 OpenGL Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.2 OpenGL Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 OpenGL Specification (State and State Requests)

Matrices are queried and returned in their transposed form by calling GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev with <pname> set to TRANSPOSE_MODELVIEW_MATRIX_ARB, TRANSPOSE_PROJECTION_MATRIX_ARB, TRANSPOSE_TEXTURE_MATRIX_ARB, or TRANSPOSE_COLOR_MATRIX_ARB.
The effect of GetFloatv(TRANSPOSE_MODELVIEW_MATRIX_ARB,m) is then the same as the effect of the command sequence

```
        float n[16];
        GetFloatv(MODELVIEW_MATRIX_ARB,n);
        transpose(n,m);
```

Similar results occur for TRANSPOSE_PROJECTION_MATRIX_ARB, TRANSPOSE_TEXTURE_MATRIX_ARB, and TRANSPOSE_COLOR_MATRIX_ARB.

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

None

Additions to the GLX Specification

None

GLX Protocol

LoadTransposeMatrix and MultTransposeMatrix are layered on top of LoadMatrix and MultMatrix protocol performing client-side translation. The Get commands are passed over the wire as part of the generic Get protocol with no translation required.

Errors

No new errors, but error behavoir is inherited by the commands that the transpose commands are implemented on top of (LoadMatrix, MultMatrix, and Get*).

New State

None

TRANSPOSE_*_MATRIX_ARB refer to the same state as their non-transposed counterparts.

New Implementation Dependent State

None

Revision History

- Revision 1.1 - initial draft (18 Mar 1999)
- Revision 1.2 - changed to use layered specification and ARB affix (23 Nov 1999)
- Revision 1.3 - Minor tweaks to GLX protocol and Errors. (7 Dec 1999)

Conformance Testing

Load and Multiply the modelview matrix (initialized to identity each time) using LoadTransposeMatrixfARB and MultTransposeMatrixfARB with the matrix:

```
( 1  2  3  4 )
( 5  6  7  8 )
( 9 10 11 12 )
(13 14 15 16 )
```

and get the modelview matrix using TRANSPOSE_MODELVIEW_MATRIX_ARB and validate that the matrix is correct. Get the matrix using MODELVIEW_MATRIX and verify that it is the transpose of the above matrix. Load and Multiply the modelview matrix using LoadMatrixf and MultMatrixf with the above matrix and verify that the correct matrix is on the modelview stack using gets of MODELVIEW_MATRIX and TRANSPOSE_MODELVIEW_MATRIX_ARB.

Name

EXT_abgr

Name Strings

GL_EXT_abgr

Version

$Date: 1995/03/31 04:40:18 $ $Revision: 1.10 $

Number

1

Dependencies

None

Overview

EXT_abgr extends the list of host-memory color formats. Specifically, it provides a reverse-order alternative to image format RGBA. The ABGR component order matches the cpack Iris GL format on big-endian machines.

New Procedures and Functions

None

New Tokens

Accepted by the <format> parameter of DrawPixels, GetTexImage, ReadPixels, TexImage1D, and TexImage2D:

ABGR_EXT	0x8000

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

One entry is added to table 3.5 (DrawPixels and ReadPixels formats). The new table is:

Name	Type	Elements	Target Buffer
COLOR_INDEX	Index	Color Index	Color
STENCIL_INDEX	Index	Stencil value	Stencil
DEPTH_COMPONENT	Component	Depth value	Depth
RED	Component	R	Color
GREEN	Component	G	Color
BLUE	Component	B	Color
ALPHA	Component	A	Color
RGB	Component	R, G, B	Color
RGBA	Component	R, G, B, A	Color
LUMINANCE	Component	Luminance value	Color
LUMINANCE_ALPHA	Component	Luminance value, A	Color
ABGR_EXT	Component	A, B, G, R	Color

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

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

The new format is added to the discussion of Obtaining Pixels from the Framebuffer. It should read " If the <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained."

The new format is added to the discussion of Index Lookup. It should read "If <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, ABGR_EXT, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

One entry is added to tables 1 and 5 in the GLX Protocol Specification:

format	encoding
GL_ABGR_EXT	0x8000

Table A.2 is also extended:

format	nelements
GL_ABGR_EXT	4

Errors

None

New State

None

New Implementation Dependent State

None

Name

EXT_bgra

Name Strings

GL_EXT_bgra

Version

Microsoft revision 1.0, May 19, 1997 (drewb)
$Date: 1997/09/22 23:03:13 $ $Revision: 1.1 $

Number

129

Dependencies

None

Overview

EXT_bgra extends the list of host-memory color formats. Specifically, it provides formats which match the memory layout of Windows DIBs so that applications can use the same data in both Windows API calls and OpenGL pixel API calls.

New Procedures and Functions

None

New Tokens

Accepted by the <format> parameter of DrawPixels, GetTexImage, ReadPixels, TexImage1D, and TexImage2D:

BGR_EXT	0x80E0
BGRA_EXT	0x80E1

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

One entry is added to table 3.5 (DrawPixels and ReadPixels formats). The new table is:

Name	Type	Elements	Target Buffer
COLOR_INDEX	Index	Color Index	Color
STENCIL_INDEX	Index	Stencil value	Stencil
DEPTH_COMPONENT	Component	Depth value	Depth
RED	Component	R	Color
GREEN	Component	G	Color
BLUE	Component	B	Color
ALPHA	Component	A	Color
RGB	Component	R, G, B	Color
RGBA	Component	R, G, B, A	Color
LUMINANCE	Component	Luminance value	Color
LUMINANCE_ALPHA	Component	Luminance value,A	Color
BGR_EXT	Component	B, G, R	Color
BGRA_EXT	Component	B, G, R, A	Color

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

The new format is added to the discussion of Obtaining Pixels from the Framebuffer. It should read " If the <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR_EXT, BGRA_EXT, LUMINANCE, or LUMINANCE_ALPHA, and the GL is in color index mode, then the color index is obtained."

The new format is added to the discussion of Index Lookup. It should read "If <format> is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR_EXT, BGRA_EXT, LUMINANCE, or LUMINANCE_ALPHA, then the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Revision History

Original draft, revision 0.9, October 13, 1995 (drewb)
 Created
Minor revision, revision 1.0, May 19, 1997 (drewb)
 Removed Microsoft Confidential.

Name

EXT_blend_color

Name Strings

GL_EXT_blend_color

Version

$Date: 1995/03/31 04:40:19 $ $Revision: 1.7 $

Number

2

Dependencies

None

Overview

Blending capability is extended by defining a constant color that can be included in blending equations. A typical usage is blending two RGB images. Without the constant blend factor, one image must have an alpha channel with each pixel set to the desired blend factor.

New Procedures and Functions

```
void BlendColorEXT(clampf red,
                   clampf green,
                   clampf blue,
                   clampf alpha);
```

New Tokens

Accepted by the <sfactor> and <dfactor> parameters of BlendFunc:

```
CONSTANT_COLOR_EXT                0x8001
ONE_MINUS_CONSTANT_COLOR_EXT      0x8002
CONSTANT_ALPHA_EXT                0x8003
ONE_MINUS_CONSTANT_ALPHA_EXT      0x8004
```

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

```
BLEND_COLOR_EXT                   0x8005
```

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

The commands that control blending are now BlendFunc and BlendColorEXT. A constant color to be used in the blending equation is specified by BlendColorEXT. The four parameters are clamped to the range [0,1] before being stored. The default value for the constant blending color is (0,0,0,0).

The constant color can be used in both the source and destination blending factors. Four lines are added to table 4.1 and table 4.2:

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
DST_COLOR	(Rd/Kr, Gd/Kg, Bd/Kb, Ad/Ka)	
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd/Kr,Gd/Kg,Bd/Kb,Ad/Ka)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	NEW
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	NEW
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	NEW
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	NEW
SRC_ALPHA_SATURATE	(f, f, f, 1)	

Table 4.1: Values controlling the source blending function and the source blending values they compute. Ka = 2**m - 1, where m is the number of bits in the A color component. Kr, Kg, and Kb are similarly determined by the number of bits in the R, G, and B color components. f = min(As, 1-Ad) / Ka.

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs/Kr, Gs/Kg, Bs/Kb, As/Ka)	
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs/Kr,Gs/Kg,Bs/Kb,As/Ka)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	NEW
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	NEW
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	NEW
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	NEW

Table 4.2: Values controlling the destination blending function and the destination blending values they compute. Ka = 2**m - 1, where m is the number of bits in the A color component. Kr, Kg, and Kb are similarly determined by the number of bits in the R, G, and B color components.

Rc, Gc, Bc, and Ac are the four components of the constant blending color. These blend factors are not scaled by Kr, Kg, Kb, and Ka, because they are already in the range [0,1].

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
BlendColorEXT
    2           20              rendering command length
    2           4096            rendering command opcode
    4           FLOAT32  red
    4           FLOAT32  green
    4           FLOAT32  blue
    4           FLOAT32  alpha
```

Errors

INVALID_OPERATION is generated if BlendColorEXT is called between execution of Begin and the corresponding call to End.

New State

Get Value	Get Command	Type	Initial Value	Attrib
BLEND_COLOR_EXT	GetFloatv	C	(0,0,0,0)	color-buffer

New Implementation Dependent State

None

Name

EXT_blend_minmax

Name Strings

GL_EXT_blend_minmax

Version

$Date: 1995/03/31 04:40:34 $ $Revision: 1.3 $

Number

37

Dependencies

None

Overview

Blending capability is extended by respecifying the entire blend equation. While this document defines only two new equations, the BlendEquationEXT procedure that it defines will be used by subsequent extensions to define additional blending equations.

The two new equations defined by this extension produce the minimum (or maximum) color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

Issues

* I've prefixed the ADD token with FUNC, to indicate that the blend equation includes the parameters specified by BlendFunc. (The min and max equations don't.) Is this necessary? Is it too ugly? Is there a better way to accomplish the same thing?

New Procedures and Functions

```
void BlendEquationEXT(enum mode);
```

New Tokens

Accepted by the <mode> parameter of BlendEquationEXT:

```
FUNC_ADD_EXT                      0x8006
MIN_EXT                           0x8007
MAX_EXT                           0x8008
```

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

```
BLEND_EQUATION_EXT                0x8009
```

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

The GL Specification defines a single blending equation. This extension introduces a blend equation mode that is specified by calling BlendEquationEXT with one of three enumerated values. The default value FUNC_ADD_EXT specifies that the blending equation defined in the GL Specification be used. This equation is

```
 C’ = (Cs * S) + (Cd * D)

      /  1.0 C’ > 1.0
 C = (
      \   C’ C’ <= 1.0
```

where Cs and Cd are the source and destination colors, and S and D are as specified by BlendFunc.

If BlendEquationEXT is called with <mode> set to MIN_EXT, the blending equation becomes

```
 C = min (Cs, Cd)
```

Finally, if BlendEquationEXT is called with <mode> set to MAX_EXT, the blending equation becomes

```
 C = max (Cs, Cd)
```

In all cases the blending equation is evaluated separately for each color component.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
BlendEquationEXT
        2           8               rendering command length
        2           4097            rendering command opcode
        4           ENUM       mode
```

Errors

INVALID_ENUM is generated by BlendEquationEXT if its single parameter is not FUNC_ADD_EXT, MIN_EXT, or MAX_EXT.

INVALID_OPERATION is generated if BlendEquationEXT is executed between the execution of Begin and the corresponding execution to End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
BLEND_EQUATION_EXT	GetIntegerv	Z3	FUNC_ADD_EXT	color-buffer

New Implementation Dependent State

None

Name

EXT_blend_subtract

Name Strings

GL_EXT_blend_subtract

Version

$Date: 1995/03/31 04:40:39 $ $Revision: 1.4 $

Number

38

Dependencies

EXT_blend_minmax affects the definition of this extension

Overview

Two additional blending equations are specified using the interface defined by EXT_blend_minmax. These equations are similar to the default blending equation, but produce the difference of its left and right hand sides, rather than the sum. Image differences are useful in many image processing applications.

New Procedures and Functions

None

New Tokens

Accepted by the <mode> parameter of BlendEquationEXT:

```
 FUNC_SUBTRACT_EXT                 0x800A
 FUNC_REVERSE_SUBTRACT_EXT         0x800B
```

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

Two additional blending equations are defined. If BlendEquationEXT is called with <mode> set to FUNC_SUBTRACT_EXT, the blending equation becomes

```
C’ = (Cs * S) - (Cd * D)

     /  0.0 C’ < 0.0
C = (
     \   C’ C’ >= 0.0
```

where Cs and Cd are the source and destination colors, and S and D are as specified by BlendFunc.

If BlendEquationEXT is called with <mode> set to FUNC_REVERSE_SUBTRACT_EXT, the blending equation becomes

```
C’ = (Cd * D) - (Cs * S)

     /  0.0 C’ < 0.0
C = (
     \   C’ C’ >= 0.0
```

In all cases the blending equation is evaluated separately for each color component.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on EXT_blend_minmax

If this extension is supported, but EXT_blend_minmax is not, then this extension effectively defines the procedure BlendEquationEXT, its parameter FUNC_ADD_EXT, and the query target BLEND_EQUATION_EXT, as described in EXT_blend_minmax. It is therefore as though EXT_blend_minmax were also supported, except that equations MIN_EXT and MAX_EXT are not supported.

Errors

INVALID_ENUM is generated by BlendEquationEXT if its single parameter is not FUNC_ADD_EXT, MIN_EXT, MAX_EXT, FUNC_SUBTRACT_EXT, or FUNC_REVERSE_SUBTRACT_EXT.

INVALID_OPERATION is generated if BlendEquationEXT is executed between the execution of Begin and the corresponding execution to End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
BLEND_EQUATION_EXT	GetIntegerv	Z5	FUNC_ADD_EXT	color-buffer

New Implementation Dependent State

None

XXX - Not complete yet!!!

Name

EXT_compiled_vertex_array

Name Strings

GL_EXT_compiled_vertex_array

Version

$Date: 1996/11/21 00:52:19 $ $Revision: 1.3 $

Number

97

Dependencies

None

Overview

This extension defines an interface which allows static vertex array data to be cached or pre-compiled for more efficient rendering. This is useful for implementations which can cache the transformed results of array data for reuse by several DrawArrays, ArrayElement, or DrawElements commands. It is also useful for implementations which can transfer array data to fast memory for more efficient processing.

For example, rendering an M by N mesh of quadrilaterals can be accomplished by setting up vertex arrays containing all of the vertexes in the mesh and issuing M DrawElements commands each of which operate on 2 * N vertexes. Each DrawElements command after the first will share N vertexes with the preceding DrawElements command. If the vertex array data is locked while the DrawElements commands are executed, then OpenGL may be able to transform each of these shared vertexes just once.

Issues

- Is compiled_vertex_array the right name for this extension?

- Should there be an implementation defined maximum number of array elements which can be locked at a time (i.e. MAX_LOCKED_ARRAY_SIZE)?

  Probably not, the lock request can always be ignored with no resulting change in functionality if there are insufficent resources, and allowing the GL to define this limit can make things difficult for applications.

- Should there be any restrictions on what state can be changed while the vertex array data is locked?

  Probably not. The GL can check for state changes and invalidate any cached vertex state that may be affected. This is likely to cause a performance hit, so the preferred use will be to not change state while the vertex array data is locked.

New Procedures and Functions

```
void LockArraysEXT (int first, sizei count)
void UnlockArraysEXT (void)
```

New Tokens

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

```
ARRAY_ELEMENT_LOCK_FIRST_EXT         0x81A8
ARRAY_ELEMENT_LOCK_COUNT_EXT         0x81A9
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

After the discussion of InterleavedArrays, add a description of array compiling/locking.

The currently enabled vertex arrays can be locked with the command LockArraysEXT. When the vertex arrays are locked, the GL can compile the array data or the transformed results of array data associated with the currently enabled vertex arrays. The vertex arrays are unlocked by the command UnlockArraysEXT.

Between LockArraysEXT and UnlockArraysEXT the application should ensure that none of the array data in the range of elements specified by <first> and <count> are changed. Changes to the array data between the execution of LockArraysEXT and UnlockArraysEXT commands may affect calls may affect DrawArrays, ArrayElement, or DrawElements commands in non-sequential ways.

While using a compiled vertex array, references to array elements by the commands DrawArrays, ArrayElement, or DrawElements which are outside of the range specified by <first> and <count> are undefined.

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

LockArraysEXT and UnlockArraysEXT are not complied into display lists but are executed immediately.

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

None

Additions to the GLX Specification

XXX - Not complete yet!!!

GLX Protocol

XXX - Not complete yet!!!

Errors

INVALID_VALUE is generated if LockArrarysEXT parameter <first> is less than zero.

INVALID_VALUE is generated if LockArraysEXT parameter <count> is less than or equal to zero.

INVALID_OPERATION is generated if LockArraysEXT is called between execution of LockArraysEXT and corresponding execution of UnlockArraysEXT.

INVALID_OPERATION is generated if UnlockArraysEXT is called without a corresponding previous execution of LockArraysEXT.

INVALID_OPERATION is generated if LockArraysEXT or UnlockArraysEXT is called between execution of Begin and the corresponding execution of End.

New State

Get Value	Get Command	Type	Initial Value	Attrib
ARRAY_ELEMENT_LOCK_FIRST_EXT	GetIntegerv	Z+	0	client-vertex-array
ARRAY_ELEMENT_LOCK_COUNT_EXT	GetIntegerv	Z+	0	client-vertex-array

New Implementation Dependent State

None

Name

EXT_fog_coord

Name Strings

GL_EXT_fog_coord

Contact

Jon Leech, Silicon Graphics (ljp ’at’ sgi.com)

Status

Shipping (version 1.6)

Version

$Date: 1999/06/21 19:57:19 $ $Revision: 1.11 $

Number

149

Dependencies

OpenGL 1.1 is required.
The extension is written against the OpenGL 1.2 Specification.

Overview

This extension allows specifying an explicit per-vertex fog coordinate to be used in fog computations, rather than using a fragment depth-based fog equation.

Issues

* Should the specified value be used directly as the fog weighting factor, or in place of the z input to the fog equations?

  As the z input; more flexible and meets ISV requests.

* Do we want vertex array entry points? Interleaved array formats?

  Yes for entry points, no for interleaved formats, following the argument for secondary_color.

* Which scalar types should FogCoord accept? The full range, or just the unsigned and float versions? At the moment it follows Index(), which takes unsigned byte, signed short, signed int, float, and double.

  Since we’re now specifying a number which behaves like an eye-space distance, rather than a [0,1] quantity, integer types are less useful. However, restricting the commands to floating point forms only introduces some nonorthogonality.

Restrict to only float and double, for now.

* Interpolation of the fog coordinate may be perspective-correct or not. Should this be affected by PERSPECTIVE_CORRECTION_HINT, FOG_HINT, or another to-be-defined hint?

  PERSPECTIVE_CORRECTION_HINT; this is already defined to affect all interpolated parameters. Admittedly this is a loss of orthogonality.

* Should the current fog coordinate be queryable?

  Yes, but it’s not returned by feedback.

* Control the fog coordinate source via an Enable instead of a fog parameter?

  No. We might want to add more sources later.

* Should the fog coordinate be restricted to non-negative values?

  Perhaps. Eye-coordinate distance of fragments will be non-negative due to clipping. Specifying explicit negative coordinates may result in very large computed f values, although they are defined to be clipped after computation.

* Use existing DEPTH enum instead of FRAGMENT_DEPTH? Change name of FRAGMENT_DEPTH_EXT to FOG_FRAGMENT_DEPTH_EXT?

  Use FRAGMENT_DEPTH_EXT; FOG_FRAGMENT_DEPTH_EXT is somewhat misleading, since fragment depth itself has no dependence on fog.

New Procedures and Functions

```
void FogCoord[fd]EXT(T coord)
void FogCoord[fd]vEXT(T coord)
void FogCoordPointerEXT(enum type, sizei stride, void *pointer)
```

New Tokens

Accepted by the <pname> parameter of Fogi and Fogf:

```
FOG_COORDINATE_SOURCE_EXT         0x8450
```

Accepted by the <param> parameter of Fogi and Fogf:

```
FOG_COORDINATE_EXT                0x8451
FRAGMENT_DEPTH_EXT                0x8452
```

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

```
CURRENT_FOG_COORDINATE_EXT        0x8453
FOG_COORDINATE_ARRAY_TYPE_EXT     0x8454
FOG_COORDINATE_ARRAY_STRIDE_EXT   0x8455
```

Accepted by the <pname> parameter of GetPointerv:

```
FOG_COORDINATE_ARRAY_POINTER_EXT    0x8456
```

Accepted by the <array> parameter of EnableClientState and DisableClientState:

```
FOG_COORDINATE_ARRAY_EXT            0x8457
```

Additions to Chapter 2 of the OpenGL 1.2 Specification (OpenGL Operation)

These changes describe a new current state type, the fog coordinate, and the commands to specify it:

- (2.6, p. 12) Second paragraph changed to:

  "Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current fog coordinate may be used in processing each vertex."

- 2.6.3, p. 19) First paragraph changed to

  "The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, texture coordinates, and fog coordinates (Vertex, Color, Index, Normal, TexCoord, FogCoord)..."

- (2.7, p. 20) Insert the following paragraph following the third paragraph describing current normals:

```
"   The current fog coodinate is set using
      void FogCoord[fd]EXT(T coord)
      void FogCoord[fd]vEXT(T coord)."
```

  The last paragraph is changed to read:

  "The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, one floating-point value to store the current fog coordinate, four floating-point values to store the current RGBA color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial fog coordinate is zero. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial color index is 1."

- (2.8, p. 21) Added fog coordinate command for vertex arrays:

  Change first paragraph to read:

  "The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client’s address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to seven arrays: one each to store edge flags, texture coordinates, fog coordinates, colors, color indices, normals, and vertices. The commands"

Add to functions listed following first paragraph:

```
void FogCoordPointerEXT(enum type, sizei stride, void *pointer)
```

Add to table 2.4 (p. 22):

Command	Sizes	Types
FogCoordPointerEXT	1	float,double

Starting with the second paragraph on p. 23, change to add FOG_COORDINATE_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
    void EnableClientState(enum array)
    void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, FOG_COORDINATE_ARRAY_EXT, COLOR_ARRAY, INDEX_ARRAY, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, fog coordinate, color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
    void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, fog coordinate, color, color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, FogCoord<type>v, Color<size><type>v, Index<type>v, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable fog coordinate array for canned interleaved array formats. After the lines

```
    DisableClientState(EDGE_FLAG_ARRAY);
    DisableClientState(INDEX_ARRAY);
```

insert the line

```
DisableClientState(FOG_COORDINATE_ARRAY_EXT);
```

Substitute "seven" for every occurence of "six" in the final paragraph on p. 27.

- (2.12, p. 41) Add fog coordinate to the current rasterpos state.

Change the first sentence of the first paragraph to read

"The state required for the current raster position consists of three window coordinates x_w, y_w, and z_w, a clip coordinate w_c value, an eye coordinate distance, a fog coordinate, a valid bit, and associated data consisting of a color and texture coordinates."

Change the last paragraph to read

"The current raster position requires six single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, its eye coordinate distance, and its fog coordinate, a single valid bit, a color (RGBA color and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both (0,0,0,1), the fog coordinate is 0, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is (1,1,1,1), and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value."

- (3.10, p. 139) Change the second and third paragraphs to read

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

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

If the fog source (as defined below) is FRAGMENT_DEPTH_EXT, then c is the eye-coordinate distance from the eye, (0 0 0 1) in eye coordinates, to the fragment center. If the fog source is FOG_COORDINATE_EXT, then c is the interpolated value of the fog coordinate for this fragment. The equation and the fog source, along with either d or e and s, is specified with

```
void Fog{if}(enum pname, T param);
void Fog{if}v(enum pname, T params);
```

If <pname> is FOG_MODE, then <param> must be, or <param> must point to an integer that is one of the symbolic constants EXP, EXP2, or LINEAR, in which case equation 3.24, 3.25, or 3.26,, respectively, is selected for the fog calculation (if, when 3.26 is selected, e = s, results are undefined). If <pname> is FOG_COORDINATE_SOURCE_EXT, then <param> is or <params> points to an integer that is one of the symbolic constants FRAGMENT_DEPTH_EXT or FOG_COORDINATE_EXT. If <pname> is FOG_DENSITY, FOG_START, or FOG_END, then <param> is or <params> points to a value that is d, s, or e, respectively. If d is specified less than zero, the error INVALID_VALUE results."

- (3.10, p. 140) Change the last paragraph preceding section 3.11 to read

  "The state required for fog consists of a three valued integer to select the fog equation, three floating-point values d, e, and s, an RGBA fog color and a fog color index, a two-valued integer to select the fog coordinate source, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_COORDINATE_SOURCE_EXT is FRAGMENT_DEPTH_EXT, FOG_MODE is EXP, d = 1.0, e = 1.0, and s = 0.0; C_f = (0,0,0,0) and i_f=0."

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

None

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

None

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

None

Additions to Chapter 6 of the OpenGL 1.2 Specification (State and State Requests)

None

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

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

Two new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

```
 FogCoordfvEXT
     2       8           rendering command length
     2       4124        rendering command opcode
     4       FLOAT32     v[0]
```

```
    FogCoorddvEXT
        2       12          rendering command length
        2       4125        rendering command opcode
        8       FLOAT64     v[0]
```

Errors

INVALID_ENUM is generated if FogCoordPointerEXT parameter <type> is not FLOAT or DOUBLE.

INVALID_VALUE is generated if FogCoordPointerEXT parameter <stride> is negative.

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_FOG_COORDINATE_EXT	R	GetIntegerv, GetFloatv	0	Current fog coordinate	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_COORDINATE_ARRAY_EXT	B	IsEnabled	False	Fog coord array enable	2.8	vertex-array
FOG_COORDINATE_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of fog coordinate	2.8	vertex-array
FOG_COORDINATE_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between fog coords	2.8	vertex-array
FOG_COORDINATE_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the fog coord array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_COORDINATE_SOURCE_EXT	Z2	GetIntegerv, GetFloatv	FRAGMENT_DEPTH_EXT	Source of fog coordinate for fog calculation	3.10	fog

Revision History

- Revision 1.6 - Functionality complete

- Revision 1.7-1.9 - Fix typos and add fields to bring up to date with the new extension template. No functionality changes.

Name

EXT_packed_pixels

Name Strings

GL_EXT_packed_pixels

Version

$Date: 1997/09/22 23:23:58 $ $Revision: 1.21 $

Number

23

Dependencies

EXT_abgr affects the definition of this extension
EXT_texture3D affects the definition of this extension
EXT_subtexture affects the definition of this extension
EXT_histogram affects the definition of this extension
EXT_convolution affects the definition of this extension
SGI_color_table affects the definition of this extension
SGIS_texture4D affects the definition of this extension
EXT_cmyka affects the definition of this extension

Overview

This extension provides support for packed pixels in host memory. A packed pixel is represented entirely by one unsigned byte, one unsigned short, or one unsigned integer. The fields with the packed pixel are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes, including PACK_SKIP_PIXELS, PACK_ROW_LENGTH, PACK_SKIP_ROWS, PACK_IMAGE_HEIGHT_EXT, PACK_SKIP_IMAGES_EXT, PACK_SWAP_BYTES, PACK_ALIGNMENT, and their unpacking counterparts all work correctly with packed pixels.

New Procedures and Functions

None

New Tokens

Accepted by the <type> parameter of DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, GetHistogramEXT, GetMinmaxEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, GetSeparableFilterEXT, ColorTableSGI, GetColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS:

UNSIGNED_BYTE_3_3_2_EXT	0x8032
UNSIGNED_SHORT_4_4_4_4_EXT	0x8033
UNSIGNED_SHORT_5_5_5_1_EXT	0x8034
UNSIGNED_INT_8_8_8_8_EXT	0x8035
UNSIGNED_INT_10_10_10_2_EXT	0x8036

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

The five tokens defined by this extension are added to Table 3.4:

<type> Parameter Token Value	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	No
BYTE	byte	No
UNSIGNED_SHORT	ushort	No
SHORT	short	No
UNSIGNED_INT	uint	No
INT	int	No
FLOAT	float	No
BITMAP	ubyte	Yes
UNSIGNED_BYTE_3_3_2_EXT	ubyte	Yes
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	Yes
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	Yes
UNSIGNED_INT_8_8_8_8_EXT	uint	Yes
UNSIGNED_INT_10_10_10_2_EXT	uint	Yes

Table 3.4: DrawPixels and ReadPixels <type> parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.3.

[Section 3.6.3 of the GL Specification (Rasterization of Pixel Rectangles) is rewritten as follows:]

3.6.3 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in host memory is diagrammed in Figure 3.7. We describe the stages of this process in the order in which they occur.

Pixels are drawn using

```
void DrawPixels(sizei width,
            sizei height,
            enum format,
            enum type,
            void* data);
```

<format> is a symbolic constant indicating what the values in memory represent. <width> and <height> are the width and height, respectively, of the pixel rectangle to be drawn. <data> is a pointer to the data to be drawn. These data are represented with one of seven GL data types, specified by <type>. The correspondence between the thirteen <type> token values and the GL data types they indicate is given in Table 3.4. If the GL is in color index mode and <format> is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION occurs. Some additional constraints on the combinations of <format>

and <type> values that are accepted are discussed below.

Unpacking

Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating-point values (GL data type float). These elements are grouped into sets of one, two, three, four, or five values, depending on the <format>, to form a group. Table 3.5 summarizes the format of groups obtained from memory. It also indicates those formats that yield indices and those that yield components.

Format Name	Target Buffer	Element Meaning and Order
COLOR_INDEX	Color	Color index
STENCIL_INDEX	Stencil	Stencil index
DEPTH_COMPONENT	Depth	Depth component
RED	Color	R component
GREEN	Color	G component
BLUE	Color	B component
ALPHA	Color	A component
RGB	Color	R, G, B components
RGBA	Color	R, G, B, A components
ABGR_EXT	Color	A, B, G, R components
CMYK_EXT	Color	Cyan, Magenta, Yellow, Black components
CMYKA_EXT	Color	Cyan, Magenta, Yellow, Black, A components
LUMINANCE	Color	Luminance component
LUMINANCE_ALPHA	Color	Luminance, A components

Table 3.5: DrawPixels and ReadPixels formats. The third column gives a description of and the number and order of elements in a group.

By default the values of each GL data type are interpreted as they would be specified in the language of the client’s GL binding. If UNPACK_SWAP_BYTES is set to TRUE, however, then the values are interpreted with the bit orderings modified as per the table below. The modified bit orderings are defined only if the GL data type ubyte has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

Element Size	Default Bit Ordering	Modified Bit Ordering
8-bit	[7..0]	[7..0]
16-bit	[15..0]	[7..0] [15..8]
32-bit	[31..0]	[7..0] [15..8] [23..16] [31..24]

Table: Bit ordering modification of elements when UNPACK_SWAP_BYTES is TRUE. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of rows, with the first element of the first group of the first row pointed to by the pointer passed to

DrawPixels. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is <width>; otherwise the number of groups is UNPACK_ROW_LENGTH. If the first element of the first row is at location p in memory, then the location of the first element of the Nth row is

$$p + Nk$$

where N is the row number (counting from zero) and k is defined as

$$k = \begin{cases} nl & s >= a \\ a/s * \text{ceiling}(snl/a) & s < a \end{cases}$$

where n is the number of elements in a group, l is the number of groups in a row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then k = nl for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to DrawPixels is effectively advanced by

UNPACK_SKIP_PIXELS * n + UNPACK_SKIP_ROWS * k

elements. Then <width> groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. <height> sets of <width> groups of values are obtained this way. See Figure 3.8.

Calling DrawPixels with a <type> of UNSIGNED_BYTE_3_3_2, UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_INT_8_8_8_8, or UNSIGNED_INT_10_10_10_2 is a special case in which all the elements of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. The number of elements per packed pixel is fixed by the type, and must match the number of elements per group indicated by the <format> parameter. (See the table below.) The error INVALID_OPERATION is generated if a mismatch occurs.

<type> Parameter Token Name	GL Data Type	Number of Elements	Matching Pixel Formats
UNSIGNED_BYTE_3_3_2_EXT	ubyte	3	RGB
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_INT_8_8_8_8_EXT	uint	4	RGBA,ABGR_EXT,CMYK_EXT
UNSIGNED_INT_10_10_10_2_EXT	uint	4	RGBA,ABGR_EXT,CMYK_EXT

Bitfield locations of the first, second, third, and fourth elements of each packed pixel type are illustrated in the diagrams below. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed elements are right-justified in the pixel.

```
    UNSIGNED_BYTE_3_3_2_EXT:

            7   6   5   4   3   2   1   0
          +-----------+-----------+-------+
          |           |           |       |
          +-----------+-----------+-------+

             first       second     third
             element     element    element

    UNSIGNED_SHORT_4_4_4_4_EXT:

            15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
          +---------------+---------------+---------------+---------------+
          |               |               |               |               |
          +---------------+---------------+---------------+---------------+

              first           second          third           fourth
              element         element         element         element

    UNSIGNED_SHORT_5_5_5_1_EXT:

            15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
          +-------------------+-------------------+-------------------+---+
          |                   |                   |                   |   |
          +-------------------+-------------------+-------------------+---+

              first                second              third           fourth
              element              element             element         element

    UNSIGNED_INT_8_8_8_8_EXT:

 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+-----------------------+-----------------------+-----------------------+-----------------------+
|                       |                       |                       |                       |
+-----------------------+-----------------------+-----------------------+-----------------------+

         first                   second                  third                   fourth
         element                 element                 element                 element

    UNSIGNED_INT_10_10_10_2_EXT:

 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+-----------------------------+-----------------------------+-----------------------------+-----+
|                             |                             |                             |     |
+-----------------------------+-----------------------------+-----------------------------+-----+

           first                         second                        third          fourth
           element                       element                       element        element
```

The assignment of elements to fields in the packed pixel is as described in the table below:

Format	First Element	Second Element	Third Element	Fourth Element
RGB	red	green	blue	
RGBA	red	green	blue	alpha
ABGR_EXT	alpha	blue	green	red
CMYK_EXT	cyan	magenta	yellow	black

Byte swapping, if enabled, is performed before the elements are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "element" and the number of elements per group is understood to be one.

Calling DrawPixels with a <type> of BITMAP is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to DrawPixels. The first element of the second row is the first bit (again as defined above) of the ubyte at location p+k, where k is computed as

```
 k = a * ceiling(nl/8a)
```

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to DrawPixels is effectively advanced by

```
 UNPACK_SKIP_ROWS * k
```

ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent <width> 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by k ubytes. <height> sets of <width> elements are obtained this way.

Conversion to floating-point

This step applies only to groups of components. It is not performed on indices. Each element in a group is converted to a floating-point value according to the appropriate formula in Table 2.4 (section 2.12). Unsigned integer bitfields extracted from packed pixels are interpreted using the formula

```
 f = c / ((2**N)-1)
```

where c is the value of the bitfield (interpreted as an unsigned integer), N is the number of bits in the bitfield, and the division is performed in floating point.

[End of changes to Section 3.6.3]

If this extension is supported, all commands that accept pixel data also accept packed pixel data. These commands are DrawPixels, TexImage1D, TexImage2D, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, ColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

[Make the following changes to Section 4.3.2 (Reading Pixels):]

Final Conversion

For an index, if the <type> is not FLOAT, final conversion consists of masking the index with the value given in Table 4.6; if the <type> is FLOAT, then the integer index is converted to a GL float data value. For a component, each component is first clamped to [0,1]. Then, the appropriate conversion formula from Table 4.7 is applied to the component.

<type> Parameter	Index Mask
UNSIGNED_BYTE	2**8 - 1
BITMAP	1
BYTE	2**7 - 1
UNSIGNED_SHORT	2**16 - 1
SHORT	2**15 - 1
UNSIGNED_INT	2**32 - 1
INT	2**31 - 1

Table 4.6: Index masks used by ReadPixels. Floating point data are not masked.

<type> Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	c = ((2**8)-1)*f
BYTE	byte	c = (((2**8)-1)*f-1)/2
UNSIGNED_SHORT	ushort	c = ((2**16)-1)*f
SHORT	short	c = (((2**16)-1)*f-1)/2
UNSIGNED_INT	uint	c = ((2**32)-1)*f
INT	int	c = (((2**32)-1)*f-1)/2
FLOAT	float	c = f
UNSIGNED_BYTE_3_3_2_EXT	ubyte	c = ((2**N)-1)*f
UNSIGNED_SHORT_4_4_4_4_EXT	ushort	c = ((2**N)-1)*f
UNSIGNED_SHORT_5_5_5_1_EXT	ushort	c = ((2**N)-1)*f
UNSIGNED_INT_8_8_8_8_EXT	uint	c = ((2**N)-1)*f
UNSIGNED_INT_10_10_10_2_EXT	uint	c = ((2**N)-1)*f

Table 4.7: Reversed component conversions - used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation (f) to a datum of the specified GL data type (c) using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2.) Equations with N as the exponent are performed for each bitfield of the packed data type, with N set to the number of bits in the bitfield.

Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for DrawPixels. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory must where the ith group of the jth row would be taken from for DrawPixels. See Unpacking under section 3.6.3. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_.

[End of changes to Section 4.3.2]

If this extension is supported, all commands that return pixel data also return packed pixel data. These commands are ReadPixels, GetTexImage, GetHistogramEXT, GetMinmaxEXT, GetConvolutionFilterEXT, GetSeparableFilterEXT, and GetColorTableSGI.

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Dependencies on EXT_abgr

If EXT_abgr is not implemented, then the references to ABGR_EXT in this file are invalid, and should be ignored.

Dependencies on EXT_texture3D

If EXT_texture3D is not implemented, then the references to TexImage3DEXT in this file are invalid, and should be ignored.

Dependencies on EXT_subtexture

If EXT_subtexture is not implemented, then the references to TexSubImage1DEXT, TexSubImage2DEXT, and TexSubImage3DEXT in this file are invalid, and should be ignored.

Dependencies on EXT_histogram

If EXT_histogram is not implemented, then the references to GetHistogramEXT and GetMinmaxEXT in this file are invalid, and should be ignored.

Dependencies on EXT_convolution

If EXT_convolution is not implemented, then the references to ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, and GetSeparableFilterEXT in this file are invalid, and should be ignored.

Dependencies on SGI_color_table

If SGI_color_table is not implemented, then the references to ColorTableSGI and GetColorTableSGI in this file are invalid, and should be ignored.

Dependencies on SGIS_texture4D

If SGIS_texture4D is not implemented, then the references to TexImage4DSGIS and TexSubImage4DSGIS in this file are invalid, and should be ignored.

Dependencies on EXT_cmyka

If EXT_cmyka is not implemented, then the references to CMYK_EXT and CMYKA_EXT in this file are invalid, and should be ignored.

Errors

[For the purpose of this enumeration of errors, GenericPixelFunction represents any OpenGL function that accepts or returns pixel data, using parameters <type> and <format> to define the type and format of that data. Currently these functions are DrawPixels, ReadPixels, TexImage1D, TexImage2D, GetTexImage, TexImage3DEXT, TexSubImage1DEXT, TexSubImage2DEXT, TexSubImage3DEXT, GetHistogramEXT, GetMinmaxEXT, ConvolutionFilter1DEXT, ConvolutionFilter2DEXT, ConvolutionFilter3DEXT, GetConvolutionFilterEXT, SeparableFilter2DEXT, SeparableFilter3DEXT, GetSeparableFilterEXT, ColorTableSGI, GetColorTableSGI, TexImage4DSGIS, and TexSubImage4DSGIS.]

INVALID_OPERATION is generated by GenericPixelFunction if its <type> parameter is UNSIGNED_BYTE_3_3_2_EXT and its <format> parameter does not specify three components. Currently the only 3-component format is RGB.

INVALID_OPERATION is generated by GenericPixelFunction if its <type> parameter is UNSIGNED_SHORT_4_4_4_4_EXT, UNSIGNED_SHORT_5_5_5_1_EXT, UNSIGNED_INT_8_8_8_8_EXT, or UNSIGNED_INT_10_10_10_2_EXT and its <format> parameter does not specify four components. Currently the only 4-component formats are RGBA, ABGR_EXT, and CMYK_EXT.

New State

None

New Implementation Dependent State

None

Name

EXT_paletted_texture

Name Strings

GL_EXT_paletted_texture

Version

$Date: 1997/06/12 01:07:42 $ $Revision: 1.2 $

Number

78

Dependencies

GL_EXT_paletted_texture shares routines and enumerants with GL_SGI_color_table with the minor modification that EXT replaces SGI. In all other ways these calls should function in the same manner and the enumerant values should be identical. The portions of GL_SGI_color_table that are used are:

```
ColorTableSGI, GetColorTableSGI, GetColorTableParameterivSGI,
GetColorTableParameterfvSGI.
COLOR_TABLE_FORMAT_SGI, COLOR_TABLE_WIDTH_SGI,
COLOR_TABLE_RED_SIZE_SGI, COLOR_TABLE_GREEN_SIZE_SGI,
COLOR_TABLE_BLUE_SIZE_SGI, COLOR_TABLE_ALPHA_SIZE_SGI,
COLOR_TABLE_LUMINANCE_SIZE_SGI, COLOR_TABLE_INTENSITY_SIZE_SGI.
```

Portions of GL_SGI_color_table which are not used in GL_EXT_paletted_texture are:

```
CopyColorTableSGI, ColorTableParameterivSGI,
ColorTableParameterfvSGI.
COLOR_TABLE_SGI, POST_CONVOLUTION_COLOR_TABLE_SGI,
POST_COLOR_MATRIX_COLOR_TABLE_SGI, PROXY_COLOR_TABLE_SGI,
PROXY_POST_CONVOLUTION_COLOR_TABLE_SGI,
PROXY_POST_COLOR_MATRIX_COLOR_TABLE_SGI, COLOR_TABLE_SCALE_SGI,
COLOR_TABLE_BIAS_SGI.
```

EXT_paletted_texture can be used in conjunction with EXT_texture3D. EXT_paletted_texture modifies TexImage3DEXT to accept paletted image data and allows TEXTURE_3D_EXT and PROXY_TEXTURE_3D_EXT to be used a targets in the color table routines. If EXT_texture3D is unsupported then references to 3D texture support in this spec are invalid and should be ignored.

Overview

EXT_paletted_texture defines new texture formats and new calls to support the use of paletted textures in OpenGL. A paletted texture is defined by giving both a palette of colors and a set of image data which is composed of indices into the palette. The paletted texture cannot function properly without both pieces of information so it increases the work required to define a texture. This is offset by the fact that the overall amount of texture data can be reduced dramatically by factoring redundant information out of the logical view of the texture and placing it in the palette.

Paletted textures provide several advantages over full-color textures:

* As mentioned above, the amount of data required to define a texture can be greatly reduced over what would be needed for full-color specification. For example, consider a source texture that has only 256 distinct colors in a 256 by 256 pixel grid. Full-color representation requires three bytes per pixel, taking 192K of texture data. By putting the distinct colors in a palette only eight bits are required per pixel, reducing the 192K to 64K plus 768 bytes for the palette. Now add an alpha channel to the texture. The full-color representation increases by 64K while the paletted version would only increase by 256 bytes. This reduction in space required is particularly important for hardware accelerators where texture space is limited.

* Paletted textures allow easy reuse of texture data for images which require many similar but slightly different colored objects. Consider a driving simulation with heavy traffic on the road. Many of the cars will be similar but with different color schemes. If full-color textures are used a separate texture would be needed for each color scheme, while paletted textures allow the same basic index data to be reused for each car, with a different palette to change the final colors.

* Paletted textures also allow use of all the palette tricks developed for paletted displays. Simple animation can be done, along with strobing, glowing and other palette-cycling effects. All of these techniques can enhance the visual richness of a scene with very little data.

New Procedures and Functions

```
void ColorTableEXT(
 enum target,
 enum internalFormat,
 sizei width,
 enum format,
 enum type,
 const void *data);

void ColorSubTableEXT(
 enum target,
 sizei start,
 sizei count,
 enum format,
 enum type,
 const void *data);

void GetColorTableEXT(
 enum target,
 enum format,
 enum type,
 void *data);
```

```
    void GetColorTableParameterivEXT(
     enum target,
     enum pname,
     int *params);

    void GetColorTableParameterfvEXT(
     enum target,
     enum pname,
     float *params);
```

New Tokens

```
    Accepted by the internalformat parameter of TexImage1D, TexImage2D and
    TexImage3DEXT:
     COLOR_INDEX1_EXT                    0x80E2
     COLOR_INDEX2_EXT                    0x80E3
     COLOR_INDEX4_EXT                    0x80E4
     COLOR_INDEX8_EXT                    0x80E5
     COLOR_INDEX12_EXT                   0x80E6
     COLOR_INDEX16_EXT                   0x80E7

    Accepted by the pname parameter of GetColorTableParameterivEXT and
    GetColorTableParameterfvEXT:
     COLOR_TABLE_FORMAT_EXT              0x80D8
     COLOR_TABLE_WIDTH_EXT               0x80D9
     COLOR_TABLE_RED_SIZE_EXT            0x80DA
     COLOR_TABLE_GREEN_SIZE_EXT          0x80DB
     COLOR_TABLE_BLUE_SIZE_EXT           0x80DC
     COLOR_TABLE_ALPHA_SIZE_EXT          0x80DD
     COLOR_TABLE_LUMINANCE_SIZE_EXT      0x80DE
     COLOR_TABLE_INTENSITY_SIZE_EXT      0x80DF

    Accepted by the value parameter of GetTexLevelParameter{if}v:
     TEXTURE_INDEX_SIZE_EXT              0x80ED
```

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Section 3.6.4, ’Pixel Transfer Operations,’ subsection ’Color Index Lookup,’

Point two is modified from ’The groups will be loaded as an image into texture memory’ to ’The groups will be loaded as an image into texture memory and the internalformat parameter is not one of the color index formats from table 3.8.’

Section 3.8, ’Texturing,’ subsection ’Texture Image Specification’ is modified as follows:

The portion of the first paragraph discussing interpretation of format, type and data is split from the portion discussing target, width and height. The target, width and height section now ends with the sentence ’Arguments width and height specify the image’s width and height.’

The format, type and data section is moved under a subheader 'Direct Color Texture Formats' and begins with 'If internalformat is not one of the color index formats from table 3.8,' and continues with the existing text through the internalformat discussion.

After that section, a new section 'Paletted Texture Formats' has the text:

> If format is given as COLOR_INDEX then the image data is composed of integer values representing indices into a table of colors rather than colors themselves. If internalformat is given as one of the color index formats from table 3.8 then the texture will be stored internally as indices rather than undergoing index-to-RGBA mapping as would previously have occurred. In this case the only valid values for type are BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT and UNSIGNED_INT.
>
> The image data is unpacked from memory exactly as for a DrawPixels command with format of COLOR_INDEX for a context in color index mode. The data is then stored in an internal format derived from internalformat. In this case the only legal values of internalformat are COLOR_INDEX1_EXT, COLOR_INDEX2_EXT, COLOR_INDEX4_EXT, COLOR_INDEX8_EXT, COLOR_INDEX12_EXT and COLOR_INDEX16_EXT and the internal component resolution is picked according to the index resolution specified by internalformat. Any excess precision in the data is silently truncated to fit in the internal component precision.
>
> An application can determine whether a particular implementation supports a particular paletted format (or any paletted formats at all) by attempting to use the paletted format with a proxy target. TEXTURE_INDEX_SIZE_EXT will be zero if the implementation cannot support the texture as given.
>
> An application can determine an implementation's desired format for a particular paletted texture by making a TexImage call with COLOR_INDEX as the internalformat, in which case target must be a proxy target. After the call the application can query TEXTURE_INTERNAL_FORMAT to determine what internal format the implementation suggests for the texture image parameters. TEXTURE_INDEX_SIZE_EXT can be queried after such a call to determine the suggested index resolution numerically. The index resolution suggested by the implementation does not have to be as large as the input data precision. The resolution may also be zero if the implementation is unable to support any paletted format for the given texture image.

Table 3.8 should be augmented with a column titled 'Index bits.' All existing formats have zero index bits. The following formats are added with zeroes in all existing columns:

Name	Index bits
COLOR_INDEX1_EXT	1
COLOR_INDEX2_EXT	2
COLOR_INDEX4_EXT	4
COLOR_INDEX8_EXT	8
COLOR_INDEX12_EXT	12
COLOR_INDEX16_EXT	16

At the end of the discussion of level the following text should be added:

All mipmapping levels share the same palette. If levels are created with different precision indices then their internal formats will not match and the texture will be inconsistent, as discussed above.

In the discussion of internalformat for CopyTexImage{12}D, at end of the sentence specifying that 1, 2, 3 and 4 are illegal there should also be a mention that paletted internalformat values are illegal.

At the end of the width, height, format, type and data section under TexSubImage there should be an additional sentence:

If the target texture has an color index internal format then format may only be COLOR_INDEX.

At the end of the first paragraph describing TexSubImage and CopyTexSubImage the following sentence should be added:

If the target of a CopyTexSubImage is a paletted texture image then INVALID_OPERATION is returned.

After the Alternate Image Specification Commands section, a new ’Palette Specification Commands’ section should be added.

Paletted textures require palette information to translate indices into full colors. The command

```
void ColorTableEXT(enum target, enum internalformat, sizei width,
      enum format, enum type, const void *data);
```

is used to specify the format and size of the palette for paletted textures. target specifies which texture is to have its palette changed and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT or PROXY_TEXTURE_3D_EXT. internalformat specifies the desired format and resolution of the palette when in its internal form. internalformat can be any of the non-index values legal for TexImage internalformat although implementations are not required to support palettes of all possible formats. width controls the size of the palette and must be a power of two greater than or equal to one. format and type specify the number of components and type of the data given by data. format can be any of the formats legal for DrawPixels although implementations are not required to support all possible formats. type can be any of the types legal for DrawPixels except GL_BITMAP.

Data is taken from memory and converted just as if each palette entry were a single pixel of a 1D texture. Pixel unpacking and transfer modes apply just as with texture data. After unpacking and conversion the data is translated into a internal format that matches the given format as closely as possible. An implementation does not, however, have a responsibility to support more than one precision for the base formats.

If the palette’s width is greater than than the range of the color indices in the texture data then some of the palettes entries will be unused. If the palette’s width is less than the range of the color indices in the texture data then the most-significant bits of the texture data are ignored and only the appropriate number of bits of the index are used when accessing the palette.

Specifying a proxy target causes the proxy texture’s palette to be resized and its parameters set but no data is transferred or accessed. If an implementation cannot handle the palette data given in the call then the color table width and component resolutions are set to zero.

Portions of the current palette can be replaced with

```
void ColorSubTableEXT(enum target, sizei start, sizei count,
      enum format, enum type, const void *data);
```

target can be any of the non-proxy values legal for ColorTableEXT. start and count control which entries of the palette are changed out of the range allowed by the internal format used for the palette indices. count is silently clamped so that all modified entries all within the legal range. format and type can be any of the values legal for ColorTableEXT. The data is treated as a 1D texture just as in ColorTableEXT.

In the ’Texture State and Proxy State’ section the sentence fragment beginning ’six integer values describing the resolutions...’ should be changed to refer to seven integer values, with the seventh being the index resolution.

Palette data should be added in as a third category of texture state.

After the discussion of properties, the following should be added:

Next there is the texture palette. All textures have a palette, even if their internal format is not color index. A texture’s palette is initially one RGBA element with all four components set to 1.0.

The sentence mentioning that proxies do not have image data or properties should be extended with ’or palettes.’

The sentence beginning ’If the texture array is too large’ describing the effects of proxy failure should change to read:

If the implementation is unable to handle the texture image data the proxy width, height, border width and component resolutions are set to zero. This situation can occur when the texture array is too large or an unsupported paletted format was requested.

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

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

In the section on GetTexImage, the sentence saying 'The components are assigned among R, G, B and A according to' should be changed to be

> If the internal format of the texture is not a color index format then the components are assigned among R, G, B, and A according to Table 6.1. Specifying COLOR_INDEX for format in this case will generate the error INVALID_ENUM. If the internal format of the texture is color index then the components are handled in one of two ways depending on the value of format. If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette and the resulting components are assigned among R, G, B, and A according to Table 6.1. If format is COLOR_INDEX then the data is treated as single components and the palette indices are returned. Components are taken starting...

Following the GetTexImage section there should be a new section:

> GetColorTableEXT is used to get the current texture palette.

```
void GetColorTableEXT(enum target, enum format, enum type, void *data);
```

> GetColorTableEXT retrieves the texture palette of the texture given by target. target can be any of the non-proxy targets valid for ColorTableEXT. format and type are interpreted just as for ColorTableEXT. All textures have a palette by default so GetColorTableEXT will always be able to return data even if the internal format of the texture is not a color index format.
>
> Palette parameters can be retrieved using

```
void GetColorTableParameterivEXT(enum target, enum pname, int *params);
void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
```

> target specifies the texture being queried and pname controls which parameter value is returned. Data is returned in the memory pointed to by params.
>
> Querying COLOR_TABLE_FORMAT_EXT returns the internal format requested by the most recent ColorTableEXT call or the default. COLOR_TABLE_WIDTH_EXT returns the width of the current palette. COLOR_TABLE_RED_SIZE_EXT, COLOR_TABLE_GREEN_SIZE_EXT, COLOR_TABLE_BLUE_SIZE_EXT and COLOR_TABLE_ALPHA_SIZE_EXT return the actual size of the components used to store the palette data internally, not the size requested when the palette was defined.

Table 6.11, "Texture Objects" should have a line appended for TEXTURE_INDEX_SIZE_EXT:

```
TEXTURE_INDEX_SIZE_EXT  n x Z+  GetTexLevelParameter 0 xD texture image i's index resolution 3.8 -
```

Revision History

Original draft, revision 0.5, December 20, 1995 (drewb) Created

Minor revisions and clarifications, revision 0.6, January 2, 1996 (drewb)
    Replaced all request-for-comment blocks with final text
    based on implementation.

Minor revisions and clarifications, revision 0.7, Feburary 5, 1996 (drewb)
    Specified the state of the palette color information
    when existing data is replaced by new data.

    Clarified behavior of TexPalette on inconsistent textures.

Major changes due to ARB review, revision 0.8, March 1, 1996 (drewb)
    Switched from using TexPaletteEXT and GetTexPaletteEXT
    to using SGI’s ColorTableEXT routines. Added ColorSubTableEXT so
    equivalent functionality is available.

    Allowed proxies in all targets.

    Changed PALETTE?_EXT values to COLOR_INDEX?_EXT. Added
    support for one and two bit palettes. Removed PALETTE_INDEX_EXT in
    favor of COLOR_INDEX.

    Decoupled palette size from texture data type. Palette
    size is controlled only by ColorTableEXT.

Changes due to ARB review, revision 1.0, May 23, 1997 (drewb)
    Mentioned texture3D.

    Defined TEXTURE_INDEX_SIZE_EXT.

    Allowed implementations to return an index size of zero to indicate
    no support for a particular format.

    Allowed usage of GL_COLOR_INDEX as a generic format in
    proxy queries for determining an optimal index size for a particular
    texture.

    Disallowed CopyTexImage and CopyTexSubImage to paletted
    formats.

    Deleted mention of index transfer operations during GetTexImage with
    paletted formats.

Name

EXT_point_parameters

Name Strings

GL_EXT_point_parameters

Version

$Date: 1997/08/21 21:26:36 $ $Revision: 1.6 $

Number

54

Dependencies

SGIS_multisample affects the definition of this extension.

Overview

This extension supports additional geometric characteristics of points. It can be used to render particles or tiny light sources, commonly referred as "Light points".

The raster brightness of a point is a function of the point area, point color, point transparency, and the response of the display’s electron gun and phosphor. The point area and the point transparency are derived from the point size, currently provided with the <size> parameter of glPointSize.

The primary motivation is to allow the size of a point to be affected by distance attenuation. When distance attenuation has an effect, the final point size decreases as the distance of the point from the eye increases.

The secondary motivation is a mean to control the mapping from the point size to the raster point area and point transparency. This is done in order to increase the dynamic range of the raster brightness of points. In other words, the alpha component of a point may be decreased (and its transparency increased) as its area shrinks below a defined threshold.

This extension defines a derived point size to be closely related to point brightness. The brightness of a point is given by:

$$\mathrm{dist\_atten}(d) = \frac{1}{a + b * d + c * d^2}$$

$$\mathrm{brightness}(Pe) = \mathrm{Brightness} * \mathrm{dist\_atten}(|Pe|)$$

where ’Pe’ is the point in eye coordinates, and ’Brightness’ is some initial value proportional to the square of the size provided with glPointSize. Here we simplify the raster brightness to be a function of the rasterized point area and point transparency.

```
            brightness(Pe)              brightness(Pe) >= Threshold_Area
area(Pe) =
           Threshold_Area               Otherwise
```

```
factor(Pe) = brightness(Pe)/Threshold_Area
```

```
alpha(Pe) = Alpha * factor(Pe)
```

where ’Alpha’ comes with the point color (possibly modified by lighting).

’Threshold_Area’ above is in area units. Thus, it is proportional to the square of the threshold provided by the programmer through this extension.

The new point size derivation method applies to all points, while the threshold applies to multisample points only.

Issues

*   Does point alpha modification affect the current color ?

No.

*   Do we need a special function glGetPointParameterfvEXT, or get by with glGetFloat ?

No.

*   If alpha is 0, then we could toss the point before it reaches the fragment stage.

No. This can be achieved with enabling the alpha test with reference of 0 and function of LEQUAL.

*   Do we need a disable for applying the threshold ?

The default threshold value is 1.0. It is applied even if the point size is constant.

If the default threshold is not overriden, the area of multisample points with provided constant size of less than 1.0, is mapped to 1.0, while the alpha component is modulated accordingly, to compensate for the larger area. For multisample points this is not a problem, as there are no relevant applications yet. As mentioned above, the threshold does not apply to alias or antialias points.

The alternative is to have a disable of threshold application, and state that threshold (if not disabled) applies to non antialias points only (that is, alias and multisample points).

The behavior without an enable/disable looks fine.

*   Future extensions (to the extension)

1. GL_POINT_FADE_ALPHA_CLAMP_EXT

When the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, it might be desired to

```
    clamp the computed alpha to a minimum value, in order to keep the point
    visible. In this case the formula below change:

    factor = (derived_size/threshold)^2

                    factor                 clamp <= factor
    clamped_value =
                    clamp                  factor < clamp

                1.0                        derived_size >= threshold
    alpha *=
                clamped_value              Otherwise

    where clamp is defined by the GL_POINT_FADE_ALPHA_CLAMP_EXT new parameter.
```

New Procedures and Functions

```
    void glPointParameterfEXT ( GLenum pname, GLfloat param );
    void glPointParameterfvEXT ( GLenum pname, GLfloat *params );
```

New Tokens

Accepted by the <pname> parameter of glPointParameterfEXT, and the <pname> of glGet:

```
     GL_POINT_SIZE_MIN_EXT
     GL_POINT_SIZE_MAX_EXT
     GL_POINT_FADE_THRESHOLD_SIZE_EXT
```

Accepted by the <pname> parameter of glPointParameterfvEXT, and the <pname> of glGet:

```
     GL_POINT_SIZE_MIN_EXT              0x8126
     GL_POINT_SIZE_MAX_EXT              0x8127
     GL_POINT_FADE_THRESHOLD_SIZE_EXT   0x8128
     GL_DISTANCE_ATTENUATION_EXT        0x8129
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

All parameters of the glPointParameterfEXT and glPointParameterfvEXT functions set various values applied to point rendering. The derived point size is defined to be the <size> provided with glPointSize modulated with a distance attenuation factor.

The parameters GL_POINT_SIZE_MIN_EXT and GL_POINT_SIZE_MAX_EXT simply define an upper and lower bounds respectively on the derived point size.

The above parameters affect non multisample points as well as multisample points, while the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, has no effect on non multisample points. If the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_EXT parameter, the derived point size is used as the diameter of the rasterized point, and the alpha component is intact. Otherwise, the threshold size is set to be the diameter of the rasterized point, while the alpha component is modulated accordingly, to compensate for the larger area.

The distance attenuation function coefficients, namely a, b, and c in:

```
                1
dist_atten(d) = -------------------
                a + b * d + c * d^2
```

are defined by the <pname> parameter GL_DISTANCE_ATTENUATION_EXT of the function glPointParameterfvEXT. By default a = 1, b = 0, and c = 0.

Let ’size’ be the point size provided with glPointSize, let ’dist’ be the distance of the point from the eye, and let ’threshold’ be the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE parameter of glPointParameterfEXT. The derived point size is given by:

```
derived_size = size * sqrt(dist_atten(dist))
```

Note that when default values are used, the above formula reduces to:

```
derived_size = size
```

the diameter of the rasterized point is given by:

```
           derived_size             derived_size >= threshold
diameter =
           threshold                Otherwise
```

The alpha of a point is calculated to allow the fading of points instead of shrinking them past a defined threshold size. The alpha component of the rasterized point is given by:

```
            1                              derived_size >= threshold
alpha *=
            (derived_size/threshold)^2     Otherwise
```

The threshold defined by GL_POINT_FADE_THRESHOLD_SIZE_EXT is not clamped to the minimum and maximum point sizes.

Points do not affect the current color.

This extension doesn’t change the feedback or selection behavior of points.

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_multisample

If SGIS_multisample is not implemented, then the references to multisample points are invalid, and should be ignored.

Errors

INVALID_ENUM is generated if PointParameterfEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, or GL_POINT_FADE_THRESHOLD_SIZE_EXT.

INVALID_ENUM is generated if PointParameterfvEXT parameter <pname> is not GL_POINT_SIZE_MIN_EXT, GL_POINT_SIZE_MAX_EXT, GL_POINT_FADE_THRESHOLD_SIZE_EXT, or GL_DISTANCE_ATTENUATION_EXT

INVALID_VALUE is generated when values are out of range according to:

<pname>	valid range
GL_POINT_SIZE_MIN_EXT	>= 0
GL_POINT_SIZE_MAX_EXT	>= 0
GL_POINT_FADE_THRESHOLD_SIZE_EXT	>= 0

Issues

- should we generate INVALID_VALUE or just clamp?

New State

Get Value	Get Command	Type	Initial Value	Attribute
GL_POINT_SIZE_MIN_EXT	GetFloatv	R	0	point
GL_POINT_SIZE_MAX_EXT	GetFloatv	R	M	point
GL_POINT_FADE_THRESHOLD_SIZE_EXT	GetFloatv	R	1	point
GL_DISTANCE_ATTENUATION_EXT	GetFloatv	3xR	(1,0,0)	point

M is the largest available point size.

New Implementation Dependent State

None

Backwards Compatibility

This extension replaces SGIS_point_parameters. The procedures, tokens, and name strings now refer to EXT instead of SGIS. Enumerant values are unchanged. SGI implementations which previously provided this functionality should support both forms of the extension.

Name

EXT_rescale_normal

Name Strings

GL_EXT_rescale_normal

Version

$Date: 1997/07/02 23:38:17 $ $Revision: 1.7 $

Number

27

Dependencies

None

Overview

When normal rescaling is enabled a new operation is added to the transformation of the normal vector into eye coordinates. The normal vector is rescaled after it is multiplied by the inverse modelview matrix and before it is normalized.

The rescale factor is chosen so that in many cases normal vectors with unit length in object coordinates will not need to be normalized as they are transformed into eye coordinates.

New Procedures and Functions

None

New Tokens

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

RESCALE_NORMAL_EXT	0x803A

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

Section 2.10.3

Finally, we consider how the ModelView transformation state affects normals. Normals are of interest only in eye coordinates, so the rules governing their transformation to other coordinate systems are not examined.

Normals which have unit length when sent to the GL, have their length changed by the inverse of the scaling factor after transformation by the model-view inverse matrix when the model-view matrix represents a uniform scale. If rescaling is enabled, then normals specified with the Normal command are rescaled after transformation by the ModelView Inverse.

Normals sent to the GL may or may not have unit length. In addition, the length of the normals after transformation might be altered due to transformation by the model-view inverse matrix. If normalization is enabled, then normals specified with the Normal3 command are normalized after transformation by the model-view inverse matrix and after rescaling if rescaling is enabled. Normalization and rescaling are controlled with

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

and

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

with target equal to NORMALIZE or RESCALE_NORMAL. This requires two bits of state. The initial state is for normals not to be normalized or rescaled.

.

.

.

Therefore, if the modelview matrix is M, then the transformed plane equation is

$$(n_x' \; n_y' \; n_z' \; q') = ((n_x \; n_y \; n_z \; q) * (M^{-1})),$$

the rescaled normal is

$$(n_x'' \; n_y'' \; n_z'') = f * (n_x' \; n_y' \; n_z'),$$

and the fully transformed normal is

$$\frac{1}{\sqrt{(n_x'')^2 + (n_y'')^2 + (n_z'')^2}} \begin{pmatrix} n_x'' \\ n_y'' \\ n_z'' \end{pmatrix} \tag{2.1}$$

If rescaling is disabled then f is 1, otherwise f is computed as follows:

Let m_ij denote the matrix element in row i and column j of M^-1, numbering the topmost row of the matrix as row 1, and the leftmost column as column 1. Then

$$f = \frac{1}{\sqrt{(m_{31})^2 + (m_{32})^2 + (m_{33})^2}}$$

Alternatively, an implementation my chose to normalize the normal instead of rescaling the normal. Then

$$f = \frac{1}{\sqrt{(n\_x')^2 + (n\_y')^2 + (n\_z')^2}}$$

If normalization is disabled, then the square root in equation 2.1 is replaced with 1, otherwise . . . .

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

New State

Get Value	Get Command	Type	Initial Value	Attribute
RESCALE_NORMAL_EXT	IsEnabled	B	FALSE	transform/enable

New Implementation Dependent State

None

Name

EXT_secondary_color

Name Strings

GL_EXT_secondary_color

Version

NVIDIA Date: February 22, 2000
$Date: 1999/06/21 19:57:47 $ $Revision: 1.8 $

Number

145

Dependencies

Either EXT_separate_specular_color or OpenGL 1.2 is required, to specify the "Color Sum" stage and other handling of the secondary color. This is written against the 1.2 specification (available from www.opengl.org).

Overview

This extension allows specifying the RGB components of the secondary color used in the Color Sum stage, instead of using the default (0,0,0,0) color. It applies only in RGBA mode and when LIGHTING is disabled.

Issues

* Can we use the secondary alpha as an explicit fog weighting factor?

  ISVs prefer a separate interface (see GL_EXT_fog_coord). The current interface specifies only the RGB elements, leaving the option of a separate extension for SecondaryColor4() entry points open (thus the apparently useless ARRAY_SIZE state entry).

  There is an unpleasant asymmetry with Color3() - one assumes A = 1.0, the other assumes A = 0.0 - but this appears unavoidable given the 1.2 color sum specification language. Alternatively, the color sum language could be rewritten to not sum secondary A.

* What about multiple "color iterators" for use with aggrandized multitexture implementations?

  We may need this eventually, but the secondary color is well defined and a more generic interface doesn’t seem justified now.

* Interleaved array formats?

  No. The multiplicative explosion of formats is too great.

* Do we want to be able to query the secondary color value? How does it interact with lighting?

The secondary color is not part of the GL state in the separate_specular_color extension that went into OpenGL 1.2. There, it can't be queried or obtained via feedback.

The secondary_color extension is slightly more general-purpose, so the secondary color is explicitly in the GL state and can be queried - but it's still somewhat limited and can't be obtained via feedback, for example.

New Procedures and Functions

```
    void SecondaryColor3[bsifd ubusui]EXT(T components)
    void SecondaryColor3[bsifd ubusui]vEXT(T components)
    void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                                  void *pointer)
```

New Tokens

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

```
    COLOR_SUM_EXT                              0x8458
```

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

```
    CURRENT_SECONDARY_COLOR_EXT                0x8459
    SECONDARY_COLOR_ARRAY_SIZE_EXT             0x845A
    SECONDARY_COLOR_ARRAY_TYPE_EXT             0x845B
    SECONDARY_COLOR_ARRAY_STRIDE_EXT           0x845C
```

Accepted by the <pname> parameter of GetPointerv:

```
    SECONDARY_COLOR_ARRAY_POINTER_EXT          0x845D
```

Accepted by the <array> parameter of EnableClientState and DisableClientState:

```
    SECONDARY_COLOR_ARRAY_EXT                  0x845E
```

Additions to Chapter 2 of the 1.2 Draft Specification (OpenGL Operation)

These changes describe a new current state type, the secondary color, and the commands to specify it:

- (2.6, p. 12) Second paragraph changed to:

  "Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current secondary color may be used in processing each vertex."

  Third paragraph, second sentence changed to:

  "These associated colors are either based on the current color and current secondary color, or produced by lighting, depending on whether or not lighting is enabled."

- 2.6.3, p. 19) First paragraph changed to

  "The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, and texture coordinates (Vertex, Color, SecondaryColorEXT, Index, Normal, TexCoord)..."

- (2.7, p. 20) Starting with the fourth paragraph, change to:

  "Finally, there are several ways to set the current color and secondary color. The GL stores a current single-valued color index as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in color index mode or RGBA mode. The mode selection is made when the GL is initialized.

  The commands to set RGBA colors and secondary colors are:

  ```
  void Color[34][bsifd ubusui](T components)
  void Color[34][bsifd ubusui]v(T components)
  void SecondaryColor3[bsifd ubusui]EXT(T components)
  void SecondaryColor3[bsifd ubusui]vEXT(T components)
  ```

  The color command has two major variants: Color3 and Color4. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.13.)

  The secondary color command has only the three value versions. Secondary A is always set to 0.0.

  Versions of the Color and SecondaryColorEXT commands that take floating-point values accept values nominally between 0.0 and 1.0...."

  The last paragraph is changed to read:

  "The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial current normal has coordinates (0,0,1). The initial RGBA color is (R,G,B,A) = (1,1,1,1). The initial RGBA secondary color is (R,G,B,A) = (0,0,0,0). The initial color index is 1."

- (2.8, p. 21) Added secondary color command for vertex arrays:

  Change first paragraph to read:

  "The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client’s address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to seven arrays: one each to store edge flags, texture coordinates, colors, secondary colors, color indices, normals, and vertices. The commands"

Add to functions listed following first paragraph:

```
void SecondaryColorPointerEXT(int size, enum type, sizei stride,
                              void *pointer)
```

Add to table 2.4 (p. 22):

Command	Sizes	Types
SecondaryColorPointerEXT	3	byte,ubyte,short,ushort, int,uint,float,double

Starting with the second paragraph on p. 23, change to add SECONDARY_COLOR_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
void EnableClientState(enum array)
void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY, SECONDARY_COLOR_ARRAY_EXT, INDEX_ARRAY, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, color, secondary color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, color, secondary color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, Color<size><type>v, SecondaryColor3<type>vEXT, Index<type>v, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable secondary color array for canned interleaved array formats. After the lines

```
DisableClientState(EDGE_FLAG_ARRAY);
DisableClientState(INDEX_ARRAY);
```

insert the line

```
     DisableClientState(SECONDARY_COLOR_ARRAY_EXT);
```

Substitute "seven" for every occurence of "six" in the final paragraph on p. 27.

- (2.12, p. 41) Add secondary color to the current rasterpos state.

  Change the last paragraph to read

  "The current raster position requires five single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA color, RGBA secondary color, and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both $(0,0,0,1)$, the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is $(1,1,1,1)$, the associated RGBA secondary color is $(0,0,0,0)$, and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color and and secondary color always maintain their initial values."

- (2.13, p. 43) Change second paragraph to acknowledge two colors when lighting is disabled:

  "Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or current color (primary color) and current secondary color are used in further processing. After lighting, RGBA colors are clamped..."

- (Figure 2.8, p. 42) Change to show primary and secondary RGBA colors in both lit and unlit paths.

- (2.13.1, p. 44) Change so that the second paragraph starts:

  "Lighting may be in one of two states:

  1. Lighting Off. In this state, the current color and current secondary color are assigned to the vertex primary color and vertex secondary color, respectively.

  2. ..."

- (2.13.1, p. 48) Change the sentence following equation 2.5 (for spot_i) so that color sum is implicitly enabled when SEPARATE_SPECULAR_COLOR is set:

  "All computations are carried out in eye coordinates. When c_es = SEPARATE_SPECULAR_COLOR, it is as if color sum (see section 3.9) were enabled, regardless of the value of COLOR_SUM_EXT."

- (3.9, p. 136) Change the first paragraph to read

"After texturing, a fragment has two RGBA colors: a primary color c_pri (which texturing, if enabled, may have modified) and a secondary color c_sec.

If color sum is enabled, the components of these two colors are summed to produce a single post-texturing RGBA color c (the A component of the secondary color is always 0). The components of c are then clamped to the range [0,1]. If color sum is disabled, then c_pri is assigned to the post texturing color. Color sum is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant COLOR_SUM_EXT.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled."

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

GLX Protocol

Eight new GL rendering commands are added. The following commands are sent to the server as part of a glXRender request:

```
SecondaryColor3bvEXT
    2       8           rendering command length
    2       4126        rendering command opcode
    1       INT8        v[0]
    1       INT8        v[1]
    1       INT8        v[2]
    1                   unused

SecondaryColor3svEXT
    2       12          rendering command length
    2       4127        rendering command opcode
    2       INT16       v[0]
    2       INT16       v[1]
    2       INT16       v[2]
    2                   unused

SecondaryColor3ivEXT
    2       16          rendering command length
    2       4128        rendering command opcode
    4       INT32       v[0]
    4       INT32       v[1]
    4       INT32       v[2]

SecondaryColor3fvEXT
    2       16          rendering command length
    2       4129        rendering command opcode
    4       FLOAT32     v[0]
    4       FLOAT32     v[1]
    4       FLOAT32     v[2]
```

```
SecondaryColor3dvEXT
    2       28          rendering command length
    2       4130        rendering command opcode
    8       FLOAT64     v[0]
    8       FLOAT64     v[1]
    8       FLOAT64     v[2]

SecondaryColor3ubvEXT
    2       8           rendering command length
    2       4131        rendering command opcode
    1       CARD8       v[0]
    1       CARD8       v[1]
    1       CARD8       v[2]
    1                   unused

SecondaryColor3usvEXT
    2       12          rendering command length
    2       4132        rendering command opcode
    2       CARD16      v[0]
    2       CARD16      v[1]
    2       CARD16      v[2]
    2                   unused

SecondaryColor3uivEXT
    2       16          rendering command length
    2       4133        rendering command opcode
    4       CARD32      v[0]
    4       CARD32      v[1]
    4       CARD32      v[2]
```

Errors

INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <size> is not 3.

INVALID_ENUM is generated if SecondaryColorPointerEXT parameter <type> is not BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE.

INVALID_VALUE is generated if SecondaryColorPointerEXT parameter <stride> is negative.

New State

(table 6.5, p. 195)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_SECONDARY_COLOR_EXT	C	GetIntegerv, GetFloatv	(0,0,0,0)	Current secondary color	2.7	current

(table 6.6, p. 197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
SECONDARY_COLOR_ARRAY_EXT	B	IsEnabled	False	Sec. color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE_EXT	Z+	GetIntegerv	3	Sec. colors per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE_EXT	Z8	GetIntegerv	FLOAT	Type of sec. color components	2.8	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE_EXT	Z+	GetIntegerv	0	Stride between sec. colors	2.8	vertex-array
SECONDARY_COLOR_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to the sec. color array	2.8	vertex-array

(table 6.8, p. 198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
COLOR_SUM_EXT	B	IsEnabled	False	True if color sum enabled	3.9	fog/enable

Name

EXT_separate_specular_color

Name Strings

GL_EXT_separate_specular_color

Version

$Date: 1997/10/05 00:16:23 $ $Revision: 1.3 $

Number

144

Dependencies

None

Overview

This extension adds a second color to rasterization when lighting is enabled. Its purpose is to produce textured objects with specular highlights which are the color of the lights. It applies only to rgba lighting.

The two colors are computed at the vertexes. They are both clamped, flat-shaded, clipped, and converted to fixed-point just like the current rgba color (see Figure 2.8). Rasterization interpolates both colors to fragments. If texture is enabled, the first (or primary) color is the input to the texture environment; the fragment color is the sum of the second color and the color resulting from texture application. If texture is not enabled, the fragment color is the sum of the two colors.

A new control to LightModel*, LIGHT_MODEL_COLOR_CONTROL_EXT, manages the values of the two colors. It takes values: SINGLE_COLOR_EXT, a compatibility mode, and SEPARATE_SPECULAR_COLOR_EXT, the object of this extension. In single color mode, the primary color is the current final color and the secondary color is 0.0. In separate specular mode, the primary color is the sum of the ambient, diffuse, and emissive terms of final color and the secondary color is the specular term.

There is much concern that this extension may not be compatible with the future direction of OpenGL with regards to better lighting and shading models. Until those impacts are resolved, serious consideration should be given before adding to the interface specified herein (for example, allowing the user to specify a second input color).

Issues

* Where is emissive included?

RESOLVED - Emissive is included with the ambient and diffuse terms. Grouping emissive with specular (the "proper" thing) could be implemented with a new value for the color control.

* Should there be two colors when not lighting or with index lighting?

  RESOLVED - The answer is probably yes--there should be two colors when lighting is disabled and there could be an incorporation of two colors with index lighting; but these are beyond the scope of this extension. Further, attempts to accomplish these may not be compatible with the future direction of OpenGL with respect to high quality lighting and shading models.

* What happens when texture is disabled?

  RESOLVED - The extension specifies to add the two colors when texture is disabled. This is compatible with the philosophy of "if texture is disabled, this mode does not apply".

New Procedures and Functions

None.

New Tokens

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

```
LIGHT_MODEL_COLOR_CONTROL_EXT        0x81F8
```

Accepted by the <param> parameter of LightModel* when <pname> is LIGHT_MODEL_COLOR_CONTROL_EXT:

```
SINGLE_COLOR_EXT                     0x81F9
SEPARATE_SPECULAR_COLOR_EXT          0x81FA
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

- (2.13, p. 40) Rework the second paragraph to acknowledge two colors:

  "Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or color is used in further processing (the current color is the primary color and the secondary color is 0). After lighting, colors are clamped..."

- (Figure 2.8, p. 41) Change RGBA to primary RGBA and secondary RGB:

  Ideally, there might be an RGB2 underneath RGBA (both places). Alternatively, a note in the caption could clarify that RGBA referred to the primary RGBA and a secondary RGB. (Speaking of the caption, the part about "m is the number of bits an R, G, B, or A component" could be removed as m doesn’t appear in the diagram.)

- (2.13.1, p. 42) Rework the opening of this section to not imply a single color:

  In the first sentence, change "a color" to "colors". Rephrase the itemization of the two lighting states to:

  "1. Lighting Off. In this state, the current color is assigned to the vertex primary color. The vertex secondary color is 0.

  2. Lighting On. In this state, the vertex primary and secondary colors are computed from the current lighting parameters."

- (Table 2.7, p.44) Add new entry (at the bottom):

Parameter	Type	Default Value	Description
c_es	enum	SINGLE_COLOR_EXT	controls computation of colors

- (p. 45, top of page) Rephrase the first line and equation:

  "Lighting produces two colors at a vertex: a primary color c_1 and a secondary color c_2. The values of c_1 and c_2 depend on the light model color control, c_es (note: c_es should be in italics and c_1 and c_2 in bold, so this really won’t be as confusing as it seems). If c_es = SINGLE_COLOR_EXT, then the equations to compute c_1 and c_2 are (note: the equation for c_1 is the current equation for c):

```
  c_1 = e_cm
      + a_cm * a_cs
      + SUM(att_i * spot_i * (a_cm * a_cli
                            + dot(n, VP_pli) * d_cm * d_cli
                            + f_i * dot(n, h_i)^s_rm * s_cm * s_cli)
  c_2 = 0
```

  If c_es = SEPARATE_SPECULAR_COLOR_EXT, then:

```
  c_1 = e_cm
      + a_cm * a_cs
      + SUM (att_i * spot_i * (a_cm * a_cli
                             + (n dot VP_pli) * d_cm * d_cli)

  c_2 = SUM(att_i * spot_i * (f_i * (n dot h_i)^s_rm * s_cm * s_cli)
```

- (p. 45, second paragraph from bottom) Clarify that A is in the primary color:

  After the sentence "The value of A produced by lighting is the alpha value associated with d_cm", add "A is always associated with the primary color c_1; c_2 has no alpha component."

- (Table 2.8, p. 48) Add a new entry (at the bottom):

Parameter	Name	Number of values
c_es	LIGHT_MODEL_COLOR_CONTROL_EXT	1

- (2.13.6, p. 51) Clarify that both primary and secondary colors are clamped:

  Replace "RGBA" in the first line of the section with "both primary and secondary".

- (2.13.7, p. 52) Clarify what happens to primary and secondary colors when flat shading--reword the first paragraph:

  "A primitive may be flatshaded, meaning that all vertices of the primitive are assigned the same color index or primary and secondary colors. These come from the vertex that spawned the primitive. For a point, these are the colors associated with the point. For a line segment, they are the colors of the second (final) vertex of the segment. For a polygon, they come from a selected vertex depending on how the polygon was generated. Table 2.9 summarizes the possibilities."

- (2.13.8, p. 52) Rework to not imply a single color:

  In the second sentence, change "If the color is" to "Those" and ", it is" to "are". In the first sentence of the next paragraph, change "the color" to "two colors".

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

- (Figure 3.1, p. 55) Add a box between texturing and fog called "color sum".

- (3.8, p. 85) In the first paragraph, second sentence, insert "primary" before RGBA. Insert after this sentence "Texturing does not affect the secondary color."

- (new section before 3.9) Insert new section titled "Color Sum":

  "At the beginning of this stage in RGBA mode, a fragment has two colors: a primary RGBA color (which texture, if enabled, may have modified) and a secondary RGB color. This stage sums the R, G, and B components of these two colors to produce a single RGBA color. If the resulting RGB values exceed 1.0, they are clamped to 1.0.

  In color index mode, a fragment only has a single color index and this stage does nothing."

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None.

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

- (5.3, p. 137) Specify that feedback returns the primary color by changing the last sentence of the large paragraph in the middle of the page to:

"The colors returned are the primary colors. These colors and the texture coordinates are those resulting from the clipping operations as described in section 2.13.8."

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

- (Table 6.9, p. 157) Add:

Get Value - LIGHT_MODEL_COLOR_CONTROL_EXT
Type - Z2
Get Cmnd - GetIntegerv
Initial Value - SINGLE_COLOR_EXT
Description - color control
Sec. - (whatever it ends up as)
Attribute - lighting

Additions to the GLX Specification

None.

GLX Protocol

None.

Errors

None.

New State

(see changes to table 6.9)

Name

EXT_shared_texture_palette

Name Strings

GL_EXT_shared_texture_palette

Version

$Date: 1997/09/10 23:23:04 $ $Revision: 1.2 $

Number

141

Dependencies

EXT_paletted_texture is required.

Overview

EXT_shared_texture_palette defines a shared texture palette which may be used in place of the texture object palettes provided by EXT_paletted_texture. This is useful for rapidly changing a palette common to many textures, rather than having to reload the new palette for each texture. The extension acts as a switch, causing all lookups that would normally be done on the texture’s palette to instead use the shared palette.

Issues

- Do we want to use a new <target> to ColorTable to specify the shared palette, or can we just infer the new target from the corresponding Enable?

- A future extension of larger scope might define a "texture palette object" and bind these objects to texture objects dynamically, rather than making palettes part of the texture object state as the current EXT_paletted_texture spec does.

- Should there be separate shared palettes for 1D, 2D, and 3D textures?

  Probably not; palette lookups have nothing to do with the dimensionality of the texture. If multiple shared palettes are needed, we should define palette objects.

- There’s no proxy mechanism for checking if a shared palette can be defined with the requested parameters. Will it suffice to assume that if a texture palette can be defined, so can a shared palette with the same parameters?

- The changes to the spec are based on changes already made for EXT_paletted_texture, which means that all three documents must be referred to. This is quite difficult to read.

- The changes to section 3.8.6, defining how shared palettes are enabled and disabled, might be better placed in section 3.8.1. However, the underlying EXT_paletted_texture does not appear to modify these sections to define exactly how palette lookups are done, and it’s not clear where to put the changes.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetBooleanv, GetIntegerv, GetFloatv, GetDoublev, IsEnabled, Enable, Disable, ColorTableEXT, ColorSubTableEXT, GetColorTableEXT, GetColorTableParameterivEXT, and GetColorTableParameterfd EXT:

```
SHARED_TEXTURE_PALETTE_EXT              0x81FB
```

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.1 Specification (Rasterization)

Section 3.8, ’Texturing,’ subsection ’Texture Image Specification’ is modified as follows:

In the Palette Specification Commands section, the sentence beginning ’target specifies which texture is to’ should be changed to:

> target specifies the texture palette or shared palette to be changed, and may be one of TEXTURE_1D, TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, TEXTURE_3D_EXT, PROXY_TEXTURE_3D_EXT, or SHARED_TEXTURE_PALETTE_EXT.

In the ’Texture State and Proxy State’ section, the sentence beginning ’A texture’s palette is initially...’ should be changed to:

> There is also a shared palette not associated with any texture, which may override a texture palette. All palettes are initially...

Section 3.8.6, ’Texture Application’ is modified by appending the following:

Use of the shared texture palette is enabled or disabled using the generic Enable or Disable commands, respectively, with the symbolic constant SHARED_TEXTURE_PALETTE_EXT.

The required state is one bit indicating whether the shared palette is enabled or disabled. In the initial state, the shared palettes is disabled.

Additions to Chapter 4 of the 1.1 Specification (Per-Fragment Operations and the Frame buffer)

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

Additions to Chapter 6 of the 1.1 Specification (State and State Requests)

In the section on GetTexImage, the sentence beginning 'If format is not COLOR_INDEX...' should be changed to:

> If format is not COLOR_INDEX, the texture's indices are passed through the texture's palette, or the shared palette if one is enabled, and the resulting components are assigned among R, G, B, and A according to Table 6.1.

In the GetColorTable section, the first sentence of the second paragraph should be changed to read:

> GetColorTableEXT retrieves the texture palette or shared palette given by target.

The first sentence of the third paragraph should be changed to read:

> Palette parameters can be retrieved using

```
void GetColorTableParameterivEXT(enum target, enum pname, int *params);
void GetColorTableParameterfvEXT(enum target, enum pname, float *params);
```

> target specifies the texture palette or shared palette being queried and pname controls which parameter value is returned.

Additions to the GLX Specification

None

New State

Get Value	Get Command	Type	Initial Value	Attribute
SHARED_TEXTURE_PALETTE_EXT	IsEnabled	B	False	texture/enable

New Implementation Dependent State

None

Name

EXT_stencil_wrap

Name Strings

GL_EXT_stencil_wrap

Version

Date: 11/15/1999 Version 1.2

Number

176

Dependencies

None

Overview

Various algorithms use the stencil buffer to "count" the number of surfaces that a ray passes through. As the ray passes into an object, the stencil buffer is incremented. As the ray passes out of an object, the stencil buffer is decremented.

GL requires that the stencil increment operation clamps to its maximum value. For algorithms that depend on the difference between the sum of the increments and the sum of the decrements, clamping causes an erroneous result.

This extension provides an enable for both maximum and minimum wrapping of stencil values. Instead, the stencil value wraps in both directions.

Two additional stencil operations are specified. These new operations are similiar to the existing INCR and DECR operations, but they wrap their result instead of saturating it. This functionality matches the new stencil operations introduced by DirectX 6.

New Procedures and Functions

None

New Tokens

Accepted by the <mode> parameter of BlendEquation:

```
INCR_WRAP_EXT             0x8507
DECR_WRAP_EXT             0x8508
```

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

Section 4.1.4 "Stencil Test" (page 144), change the 3rd paragraph to read:

"... The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP_EXT, and DECR_WRAP_EXT. The correspond to keeping the current value, setting it to zero, replacing it with the reference value, incrementing it with saturation, decrementing it with saturation, bitwise inverting it, incrementing it without saturation, and decrementing it without saturation. For purposes of incrementing and decrementing, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation will clamp values at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0 and decrementing 0 results in the maximum representable value. ..."

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated by StencilOp if any of its parameters are not KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP_EXT, or DECR_WRAP_EXT.

New State

(table 6.15, page 205)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
STENCIL_FAIL	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_PASS	Z8	GetIntegerv	KEEP	4.1.4	stencil-buffer

NOTE: the only change is that Z6 type changes to Z8

New Implementation Dependent State

None

Name

EXT_texture_compression_s3tc

Name Strings

GL_EXT_texture_compression_s3tc

Contact

Pat Brown, Intel Corporation (patrick.r.brown ’at’ intel.com)

Status

FINAL

Version

1.0, 7 July 2000

Number

198

Dependencies

OpenGL 1.1 is required.

GL_ARB_texture_compression is required.

This extension is written against the OpenGL 1.2.1 Specification.

Overview

This extension provides additional texture compression functionality specific to S3’s S3TC format (called DXTC in Microsoft’s DirectX API), subject to all the requirements and limitations described by the extension GL_ARB_texture_compression.

This extension supports DXT1, DXT3, and DXT5 texture compression formats. For the DXT1 image format, this specification supports an RGB-only mode and a special RGBA mode with single-bit "transparent" alpha.

IP Status

Contact S3 Incorporated (http://www.s3.com) regarding any intellectual property issues associated with implementing this extension.

WARNING: Vendors able to support S3TC texture compression in Direct3D drivers do not necessarily have the right to use the same functionality in OpenGL.

Issues

(1) Should DXT2 and DXT4 (premultiplied alpha) formats be supported?

RESOLVED: No -- insufficient interest. Supporting DXT2 and DXT4 would require some rework to the TexEnv definition (maybe add a new base internal format RGBA_PREMULTIPLIED_ALPHA) for these formats. Note that the EXT_texture_env_combine extension (which extends normal TexEnv modes) can be used to support textures with premultipled alpha.

(2) Should generic "RGB_S3TC_EXT" and "RGBA_S3TC_EXT" enums be supported or should we use only the DXT<n> enums?

RESOLVED: No. A generic RGBA_S3TC_EXT is problematic because DXT3 and DXT5 are both nominally RGBA (and DXT1 with the 1-bit alpha is also) yet one format must be chosen up front.

(3) Should TexSubImage support all block-aligned edits or just the minimal functionality required by the the ARB_texture_compression extension?

RESOLVED: Allow all valid block-aligned edits.

(4) A pre-compressed image with a DXT1 format can be used as either an RGB_S3TC_DXT1 or an RGBA_S3TC_DXT1 image. If the image has transparent texels, how are they treated in each format?

RESOLVED: The renderer has to make sure that an RGB_S3TC_DXT1 format is decoded as RGB (where alpha is effectively one for all texels), while RGBA_S3TC_DXT1 is decoded as RGBA (where alpha is zero for all texels with "transparent" encodings). Otherwise, the formats are identical.

(5) Is the encoding of the RGB components for DXT1 formats correct in this spec? MSDN documentation does not specify an RGB color for the "transparent" encoding. Is it really black?

RESOLVED: Yes. The specification for the DXT1 format initially required black, but later changed that requirement to a recommendation. All vendors involved in the definition of this specification support black. In addition, specifying black has a useful behavior.

When blending multiple texels (GL_LINEAR filtering), mixing opaque and transparent samples is problematic. Defining a black color on transparent texels achieves a sensible result that works like a texture with premultiplied alpha. For example, if three opaque white and one transparent sample is being averaged, the result would be a 75% intensity gray (with an alpha of 75%). This is the same result on the color channels as would be obtained using a white color, 75% alpha, and a SRC_ALPHA blend factor.

(6) Is the encoding of the RGB components for DXT3 and DXT5 formats correct in this spec? MSDN documentation suggests that the RGB blocks for DXT3 and DXT5 are decoded as described the the DXT1 format.

RESOLVED: Yes -- this appears to be a bug in the MSDN documentation. The specification for the DXT2-DXT5 formats require decoding using the opaque block encoding, regardless of the relative values of "color0" and "color1".

New Procedures and Functions

None.

New Tokens

Accepted by the <internalformat> parameter of TexImage2D, CopyTexImage2D, and CompressedTexImage2DARB and the <format> parameter of CompressedTexSubImage2DARB:

COMPRESSED_RGB_S3TC_DXT1_EXT	0x83F0
COMPRESSED_RGBA_S3TC_DXT1_EXT	0x83F1
COMPRESSED_RGBA_S3TC_DXT3_EXT	0x83F2

```
        COMPRESSED_RGBA_S3TC_DXT5_EXT                 0x83F3
```

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

None.

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

Add to Table 3.16.1: Specific Compressed Internal Formats

Compressed Internal Format	Base Internal Format
COMPRESSED_RGB_S3TC_DXT1_EXT	RGB
COMPRESSED_RGBA_S3TC_DXT1_EXT	RGBA
COMPRESSED_RGBA_S3TC_DXT3_EXT	RGBA
COMPRESSED_RGBA_S3TC_DXT5_EXT	RGBA

Modify **Section 3.8.2, Alternate Image Specification**

(add to end of TexSubImage discussion, p.123 -- after edit from the ARB_texture_compression spec)

If the internal format of the texture image being modified is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of the several S3TC compressed texture image formats. Such images are easily edited along 4x4 texel boundaries, so the limitations on TexSubImage2D or CopyTexSubImage2D parameters are relaxed. TexSubImage2D and CopyTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

* <width> is not a multiple of four or equal to TEXTURE_WIDTH, unless <xoffset> and <yoffset> are both zero.
* <height> is not a multiple of four or equal to TEXTURE_HEIGHT, unless <xoffset> and <yoffset> are both zero.
* <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an S3TC compressed texture image that does not intersect the area being modifed are preserved during valid TexSubImage2D and CopyTexSubImage2D calls.

Add to Section 3.8.2, Alternate Image Specification (adding to the end of the CompressedTexImage section introduced by the ARB_texture_compression spec)

If <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the compressed texture is stored using one of several S3TC compressed texture image formats. The S3TC texture compression algorithm supports only 2D images without borders. CompressedTexImage1DARB and CompressedTexImage3DARB produce an INVALID_ENUM error if <internalformat> is an S3TC format. CompressedTexImage2DARB will produce an INVALID_OPERATION error if <border> is non-zero.

Add to Section 3.8.2, Alternate Image Specification (adding to the end of the CompressedTexSubImage section introduced by the ARB_texture_compression spec)

If the internal format of the texture image being modified is

COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT, the texture is stored using one of the several S3TC compressed texture image formats. Since the S3TC texture compression algorithm supports only 2D images, CompressedTexSubImage1DARB and CompressedTexSubImage3DARB produce an INVALID_ENUM error if <format> is an S3TC format. Since S3TC images are easily edited along 4x4 texel boundaries, the limitations on CompressedTexSubImage2D are relaxed. CompressedTexSubImage2D will result in an INVALID_OPERATION error only if one of the following conditions occurs:

- <width> is not a multiple of four or equal to TEXTURE_WIDTH.
- <height> is not a multiple of four or equal to TEXTURE_HEIGHT.
- <xoffset> or <yoffset> is not a multiple of four.

The contents of any 4x4 block of texels of an S3TC compressed texture image that does not intersect the area being modifed are preserved during valid TexSubImage2D and CopyTexSubImage2D calls.

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

None.

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

None.

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

None.

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

None.

Additions to the AGL/GLX/WGL Specifications

None.

GLX Protocol

None.

Errors

INVALID_ENUM is generated by CompressedTexImage1DARB or CompressedTexImage3DARB if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT.

INVALID_OPERATION is generated by CompressedTexImage2DARB if if <internalformat> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and <border> is not equal to zero.

INVALID_ENUM is generated by CompressedTexSubImage1DARB or CompressedTexSubImage3DARB if <format> is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT.

INVALID_OPERATION is generated by TexSubImage2D CopyTexSubImage2D, or

CompressedTexSubImage2D if INTERNAL_FORMAT is COMPRESSED_RGB_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT1_EXT, COMPRESSED_RGBA_S3TC_DXT3_EXT, or COMPRESSED_RGBA_S3TC_DXT5_EXT and any of the following apply: <width> is not a multiple of four or equal to TEXTURE_WIDTH; <height> is not a multiple of four or equal to TEXTURE_HEIGHT; <xoffset> or <yoffset> is not a multiple of four.

The following restrictions from the ARB_texture_compression specification do not apply to S3TC texture formats, since subimage modification is straightforward as long as the subimage is properly aligned.

DELETE: INVALID_OPERATION is generated by TexSubImage1D, TexSubImage2D,
DELETE: TexSubImage3D, CopyTexSubImage1D, CopyTexSubImage2D, or
DELETE: CopyTexSubImage3D if the internal format of the texture image is
DELETE: compressed and <xoffset>, <yoffset>, or <zoffset> does not equal
DELETE: -b, where b is value of TEXTURE_BORDER.

DELETE: INVALID_VALUE is generated by CompressedTexSubImage1DARB,
DELETE: CompressedTexSubImage2DARB, or CompressedTexSubImage3DARB if the
DELETE: entire texture image is not being edited: if <xoffset>,
DELETE: <yoffset>, or <zoffset> is greater than -b, <xoffset> + <width> is
DELETE: less than w+b, <yoffset> + <height> is less than h+b, or <zoffset>
DELETE: + <depth> is less than d+b, where b is the value of
DELETE: TEXTURE_BORDER, w is the value of TEXTURE_WIDTH, h is the value of
DELETE: TEXTURE_HEIGHT, and d is the value of TEXTURE_DEPTH.

See also errors in the GL_ARB_texture_compression specification.

New State

None.

Appendix

S3TC Compressed Texture Image Formats

Compressed texture images stored using the S3TC compressed image formats are represented as a collection of 4x4 texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each 4x4 block is treated as a single pixel. If an S3TC image has a width or height less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an S3TC image with a width of <w>, height of <h>, and block size of <blocksize> (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

    ceil(<w>/4) * ceil(<h>/4) * blocksize.

When decoding an S3TC image, the block containing the texel at offset (<x>, <y>) begins at an offset (in bytes) relative to the beginning of the image of:

    blocksize * (ceil(<w>/4) * floor(<y>/4) + floor(<x>/4)).

There are four distinct S3TC image formats:

COMPRESSED_RGB_S3TC_DXT1_EXT: Each 4x4 block of texels consists of 64 bits of RGB image data.

Each RGB image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

```
c0_lo, c0_hi, c1_lo, c1_hi, bits_0, bits_1, bits_2, bits_3
```

The 8 bytes of the block are decoded into three quantities:

```
color0 = c0_lo + c0_hi * 256
color1 = c1_lo + c1_hi * 256
bits   = bits_0 + 256 * (bits_1 + 256 * (bits_2 + 256 * bits_3))
```

color0 and color1 are 16-bit unsigned integers that are unpacked to RGB colors RGB0 and RGB1 as though they were 16-bit packed pixels with a <format> of RGB and a type of UNSIGNED_SHORT_5_6_5.

bits is a 32-bit unsigned integer, from which a two-bit control code is extracted for a texel at location (x,y) in the block using:

```
code(x,y) = bits[2*(4*y+x)+1..2*(4*y+x)+0]
```

where bit 31 is the most significant and bit 0 is the least significant bit.

The RGB color for a texel at location (x,y) in the block is given by:

```
RGB0,              if color0 > color1 and code(x,y) == 0
RGB1,              if color0 > color1 and code(x,y) == 1
(2*RGB0+RGB1)/3,   if color0 > color1 and code(x,y) == 2
(RGB0+2*RGB1)/3,   if color0 > color1 and code(x,y) == 3

RGB0,              if color0 <= color1 and code(x,y) == 0
RGB1,              if color0 <= color1 and code(x,y) == 1
(RGB0+RGB1)/2,     if color0 <= color1 and code(x,y) == 2
BLACK,             if color0 <= color1 and code(x,y) == 3
```

Arithmetic operations are done per component, and BLACK refers to an RGB color where red, green, and blue are all zero.

Since this image has an RGB format, there is no alpha component and the image is considered fully opaque.

COMPRESSED_RGBA_S3TC_DXT1_EXT: Each 4x4 block of texels consists of 64 bits of RGB image data and minimal alpha information. The RGB components of a texel are extracted in the same way as COMPRESSED_RGB_S3TC_DXT1_EXT.

The alpha component for a texel at location (x,y) in the block is given by:

```
0.0,               if color0 <= color1 and code(x,y) == 3
1.0,               otherwise
```

IMPORTANT: When encoding an RGBA image into a format using 1-bit alpha, any texels with an alpha component less than 0.5 end up with an alpha of 0.0 and any texels with an alpha component greater than or equal to 0.5 end up with an alpha of 1.0. When encoding an RGBA image into the COMPRESSED_RGBA_S3TC_DXT1_EXT format, the resulting red, green, and blue components of any texels with a final alpha of 0.0 will automatically be zero (black). If this behavior is not desired by an application, it should not use COMPRESSED_RGBA_S3TC_DXT1_EXT. This format will never be used when a generic compressed internal format (Table 3.16.2) is specified, although the nearly identical format COMPRESSED_RGB_S3TC_DXT1_EXT (above) may be.

COMPRESSED_RGBA_S3TC_DXT3_EXT: Each 4x4 block of texels consists of 64 bits of uncompressed alpha image data followed by 64 bits of RGB image data.

Each RGB image data block is encoded according to the COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though color0 > color1, regardless of the actual values of color0 and color1.

Each alpha image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

```
a0, a1, a2, a3, a4, a5, a6, a7
```

The 8 bytes of the block are decoded into one 64-bit integer:

```
alpha = a0 + 256 * (a1 + 256 * (a2 + 256 * (a3 + 256 * (a4 +
             256 * (a5 + 256 * (a6 + 256 * a7))))))
```

alpha is a 64-bit unsigned integer, from which a four-bit alpha value is extracted for a texel at location (x,y) in the block using:

```
alpha(x,y) = bits[4*(4*y+x)+3..4*(4*y+x)+0]
```

where bit 63 is the most significant and bit 0 is the least significant bit.

The alpha component for a texel at location (x,y) in the block is given by alpha(x,y) / 15.

COMPRESSED_RGBA_S3TC_DXT5_EXT: Each 4x4 block of texels consists of 64 bits of compressed alpha image data followed by 64 bits of RGB image data.

Each RGB image data block is encoded according to the COMPRESSED_RGB_S3TC_DXT1_EXT format, with the exception that the two code bits always use the non-transparent encodings. In other words, they are treated as though color0 > color1, regardless of the actual values of color0 and color1.

Each alpha image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

```
alpha0, alpha1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5
```

The alpha0 and alpha1 are 8-bit unsigned bytesw converted to alpha components by multiplying by 1/255.

The 6 "bits" bytes of the block are decoded into one 48-bit integer:

```
bits = bits_0 + 256 * (bits_1 + 256 * (bits_2 + 256 * (bits_3 +
                256 * (bits_4 + 256 * bits_5))))
```

bits is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x,y) in the block using:

```
code(x,y) = bits[3*(4*y+x)+1..3*(4*y+x)+0]
```

where bit 47 is the most significant and bit 0 is the least significant bit.

```
    The alpha component for a texel at location (x,y) in the block is
    given by:

          alpha0,                     code(x,y) == 0
          alpha1,                     code(x,y) == 1

          (6*alpha0 + 1*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 2
          (5*alpha0 + 2*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 3
          (4*alpha0 + 3*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 4
          (3*alpha0 + 4*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 5
          (2*alpha0 + 5*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 6
          (1*alpha0 + 6*alpha1)/7,  alpha0 > alpha1 and code(x,y) == 7

          (4*alpha0 + 1*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 2
          (3*alpha0 + 2*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 3
          (2*alpha0 + 3*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 4
          (1*alpha0 + 4*alpha1)/5,  alpha0 <= alpha1 and code(x,y) == 5
          0.0,                      alpha0 <= alpha1 and code(x,y) == 6
          1.0,                      alpha0 <= alpha1 and code(x,y) == 7
```

Revision History

```
    1.0,  07/07/00 prbrown1:  Published final version agreed to by working
                              group members.

    0.9,  06/24/00 prbrown1:  Documented that block-aligned TexSubImage calls
                              do not modify existing texels outside the
                              modified blocks.  Added caveat to allow for a
                              (0,0)-anchored TexSubImage operation of
                              arbitrary size.

    0.7,  04/11/00 prbrown1:  Added issues on DXT1, DXT3, and DXT5 encodings
                              where the MSDN documentation doesn’t match what
                              is really done.  Added enum values from the
                              extension registry.

    0.4,  03/28/00 prbrown1:  Updated to reflect final version of the
                              ARB_texture_compression extension.  Allowed
                              block-aligned TexSubImage calls.

    0.3,  03/07/00 prbrown1:  Resolved issues pertaining to the format of RGB
                              blocks in the DXT3 and DXT5 formats (they don’t
                              ever use the "transparent" encoding).  Fixed
                              decoding of DXT1 blocks.  Pointed out issue of
                              "transparent" texels in DXT1 encodings having
                              different behaviors for RGB and RGBA internal
                              formats.

    0.2,  02/23/00 prbrown1:  Minor revisions; added several issues.

    0.11, 02/17/00 prbrown1:  Slight modification to error semantics
      (INVALID_ENUM instead of INVALID_OPERATION).

    0.1,  02/15/00 prbrown1:  Initial revisio
```

Name

EXT_texture_cube_map

Name Strings

GL_EXT_texture_cube_map

Notice

Version

January 13, 2000

Number

ARB_ version is "ARB extension #6"

Because this extension was promoted from an EXT to ARB extension before an extension number was assigned, this extension has no extension number.

Dependencies

None.

Written based on the wording of the OpenGL 1.2 specification but not dependent on it.

Overview

This extension provides a new texture generation scheme for cube map textures. Instead of the current texture providing a 1D, 2D, or 3D lookup into a 1D, 2D, or 3D texture image, the texture is a set of six 2D images representing the faces of a cube. The (s,t,r) texture coordinates are treated as a direction vector emanating from the center of a cube. At texture generation time, the interpolated per-fragment (s,t,r) selects one cube face 2D image based on the largest magnitude coordinate (the major axis). A new 2D (s,t) is calculated by dividing the two other coordinates (the minor axes values) by the major axis value. Then the new (s,t) is used to lookup into the selected 2D texture image face of the cube map.

Unlike a standard 1D, 2D, or 3D texture that have just one target, a cube map texture has six targets, one for each of its six 2D texture image cube faces. All these targets must be consistent, complete, and have a square dimension.

This extension also provides two new texture coordinate generation modes for use in conjunction with cube map texturing. The reflection map mode generates texture coordinates (s,t,r) matching the vertex’s eye-space reflection vector. The reflection map mode is useful for environment mapping without the singularity inherent in sphere mapping. The normal map mode generates texture coordinates (s,t,r) matching the vertex’s transformed eye-space normal. The normal map mode is useful for sophisticated cube map texturing-based diffuse lighting models.

The intent of the new texgen functionality is that an application using cube map texturing can use the new texgen modes to automatically generate the reflection or normal vectors used to look up into the cube map texture.

An application note: When using cube mapping with dynamic cube maps (meaning the cube map texture is re-rendered every frame), by keeping the cube map’s orientation pointing at the eye position, the texgen-computed reflection or normal vector texture coordinates can be always properly oriented for the cube map. However if the cube map is static (meaning that when view changes, the cube map texture is not updated), the texture matrix must be used to rotate the texgen-computed reflection or normal vector texture coordinates to match the orientation of the cube map. The rotation can be computed based on two vectors: 1) the direction vector from the cube map center to the eye position (both in world coordinates), and 2) the cube map orientation in world coordinates. The axis of rotation is the cross product of these two vectors; the angle of rotation is the arcsin of the dot product of these two vectors.

Issues

Should we place the normal/reflection vector in the (s,t,r) texture coordinates or (s,t,q) coordinates?

RESOLUTION: (s,t,r). Even if hardware uses "q" for the third component, the API should claim to support generation of (s,t,r) and let the texture matrix (through a concatenation with the user-supplied texture matrix) move "r" into "q".

Should the texture coordinate generation functionality for cube mapping be specified as a distinct extension from the actual cube map texturing functionality?

RESOLUTION: NO. Real applications and real implementations of cube mapping will tie the texgen and texture generation functionality together. Applications won’t have to query two separate extensions then.

While applications will almost always want to use the texgen functionality for automatically generating the reflection or normal vector as texture coordinates (s,t,r), this extension does permit an application to manually supply the reflection or normal vector through glTexCoord3f explicitly.

Note that the NV_texgen_reflection extension does "unbundle" the texgen functionality from cube maps.

Should you be able to have some texture coordinates computing REFLECTION_MAP_EXT and others not? Same question with NORMAL_MAP_EXT.

RESOLUTION: YES. This is the way that SPHERE_MAP works. It is not clear that this would ever be useful though.

Should something special be said about the handling of the q texture coordinate for this spec?

RESOLUTION: NO. But the following paragraph is useful for implementors concerned about the handling of q.

The REFLECTION_MAP_EXT and NORMAL_MAP_EXT modes are intended to supply reflection and normal vectors for cube map texturing hardware. When these modes are used for cube map texturing, the generated texture coordinates can be thought of as a reflection vector. The value of the q texture coordinate then simply scales the vector but does not change its direction. Because only the vector direction (not the vector magnitude) matters for cube map texturing, implementations are free to leave q undefined when any of the s, t, or r texture coordinates are generated using REFLECTION_MAP_EXT or NORMAL_MAP_EXT.

How should the cube faces be labeled?

RESOLUTION: Match the render man specification’s names of "px" (positive X), "nx" (negative x), "py", "ny", "pz", and "nz". There does not actually need to be an "ordering for the faces" (Direct3D 7.0 does number their cube map faces.) For this extension, the symbolic target names (TEXTURE_CUBE_MAP_POSITIVE_X_EXT, etc) is sufficient without requiring any specific ordering.

What coordinate system convention should be used? LHS or RHS?

RESOLUTION: The coordinate system is left-handed if you think of yourself within the cube. The coordinate system is right-handed if you think of yourself outside the cube.

This matches the convention of the RenderMan interface. If you look at Figure 12.8 (page 265) in "The RenderMan Companion", think of the cube being folded up with the observer inside the cube. Then the coordinate system convention is left-handed.

The spec just linearly interpolates the reflection vectors computed per-vertex across polygons. Is there a problem interpolating reflection vectors in this way?

Probably. The better approach would be to interpolate the eye vector and normal vector over the polygon and perform the reflection vector computation on a per-fragment basis. Not doing so is likely to lead to artifacts because angular changes in the normal vector result in twice as large a change in the reflection vector as normal vector changes. The effect is likely to be reflections that become glancing reflections too fast over the surface of the polygon.

Note that this is an issue for REFLECTION_MAP_EXT, but not NORMAL_MAP_EXT.

What happens if an (s,t,q) is passed to cube map generation that is close to (0,0,0), ie. a degenerate direction vector?

RESOLUTION: Leave undefined what happens in this case (but may not lead to GL interruption or termination).

Note that a vector close to (0,0,0) may be generated as a result of the per-fragment interpolation of (s,t,r) between vertices.

Do we need a distinct proxy texture mechanism for cube map textures?

RESOLUTION: YES. Cube map textures take up six times the memory as a conventional 2D image texture so proxy 2D texture determinations won’t be of value for a cube map texture. Cube maps need their own proxy target.

Should we require the 2D texture image width and height to be identical (ie, square only)?

RESOLUTION: YES. This limitation is quite a reasonable limitation and DirectX 7 has the same limitation.

This restriction is enforced by generating an INVALID_VALUE when calling TexImage2D or CopyTexImage2D with a non-equal width and height.

Some consideration was given to enforcing the "squarness" constraint as a texture consistency constraint. This is confusing however since the squareness is known up-front at texture image specification time so it seems confusing to silently report the usage error as a texture consistency issue.

Texture consistency still says that all the level 0 textures of all six faces must have the same square size.

If some combination of 1D, 2D, 3D, and cube map texturing is enabled, which really operates?

RESOLUTION: Cube map texturing. In OpenGL 1.2, 3D takes priority over 2D takes priority over 1D. Cube mapping should take priority over all conventional n-dimensional texturing schemes.

Does anything need to be said about combining cube mapping with multitexture?

RESOLUTION: NO. Cube mapping should be available on either texture unit. The hardware should fully orthogonal in its handling of cube map textures.

Does it make sense to support borders for cube map textures.

Actually, it does. It would be nice if the texture border pixels match the appropriate texels from the edges of the other cube map faces that they junction with. For this reason, we’ll leave the texture border capability implicitly supported.

How does mipmap level-of-detail selection work for cube map textures?

The existing spec’s language about LOD selection is fine.

Should the implementation dependent value for the maximum texture size for a cube map be the same as MAX_TEXTURE_SIZE?

RESOLUTION: NO. OpenGL 1.2 has a different MAX_3D_TEXTURE_SIZE for 3D textures, and cube maps should take six times more space than a 2D texture map of the same width & height. The implementation dependent MAX_CUBE_MAP_TEXTURE_SIZE_EXT constant should be used for cube maps then.

Note that the proxy cube map texture provides a better way to find out the maximum cube map texture size supported since the proxy mechanism can take into account the internal format, etc.

In section 3.8.10 when the "largest magnitude coordinate direction" is choosen, what happens if two or more of the coordinates (rx,ry,rz) have the identical magnitude?

RESOLUTION: Implementations can define their own rule to choose the largest magnitude coordinate direction whne two or more of the coordinates have the identical magnitude. The only restriction is that the rule must be deterministic and depend only on (rx,ry,rz).

In practice, (s,t,r) is interpolated across polygons so the cases where |s|==|t|, etc. are pretty arbitary (the equality depends on interpolation precision). This extension could mandate a particular rule, but that seems heavy-handed and there is no good reason that multiple vendors should be forced to implement the same rule.

Should there be limits on the supported border modes for cube maps?

RESOLUTION: NO. The specificiation is written so that cube map texturing proceeds just like conventional 2D texture mapping once the face determination is made.

Therefore, all OpenGL texture wrap modes should be supported though some modes are clearly inappropriate for cube maps. The WRAP mode is almost certainly incorrect for cube maps. Likewise, the CLAMP mode without a texture border is almost certainly incorrect for cube maps. CLAMP when a texture border is present and CLAMP_TO_EDGE are both reasonably suited for cube maps. Ideally, CLAMP with a texture border works best if the cube map edges can be replicated in the approriate texture borders of adjacent cube map faces. In practice, CLAMP_TO_EDGE works reasonably well in most circumstances.

Perhaps another extension could support a special cube map wrap mode that automatically wraps individual texel fetches to the appropriate adjacent cube map face. The benefit from such a mode is small and the implementation complexity is involved so this wrap mode should not be required for a basic cube map texture extension.

How is mipmap LOD selection handled for cube map textures?

RESOLUTION: The specification is written so that cube map texturing proceeds just like conventional 2D texture mapping once the face determination is made.

Thereforce, the partial differentials in Section 3.8.5 (page 126) should be evaluated for the u and v parameters based on the post-face determination s and t.

In Section 2.10.3 "Normal Transformation", there are several versions of the eye-space normal vector to choose from. Which one should the NORMAL_MAP_ARB texgen mode use?

RESOLUTION: nf. The nf vector is the final normal, post-rescale normal and post-normalize. In practice, the rescale normal and normalize operations do not change the direction of the vector so the choice of which version of transformed normal is used is not important for cube maps.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

```
    NORMAL_MAP_EXT                      0x8511
    REFLECTION_MAP_EXT                  0x8512
```

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain NORMAL_MAP_EXT or REFLECTION_MAP_EXT.

Accepted by the <cap> parameter of Enable, Disable, IsEnabled, and by the <pname> parameter of GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev, and by the <target> parameter of BindTexture, GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameteri, TexParameterfv, and TexParameteriv:

```
    TEXTURE_CUBE_MAP_EXT                0x8513
```

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

```
    TEXTURE_BINDING_CUBE_MAP_EXT        0x8514
```

Accepted by the <target> parameter of GetTexImage, GetTexLevelParameteriv, GetTexLevelParameterfv, TexImage2D, CopyTexImage2D, TexSubImage2D, and CopySubTexImage2D:

```
    TEXTURE_CUBE_MAP_POSITIVE_X_EXT     0x8515
    TEXTURE_CUBE_MAP_NEGATIVE_X_EXT     0x8516
    TEXTURE_CUBE_MAP_POSITIVE_Y_EXT     0x8517
    TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT     0x8518
    TEXTURE_CUBE_MAP_POSITIVE_Z_EXT     0x8519
    TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT     0x851A
```

Accepted by the <target> parameter of GetTexLevelParameteriv, GetTexLevelParameterfv, GetTexParameteriv, and TexImage2D:

```
PROXY_TEXTURE_CUBE_MAP_EXT          0x851B
```

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

```
MAX_CUBE_MAP_TEXTURE_SIZE_EXT       0x851C
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP_EXT, or NORMAL_MAP_EXT."

Add these paragraphs after the 4th paragraph:

"If TEXTURE_GEN_MODE indicates REFLECTION_MAP_EXT, compute the reflection vector r as described for the SPHERE_MAP mode. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = rx; the value assigned to a t coordinate is t = ry; and the value assigned to a r coordinate is r = rz. Calling TexGen with a <coord> of Q when <pname> indicates REFLECTION_MAP_EXT generates the error INVALID_ENUM.

If TEXTURE_GEN_MODE indicates NORMAL_MAP_EXT, compute the normal vector nf as described in section 2.10.3. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = nfx; the value assigned to a t coordinate is t = nfy; and the value assigned to a r coordinate is r = nfz. (The values nfx, nfy, and nfz are the components of nf.) Calling TexGen with a <coord> of Q when <pname> indicates NORMAL_MAP_EXT generates the error INVALID_ENUM.

The last paragraph’s first sentence should be changed to:

"The state required for texture coordinate generation comprises a five-valued integer for each coordinate indicating coordinate generation mode, ..."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.6.5 "Pixel Transfer Operations" under "Convolution"

Change this paragraph to say:

... "If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to DrawPixels, CopyPixels, ReadPixels, TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, and CopyTexSubImage3D, and returned by GetTexImage with one of the targets TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_EXT, TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, or TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT."

-- Section 3.8.1 "Texture Image Specification"

Change the first full sentence on page 117 to:

"<target> must be one of TEXTURE_2D for a 2D texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X_EXT, TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, or TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT for a cube map texture. Additionally, <target> can be either PROXY_TEXTURE_2D for a 2D proxy texture or PROXY_TEXTURE_CUBE_MAP_EXT for a cube map proxy texture as discussed in section 3.8.7."

Add the following paragraphs after the first paragraph on page 117:

"A 2D texture consists of a single 2D texture image. A cube map texture is a set of six 2D texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The TEXTURE_CUBE_MAP_*_EXT targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map 2D image tokens such as TEXTURE_CUBE_MAP_POSITIVE_X_EXT are used when specifying, updating, or querying, one of a cube map’s six 2D image, but when enabling cube map texturing or binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular 2D image), the TEXTURE_CUBE_MAP_EXT target is specified.

When the target parameter to TexImage2D is one of the six cube map 2D image targets, the error INVALID_VALUE is generated if the width and height parameters are not equal.

If cube map texturing is enabled at the time a primitive is rasterized and if the set of six targets are not "cube complete", then it is as if texture mapping were disabled. The targets of a cube map texture are "cube complete" if the array 0 of all six targets have identical and square dimensions, the array 0 of all six targets were specified with the same internalformat, and the array 0 of all six targets have the same border width."

After the 14th paragraph add:

"In a similiar fashion, the maximum allowable width and height (they must be the same) of a cube map texture must be at least 2^(k-lod)+2bt for image arrays level 0 through k, where k is the log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE_EXT."

-- Section 3.8.2 "Alternate Texture Image Specification Commands"

Update the second paragraph (page 120) to say:

... "Currently, <target> must be TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_EXT, TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, or TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT." ...

Add after the second paragraph (page 120), the following:

"When the target parameter to CopyTexImage2D is one of the six cube map 2D image targets, the error INVALID_VALUE is generated if the width and height parameters are not equal."

Update the fourth paragraph (page 121) to say:

... "Currently the target arguments of TexSubImage1D and CopyTexSubImage1D must be TEXTURE_1D, the <target> arguments of TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_X_EXT, TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, or TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT, and the <target> arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D." ...

-- Section 3.8.3 "Texture Parameters"

Change paragraph one (page 124) to say:

... "<target> is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT." ...

Add a final paragraph saying:

"Texture parameters for a cube map texture apply to cube map as a whole; the six distinct 2D texture images use the texture parameters of the cube map itself.

-- Section 3.8.5 "Texture Minification" under "Mipmapping"

Change the first full paragraph on page 130 to:

... "If texturing is enabled for one-, two-, or three-dimensional texturing but not cube map texturing (and TEXTURE_MIN_FILTER is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays TEXTURE_BASE_LEVEL through q = min{p,TEXTURE_MAX_LEVEL} is incomplete, based on the dimensions of array 0, then it is as if texture mapping were disabled."

Follow the first full paragraph on page 130 with:

"If cube map texturing is enabled and TEXTURE_MIN_FILTER is one that requires mipmap levels at the time a primitive is rasterized and if the set of six targets are not "mipmap cube complete", then it is as if texture mapping were disabled. The targets of a cube map texture are "mipmap cube complete" if the six cube map targets are "cube complete" and the set of arrays TEXTURE_BASE_LEVEL through q are not incomplete (as described above)."

-- Section 3.8.7 "Texture State and Proxy State"

Change the first sentence of the first paragraph (page 131) to say:

"The state necessary for texture can be divided into two categories. First, there are the nine sets of mipmap arrays (three for the one-, two-, and three-dimensional texture targets and six for the cube map texture targets) and their number." ...

Change the second paragraph (page 132) to say:

"In addition to the one-, two-, three-dimensional, and the six cube map sets of image arrays, the partially instantiated one-, two-, and three-dimensional and one cube map sets of proxy image arrays are maintained." ...

After the third paragraph (page 132) add:

"The cube map proxy arrays are operated on in the same manner when TexImage2D is executed with the <target> field specified as PROXY_TEXTURE_CUBE_MAP_EXT with the addition that determining that a given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP_EXT indicates that all six of the cube map 2D images are supported. Likewise, if the specified PROXY_TEXTURE_CUBE_MAP_EXT is not supported, none of the six cube map 2D images are supported."

Change the second sentence of the fourth paragraph (page 132) to:

"Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP_EXT cannot be used as textures, and their images must never be queried using GetTexImage." ...

-- Section 3.8.8 "Texture Objects"

Change the first sentence of paragraph one (page 133) to say:

"In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP_EXT, named one-, two-, and three-dimensional texture objects and cube map texture objects can be created and operated on." ...

Change the second paragraph (page 133) to say:

"A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT." ... "If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT, it remains a one-, two-, three-dimensional, or cube map texture until it is deleted."

Change the third paragraph (page 133) to say:

"BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT."

Change paragraph five (page 133) to say:

"In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP have one-dimensional, two-dimensional, three-dimensional, and cube map state vectors associated with them respectively." ... "The initial, one-dimensional, two-dimensional, three-dimensional, and cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTUER_2D, TEXTURE_3D, and TEXTURE_CUBE_MAP_EXT respectively while 0 is bound to the corresponding targets."

Change paragraph six (page 134) to say:

... "If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT is deleted, it is as though BindTexture has been executed with the same <target> and <texture> zero." ...

-- Section 3.8.10 "Texture Application"

Replace the beginning sentences of the first paragraph (page 136) with:

"Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT to enable the one-dimensional, two-dimensional, three-dimensional, or cube map texturing respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-, two-, or one-dimensional textures is enabled, then cube map texturing is used. If texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality.

However, when cube map texturing is enabled, the rules are more complicated. For cube map texturing, the (s,t,r) texture coordinates are treated as a direction vector (rx,ry,rz) emanating from the center of a cube. (The q coordinate can be ignored since it merely scales the vector without affecting the direction.) At texture application time, the interpolated per-fragment (s,t,r) selects one of the cube map face’s 2D image based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on (rx,ry,rz). The target column in the table below explains how the major axis direction maps to the 2D image of a particular cube map target.

major axis direction	target	sc	tc	ma
+rx	TEXTURE_CUBE_MAP_POSITIVE_X_EXT	-rz	-ry	rx
-rx	TEXTURE_CUBE_MAP_NEGATIVE_X_EXT	+rz	-ry	rx
+ry	TEXTURE_CUBE_MAP_POSITIVE_Y_EXT	+rx	+rz	ry
-ry	TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT	+rx	-rz	ry
+rz	TEXTURE_CUBE_MAP_POSITIVE_Z_EXT	+rx	-ry	rz
-rz	TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT	-rx	-ry	rz

Using the sc, tc, and ma determined by the major axis direction as specified in the table above, an updated (s,t) is calculated as follows

```
s   =   ( sc/|ma| + 1 ) / 2
t   =   ( tc/|ma| + 1 ) / 2
```

If |ma| is zero or very nearly zero, the results of the above two equations need not be defined (though the result may not lead to GL interruption or termination).

This new (s,t) is used to find a texture value in the determined face’s 2D texture image using the rules given in sections 3.8.5 and 3.8.6." ...

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

-- Section 5.4 "Display Lists"

In the second to the last paragraph (page 179), add PROXY_TEXTURE_CUBE_MAP_EXT to the list of PROXY_* tokens.

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the fourth paragraph (page 183) to say:

"The GetTexParameter parameter <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP_EXT, indicating the currently bound one-dimensional, two-dimensional, three-dimensional, or cube map texture object. For GetTexLevelParameter, <target> may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_CUBE_MAP_POSITIVE_X_EXT, TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, or PROXY_TEXTURE_CUBE_MAP_EXT, indicating the one-dimensional, two-dimensional, three-dimensional texture object, or distinct cube map texture 2D image, or one-dimensional, two-dimensional, three-dimensional, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP_EXT is not a valid <target> parameter for GetTexLevelParameter because it does not specify a particular cube map face."

-- Section 6.1.4 "Texture Queries"

Change the first paragraph to read:

... "It is somewhat different from the other get commands; <tex> is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D indicates a one-dimensional texture, TEXTURE_2D indicates a two-dimensional texture, TEXTURE_3D indicates a three-dimensional texture, and TEXTURE_CUBE_MAP_POSITIVE_X_EXT, TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, and TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT indicate the respective face of a cube map texture.

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <coord> of Q when <pname> indicates REFLECTION_MAP_EXT or NORMAL_MAP_EXT.

INVALID_VALUE is generated when the target parameter to TexImage2D or CopyTexImage2D is one of the six cube map 2D image targets and the width and height parameters are not equal.

New State

(table 6.12, p202) add the following entries:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_CUBE_MAP_EXT	B	IsEnabled	False	True if cube map texturing is enabled	3.8.10	texture/enable
TEXTURE_BINDING_CUBE_MAP_EXT	Z+	GetIntegerv	0	Texture object for TEXTURE_CUBE_MAP	3.8.8	texture
TEXTURE_CUBE_MAP_POSITIVE_X_EXT	nxI	GetTexImage	see 3.8	positive x face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_X_EXT	nxI	GetTexImage	see 3.8	negative x face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_POSITIVE_Y_EXT	nxI	GetTexImage	see 3.8	positive y face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT	nxI	GetTexImage	see 3.8	negative y face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_POSITIVE_Z_EXT	nxI	GetTexImage	see 3.8	positive z face cube map texture image at lod i	3.8	-
TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT	nxI	GetTexImage	see 3.8	negative z face cube map texture image at lod i	3.8	-

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture

(the type changes from 4xZ3 to 4xZ5)

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_CUBE_MAP_TEXTURE_SIZE_EXT	Z+	GetIntegerv	16	Maximum cube map texture image dimension	3.8.1	-

Name

EXT_texture_edge_clamp

Name Strings

GL_EXT_texture_edge_clamp

Version

$Date: 1997/09/22 23:04:01 $ $Revision: 1.1 $

Dependencies

SGIS_texture_filter4 affects the definition of this extension

Overview

The base OpenGL provides clamping such that the texture coordinates are limited to exactly the range [0,1]. When a texture coordinate is clamped using this algorithm, the texture sampling filter straddles the edge of the texture image, taking 1/2 its sample values from within the texture image, and the other 1/2 from the texture border. It is sometimes desirable to clamp a texture without requiring a border, and without using the constant border color.

This extension defines a new texture clamping algorithm. CLAMP_TO_EDGE_EXT clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. When used with a NEAREST or a LINEAR filter, the color returned when clamping is derived only from texels at the edge of the texture image. When used with FILTER4 filters, the filter operations of CLAMP_TO_EDGE_EXT are defined but don’t result in a nice clamp-to-edge color.

CLAMP_TO_EDGE_EXT is supported by 1, 2, and 3-dimensional textures only.

Issues

- Is the arithmetic for FILTER4 filters correct? Is this the right thing to do?

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameter of TexParameteri and TexParameterf, and by the <params> parameter of TexParameteriv and TexParameterfv, when their <pname> parameter is TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R:

```
CLAMP_TO_EDGE_EXT               0x812F
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_WRAP_R	integer	CLAMP, REPEAT, CLAMP_TO_EDGE_EXT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS, LINEAR_CLIPMAP_LINEAR_SGIX
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS, LINEAR_LEQUAL_R_SGIS, LINEAR_GEQUAL_R_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD	float	any value
TEXTURE_MAX_LOD	float	any value
TEXTURE_BASE_LEVEL	integer	any non-negative integer
TEXTURE_MAX_LEVEL	integer	any non-negative integer
GENERATE_MIPMAP_SGIS	boolean	TRUE or FALSE
TEXTURE_CLIPMAP_OFFSET_SGIX	2 floats	any 2 values

Table 3.7: Texture parameters and their values.

CLAMP_TO_EDGE_EXT texture clamping is specified by calling TexParameteri with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D, <pname> set to TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R, and <param> set to CLAMP_TO_EDGE_EXT.

Let [min,max] be the range of a clamped texture coordinate, and let N be the size of the 1D, 2D, or 3D texture image in the direction of clamping. Then in all cases

max = 1 - min

because the clamping is always symmetric about the [0,1] mapped range of a texture coordinate. When used with NEAREST or LINEAR filters, CLAMP_TO_EDGE_EXT defines a minimum clamping value of

min = 1 / 2*N

When used with FILTER4 filters, CLAMP_TO_EDGE_EXT defines a minimum clamping value of

```
    min = 3 / 2*N,          N > 2

    min = 1/2               N <= 2
```

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Framebuffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on SGIS_texture_filter4

If SGIS_texture_filter4 is not implemented, then discussions about the interaction of filter4 texture filters and the clamping function described in this file are invalid, and should be ignored.

Errors

None

New State

Only the type information changes for these parameters:

Get Value	Get Command	Type	Initial Value	Attrib
TEXTURE_WRAP_S	GetTexParameteriv	n x Z3	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z3	REPEAT	texture
TEXTURE_WRAP_R	GetTexParameteriv	n x Z3	REPEAT	texture

New Implementation Dependent State

None

Name

    EXT_texture_env_add

Name Strings

    GL_EXT_texture_env_add

Contact

    Michael Gold, NVIDIA (gold ’at’ nvidia.com)
    Tom Frisinger, ATI (tfrisinger ’at’ atitech.com)

Status

    Shipping (version 1.6)

Version

    $Date: 1999/03/22 17:28:00 $ $Revision: 1.1 $

Number

    185

Dependencies

    None

Overview

    New texture environment function ADD is supported with the following
    equation:
                        Cv = Cf + Ct

    New function may be specified by calling TexEnv with ADD token.

New Procedures and Functions

    None

New Tokens

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and
    TexEnvfi when the <pname> parameter value is GL_TEXTURE_ENV_MODE

        ADD

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

        None

Additions to Chapter 3 of the GL Specification (Rasterization)

	Texture Environment				
Base Texture Format	REPLACE	MODULATE	BLEND	DECAL	ADD
ALPHA	...	...	...	...	Rv = Rf
	...	...	...	...	Gv = Gf
	...	...	...	...	Bv = Bf
	...	...	...	...	Av = AfAt
LUMINANCE	...	...	...	...	Rv = Rf+Lt
	...	...	...	...	Gv = Gf+Lt
	...	...	...	...	Bv = Bf+Lt
	...	...	...	...	Av = Af
LUMINANCE_ALPHA	...	...	...	...	Rv = Rf+Lt
	...	...	...	...	Gv = Gf+Lt
	...	...	...	...	Bv = Bf+Lt
	...	...	...	...	Av = AfAt
INTENSITY	...	...	...	...	Rv = Rf+It
	...	...	...	...	Gv = Gf+It
	...	...	...	...	Bv = Bf+It
	...	...	...	...	Av = Af+It
RGB	...	...	...	...	Rv = Rf+Rt
	...	...	...	...	Gv = Gf+Gt
	...	...	...	...	Bv = Bf+Bt
	...	...	...	...	Av = Af
RGBA	...	...	...	...	Rv = Rf+Rt
	...	...	...	...	Gv = Gf+Gt
	...	...	...	...	Bv = Bf+Bt
	...	...	...	...	Av = AfAt

Table 3.11: Texture functions.

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

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX / WGL / AGL Specifications

None

GLX Protocol

None

Errors

None

New State

None

New Implementation Dependent State

None

Name

EXT_texture_env_combine

Name Strings

GL_EXT_texture_env_combine

Version

`$Date: 1999/04/02 13:54:17 $ $Revision: 1.7 $`

Number

158

Dependencies

SGI_texture_color_table affects the definition of this extension
SGIX_texture_scale_bias affects the definition of this extension

Overview

New texture environment function COMBINE_EXT allows programmable texture combiner operations, including:

REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_EXT	Arg0 + Arg1 - 0.5
INTERPOLATE_EXT	Arg0 * (Arg2) + Arg1 * (1-Arg2)

where Arg0, Arg1 and Arg2 are derived from

PRIMARY_COLOR_EXT	primary color of incoming fragment
TEXTURE	texture color of corresponding texture unit
CONSTANT_EXT	texture environment constant color
PREVIOUS_EXT	result of previous texture environment; on texture unit 0, this maps to PRIMARY_COLOR_EXT

and Arg2 is restricted to the alpha component of the corresponding source.

In addition, the result may be scaled by 1.0, 2.0 or 4.0.

Issues

Should the explicit bias be removed in favor of an implcit bias as part of a ADD_SIGNED_EXT function?

- Yes. This pre-scale bias is a special case and will be treated as such.

Should the primary color of the incoming fragment be available to all texture environments? Currently it is only available to the texture environment of texture unit 0.

- Yes, PRIMARY_COLOR_EXT has been added as an input source.

Should textures from other texture units be allowed as sources?

- No, not in the base spec. Too many vendors have expressed concerns about the scalability of such functionality. This can be added as a subsequent extension.

All of the 1.2 modes except BLEND can be expressed in terms of this extension. Should texture color be allowed as a source for Arg2, so all of the 1.2 modes can be expressed? If so, should all color sources be allowed, to maintain orthogonality?

- No, not in the base spec. This can be added as a subsequent extension.

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

COMBINE_EXT	0x8570

Accepted by the <pname> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <target> parameter value is TEXTURE_ENV

COMBINE_RGB_EXT	0x8571
COMBINE_ALPHA_EXT	0x8572
SOURCE0_RGB_EXT	0x8580
SOURCE1_RGB_EXT	0x8581
SOURCE2_RGB_EXT	0x8582
SOURCE0_ALPHA_EXT	0x8588
SOURCE1_ALPHA_EXT	0x8589
SOURCE2_ALPHA_EXT	0x858A
OPERAND0_RGB_EXT	0x8590
OPERAND1_RGB_EXT	0x8591
OPERAND2_RGB_EXT	0x8592
OPERAND0_ALPHA_EXT	0x8598
OPERAND1_ALPHA_EXT	0x8599
OPERAND2_ALPHA_EXT	0x859A
RGB_SCALE_EXT	0x8573
ALPHA_SCALE	

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is COMBINE_RGB_EXT or COMBINE_ALPHA_EXT

REPLACE	
MODULATE	
ADD	
ADD_SIGNED_EXT	0x8574
INTERPOLATE_EXT	0x8575

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,

```
    and TexEnviv when the <pname> parameter value is SOURCE0_RGB_EXT,
    SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE0_ALPHA_EXT,
    SOURCE1_ALPHA_EXT, or SOURCE2_ALPHA_EXT

        TEXTURE
        CONSTANT_EXT                            0x8576
        PRIMARY_COLOR_EXT                       0x8577
        PREVIOUS_EXT                            0x8578

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is
    OPERAND0_RGB_EXT or OPERAND1_RGB_EXT

        SRC_COLOR
        ONE_MINUS_SRC_COLOR
        SRC_ALPHA
        ONE_MINUS_SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is
    OPERAND0_ALPHA_EXT or OPERAND1_ALPHA_EXT

        SRC_ALPHA
        ONE_MINUS_SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is
    OPERAND2_RGB_EXT or OPERAND2_ALPHA_EXT

        SRC_ALPHA

    Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv,
    and TexEnviv when the <pname> parameter value is RGB_SCALE_EXT or
    ALPHA_SCALE

        1.0
        2.0
        4.0
```

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE_EXT, the form of the texture function depends on the values of COMBINE_RGB_EXT and COMBINE_ALPHA_EXT, according to table 3.20. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1].

COMBINE_RGB_EXT or COMBINE_ALPHA_EXT	Texture Function
REPLACE	Arg0
MODULATE	Arg0 * Arg1
ADD	Arg0 + Arg1
ADD_SIGNED_EXT	Arg0 + Arg1 - 0.5
INTERPOLATE_EXT	Arg0 * (Arg2) + Arg1 * (1-Arg2)

Table 3.20: COMBINE_EXT texture functions

The arguments Arg0, Arg1 and Arg2 are determined by the values of SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. The relationship is described in tables 3.21 and 3.22.

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.21: Arguments for COMBINE_RGB_EXT functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)

Table 3.22: Arguments for COMBINE_ALPHA_EXT functions

The mapping of texture components to source components is summarized in Table 3.23. In the following table, At, Lt, It, Rt, Gt and Bt are the filtered texel values.

Base Internal Format	RGB Values	Alpha Value
ALPHA	0, 0, 0	At
LUMINANCE	Lt, Lt, Lt	1
LUMINANCE_ALPHA	Lt, Lt, Lt	At
INTENSITY	It, It, It	It
RGB	Rt, Gt, Bt	1
RGBA	Rt, Gt, Bt	At

Table 3.23: Correspondence of texture components to source components for COMBINE_RGB_EXT and COMBINE_ALPHA_EXT arguments

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

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for COMBINE_RGB_EXT or COMBINE_ALPHA_EXT is not one of REPLACE, MODULATE, ADD, ADD_SIGNED_EXT, or INTERPOLATE_EXT.

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT or SOURCE2_ALPHA_EXT is not one of TEXTURE, CONSTANT_EXT, PRIMARY_COLOR_EXT or PREVIOUS_EXT.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_EXT or OPERAND1_RGB_EXT is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_EXT or OPERAND1_ALPHA_EXT is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND2_RGB_EXT or OPERAND2_ALPHA_EXT is not SRC_ALPHA.

INVALID_VALUE is generated if <params> value for RGB_SCALE_EXT or ALPHA_SCALE is not one of 1.0, 2.0, or 4.0.

Dependencies on SGI_texture_color_table

If SGI_texture_color_table is implemented, the expanded Rt, Gt, Bt, and At values are used directly instead of the expansion described by Table 3.23.

Dependencies on SGIX_texture_scale_bias

If SGIX_texture_scale_bias is implemented, the expanded Rt, Gt, Bt, and At values are used directly instead of the expansion described by Table 3.23.

New State

Get Value	Get Command	Type	Initial Value	Attribute
COMBINE_RGB_EXT	GetTexEnviv	n x Z4	MODULATE	texture
COMBINE_ALPHA_EXT	GetTexEnviv	n x Z4	MODULATE	texture
SOURCE0_RGB_EXT	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_RGB_EXT	GetTexEnviv	n x Z3	PREVIOUS_EXT	texture
SOURCE2_RGB_EXT	GetTexEnviv	n x Z3	CONSTANT_EXT	texture
SOURCE0_ALPHA_EXT	GetTexEnviv	n x Z3	TEXTURE	texture
SOURCE1_ALPHA_EXT	GetTexEnviv	n x Z3	PREVIOUS_EXT	texture
SOURCE2_ALPHA_EXT	GetTexEnviv	n x Z3	CONSTANT_EXT	texture
OPERAND0_RGB_EXT	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND1_RGB_EXT	GetTexEnviv	n x Z6	SRC_COLOR	texture
OPERAND2_RGB_EXT	GetTexEnviv	n x Z1	SRC_ALPHA	texture
OPERAND0_ALPHA_EXT	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND1_ALPHA_EXT	GetTexEnviv	n x Z4	SRC_ALPHA	texture
OPERAND2_ALPHA_EXT	GetTexEnviv	n x Z1	SRC_ALPHA	texture
RGB_SCALE_EXT	GetTexEnvfv	n x R3	1.0	texture
ALPHA_SCALE	GetTexEnvfv	n x R3	1.0	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

Because of a hardware limitation, TNT, TNT2, GeForce, and Quadro treat "scale by 4.0" with the COMBINE_RGB_EXT or COMBINE_ALPHA_EXT mode of ADD_SIGNED_EXT as "scale by 2.0".

Name

EXT_texture_filter_anisotropic

Name Strings

GL_EXT_texture_filter_anisotropic

Notice

Version

August 24, 1999

Number

187

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

Texture mapping using OpenGL’s existing mipmap texture filtering modes assumes that the projection of the pixel filter footprint into texture space is a square (ie, isotropic). In practice however, the footprint may be long and narrow (ie, anisotropic). Consequently, mipmap filtering severely blurs images on surfaces angled obliquely away from the viewer.

Several approaches exist for improving texture sampling by accounting for the anisotropic nature of the pixel filter footprint into texture space. This extension provides a general mechanism for supporting anisotropic texturing filtering schemes without specifying a particular formulation of anisotropic filtering.

The extension permits the OpenGL application to specify on a per-texture object basis the maximum degree of anisotropy to account for in texture filtering.

Increasing a texture object’s maximum degree of anisotropy may improve texture filtering but may also significantly reduce the implementation’s texture filtering rate. Implementations are free to clamp the specified degree of anisotropy to the implementation’s maximum supported degree of anisotropy.

A texture’s maximum degree of anisotropy is specified independent from the texture’s minification and magnification filter (as opposed to being supported as an entirely new filtering mode). Implementations are free to use the specified minification and magnification filter to select a particular anisotropic texture filtering scheme. For example, a NEAREST filter with a maximum degree of anisotropy of two could be treated as a 2-tap filter that accounts for the direction of anisotropy. Implementations are also permitted to ignore the minification or magnification filter and implement the highest quality of anisotropic filtering possible.

Applications seeking the highest quality anisotropic filtering available are advised to request a LINEAR_MIPMAP_LINEAR minification filter, a LINEAR magnification filter, and a large maximum degree of anisotropy.

Issues

Should there be a particular anisotropic texture filtering minification and magnification mode?

> RESOLUTION: NO. The maximum degree of anisotropy should control when anisotropic texturing is used. Making this orthogonal to the minification and magnification filtering modes allows these settings to influence the anisotropic scheme used. Yes, such an anisotropic filtering scheme exists in hardware.

What should the minimum value for MAX_TEXTURE_MAX_ANISTROPY_EXT be?

> RESOLUTION: 2.0. To support this extension, at least 2 to 1 anisotropy should be supported.

Should an implementation-defined limit for the maximum maximum degree of anisotropy be "get-able"?

> RESOLUTION: YES. But you should not assume that a high maximum maximum degree of anisotropy implies anything about texture filtering performance or quality.

Should anything particular be said about anisotropic 3D texture filtering?

> Not sure. Does the implementation example shown in the spec for 2D anisotropic texture filtering readily extend to 3D anisotropic texture filtering?

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of GetTexParameterfv, GetTexParameteriv, TexParameterf, TexParameterfv, TexParameteri, and TexParameteriv:

```
TEXTURE_MAX_ANISOTROPY_EXT          0x84FE
```

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

```
MAX_TEXTURE_MAX_ANISOTROPY_EXT      0x84FF
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Sections 3.8.3 "Texture Parameters"

Add the following entry to the end of Table 3.17:

Name	Type	Legal Values
TEXTURE_MAX_ANISOTROPY_EXT	float	greater or equal to 1.0

-- Sections 3.8.5 "Texture Minification" and 3.8.6 "Texture Magnification"

After the first paragraph in Section 3.8.5:

"When the texture's value of TEXTURE_MAX_ANISOTROPY_EXT is equal to 1.0, the GL uses an isotropic texture filtering approach as described in this section and Section 3.8.6. However, when the texture's value of TEXTURE_MAX_ANISOTROPY_EXT is greater than 1.0, the GL implementation should use a texture filtering scheme that accounts for a degree of anisotropy up to the smaller of the value of TEXTURE_MAX_ANISTROPY_EXT or the implementation-defined value of MAX_TEXTURE_MAX_ANISTROPY_EXT.

The particular scheme for anisotropic texture filtering is implementation dependent. Additionally, implementations are free to consider the current texture minification and magnification modes to control the specifics of the anisotropic filtering scheme used.

The anisotropic texture filtering scheme may only access mipmap levels if the minification filter is one that requires mipmaps. Additionally, when a minification filter is specified, the anisotropic texture filtering scheme may only access texture mipmap levels between the texture's values for TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL, inclusive. Implementations are also recommended to respect the values of TEXTURE_MAX_LOD and TEXTURE_MIN_LOD to whatever extent the particular anisotropic texture filtering scheme permits this."

The following describes one particular approach to implementing anisotropic texture filtering for the 2D texturing case:

"Anisotropic texture filtering substantially changes Section 3.8.5. Previously a single scale factor P was determined based on the pixel’s projection into texture space. Now two scale factors, Px and Py, are computed.

```
  Px = sqrt(dudx^2 + dvdx^2)
  Py = sqrt(dudy^2 + dvdy^2)

  Pmax = max(Px,Py)
  Pmin = min(Px,Py)

  N = min(ceil(Pmax/Pmin),maxAniso);
  Lamda’ = log2(Pmax/N)
```

where maxAniso is the smaller of the texture’s value of TEXTURE_MAX_ANISOTROPY_EXT or the implementation-defined value of MAX_TEXTURE_MAX_ANISOTROPY_EXT.

It is acceptable for implementation to round ’N’ up to the nearest supported sampling rate. For example an implementation may only support power-of-two sampling rates.

It is also acceptable for an implementation to approximate the ideal functions Px and Py with functions Fx and Fy subject to the following conditions:

```
  1.  Fx is continuous and monotonically increasing in |du/dx| and |dv/dx|.
      Fy is continuous and monotonically increasing in |du/dy| and |dv/dy|.

  2.  max(|du/dx|,|dv/dx|} <= Fx <= |du/dx| + |dv/dx|.
      max(|du/dy|,|dv/dy|} <= Fy <= |du/dy| + |dv/dy|.
```

Instead of a single sample, Tau, at (u,v,Lamda), ’N’ locations in the mipmap at LOD Lamda, are sampled within the texture footprint of the pixel. This sum TauAniso is defined using the single sample Tau. When the texture’s value of TEXTURE_MAX_ANISOTROPHY_EXT is greater than 1.0, use TauAniso instead of Tau to determine the fragment’s texture value.

```
                 i=N
                 ---
TauAniso = 1/N   \ Tau(u(x - 1/2 + i/(N+1), y), v(x - 1/2 + i/(N+1), y)),  Px > Py
                 /
                 ---
                 i=1

                 i=N
                 ---
TauAniso = 1/N   \ Tau(u(x, y - 1/2 + i/(N+1)), v(x, y - 1/2 + i/(N+1))),  Py >= Px
                 /
                 ---
                 i=1
```

It is acceptable to approximate the u and v functions with equally spaced samples in texture space at LOD Lamda:

```
                 i=N
                 ---
TauAniso = 1/N   \ Tau(u(x,y)+dudx(i/(N+1)-1/2), v(x,y)+dvdx(i/(N+1)-1/2)), Px > Py
                 /
                 ---
                 i=1

                 i=N
                 ---
TauAniso = 1/N   \ Tau(u(x,y)+dudy(i/(N+1)-1/2), v(x,y)+dvdy(i/(N+1)-1/2)), Py >= Px
                 /
                 ---
                 i=1
```

"

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_VALUE is generated when TexParameter is called with <pname> of TEXTURE_MAX_ANISOTROPY_EXT and a <param> value or value of what <params> points to less than 1.0.

New State

(table 6.13, p203) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_MAX_ANISOTROPY_EXT	R	GetTexParameterfv	1.0	Maximum degree of anisotropy	3.8.5	texture

New Implementation State

(table 6.25, p215) add the entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TEXTURE_MAX_ANISOTROPY_EXT	R	GetFloatv	2.0	Limit of maximum degree of anisotropy	3.8.5	-

Name

EXT_texture_lod_bias

Name Strings

GL_EXT_texture_lod_bias

Notice

Version

NVIDIA Date: May 23, 2000
$Date$ $Revision$

Number

186

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Affects ARB_multitexture.

Overview

OpenGL computes a texture level-of-detail parameter, called lambda in the GL specification, that determines which mipmap levels and their relative mipmap weights for use in mipmapped texture filtering.

This extension provides a means to bias the lambda computation by a constant (signed) value. This bias can provide a way to blur or pseudo-sharpen OpenGL's standard texture filtering.

This blurring or pseudo-sharpening may be useful for special effects (such as depth-of-field effects) or image processing techniques (where the mipmap levels act as pre-downsampled image versions). On some implementations, increasing the texture lod bias may improve texture filtering performance (at the cost of texture bluriness).

The extension mimics functionality found in Direct3D.

Issues

Should the texture LOD bias be settable per-texture object or per-texture stage?

RESOLUTION: Per-texture stage. This matches the Direct3D semantics for texture lod bias. Note that this differs from the semantics of SGI's SGIX_texture_lod_bias extension that has the biases per-texture object.

This also allows the same texture object to be used by two different texture units for different blurring. This is useful for extrapolating detail between various levels of detail in a mipmapped texture.

For example, you can extrapolate texture detail with ARB_multitexture and EXT_texture_env_combine by computing

```
(B0 - B2) * 2 + B2
```

where B0 is a non-biased texture (normal sharpness) and B2 is the same texture but bias by 2 levels-of-detail (fairly blurry). This has the effect of increasing the high-frequency information in the texture. There are immediate Earth Sciences and medical imaging applications for this technique.

Per-texture stage control of the LOD bias is also useful for allowing an application to control overall texture bluriness. This can be used in games to simulate disorientation (note that only textures will blur, not edges). It can also be used to globally control texturing performance. An application may be able to sustain a constant frame rate by avoiding texture fetch stalls by using slightly blurrier textures.

How does EXT_texture_lod_bias differ from SGIX_texture_lod bias?

EXT_texture_lod_bias adds a bias to lambda. The SGIX_texture_lod_bias extension changes the computation of rho (the log2 of which is lambda). The SGIX extension provides separate biases in each texture dimension. The EXT extension does not provide an "directionality" in the LOD control.

Does the texture lod bias occur before or after the TEXTURE_MAX_LOD and TEXTURE_MIN_LOD clamping?

RESOLUTION: BEFORE. This allows the texture lod bias to still be clamped within the max/min lod range.

Does anything special have to be said to keep the biased lambda value from being less than zero or greater than the maximum number of mipmap levels?

RESOLUTION: NO. The existing clamping in the specification handles these situations.

The texture lod bias is specified to be a float. In practice, what sort of range is assumed for the texture lod bias?

RESOLUTION: The MAX_TEXTURE_LOD_BIAS_EXT implementation constant advertises the maximum absolute value of the supported texture lod bias. The value is recommended to be at least the maximum mipmap level supported by the implementation.

The texture lod bias is specified to be a float. In practice, what sort of precision is assumed for the texture lod bias?

RESOLUTION; This is implementation dependent. Presumably, hardware would implement the texture lod bias as a fractional bias but the exact fractional precision supported is implementation dependent. At least 4 fractional bits is recommended.

New Procedures and Functions

None

New Tokens

Accepted by the <target> parameters of GetTexEnvfv, GetTexEnviv, TexEnvi, TexEnvf, Texenviv, and TexEnvfv:

```
TEXTURE_FILTER_CONTROL_EXT          0x8500
```

When the <target> parameter of GetTexEnvfv, GetTexEnviv, TexEnvi, TexEnvf, TexEnviv, and TexEnvfv is TEXTURE_FILTER_CONTROL_EXT, then the value of <pname> may be:

```
TEXTURE_LOD_BIAS_EXT                0x8501
```

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

```
MAX_TEXTURE_LOD_BIAS_EXT            0x84FD
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.8.5 "Texture Minification"

Change the first formula under "Scale Factor and Level of Detail" to read:

"The choice is governed by a scale factor p(x,y), the level of detail parameter lambda(x,y), defined as

lambda’(x,y) = log2[p(x,y)] + lodBias

where lodBias is the texture unit’s (signed) texture lod bias parameter (as described in Section 3.8.9) clamped between the positive and negative values of the implementation defined constant MAX_TEXTURE_LOD_BIAS_EXT."

-- Section 3.8.9 "Texture Environments and Texture Functions"

Change the first paragraph to read:

"The command

```
void TexEnv{if}(enum target, enum pname, T param);
void TexEnv{if}v(enum target, enum pname, T params);
```

sets parameters of the texture environment that specifies how texture values are interepreted when texturing a fragment or sets per-texture unit texture filtering parameters. The possible target parameters are TEXTURE_ENV or TEXTURE_FILTER_CONTROL_EXT. ... When target is TEXTURE_ENV, the possible environment parameters are TEXTURE_ENV_MODE and TEXTURE_ENV_COLOR. ... When target is TEXTURE_FILTER_CONTROL_EXT, the only possible texture filter parameter is TEXTURE_LOD_BIAS_EXT. TEXTURE_LOD_BIAS_EXT is set to a signed floating point value that is used to bias the level of detail parameter, lambda, as described in Section 3.8.5."

Add a final paragraph at the end of the section:

"The state required for the per-texture unit filtering parameters consists of one floating-point value."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexEnv is called with a <pname> of TEXTURE_FILTER_PARAMETER_EXT and the value of <param> or what is pointed to by <params> is not TEXTURE_LOD_BIAS_EXT.

New State

(table 6.14, p204) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_LOD_BIAS_EXT	R	GetTexEnvfv	0.0	Biases texture level of detail	3.8.9	texture

(When ARB_multitexture is supported, the TEXTURE_LOD_BIAS_EXT state is per-texture unit.)

New Implementation State

(table 6.24, p214) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_TEXTURE_LOD_BIAS_EXT	R+	GetFloatv	4.0	Maximum absolute texture lod bias	3.8.9	-

Name

EXT_texture_object

Name Strings

GL_EXT_texture_object

Version

$Date: 1995/10/03 05:39:56 $ $Revision: 1.27 $

Number

20

Dependencies

EXT_texture3D affects the definition of this extension

Overview

This extension introduces named texture objects. The only way to name a texture in GL 1.0 is by defining it as a single display list. Because display lists cannot be edited, these objects are static. Yet it is important to be able to change the images and parameters of a texture.

Issues

* Should the dimensions of a texture object be static once they are changed from zero? This might simplify the management of texture memory. What about other properties of a texture object?

No.

Reasoning

* Previous proposals overloaded the <target> parameter of many Tex commands with texture object names, as well as the original enumerated values. This proposal eliminated such overloading, choosing instead to require an application to bind a texture object, and then operate on it through the binding reference. If this constraint ultimately proves to be unacceptable, we can always extend the extension with additional binding points for editing and querying only, but if we expect to do this, we might choose to bite the bullet and overload the <target> parameters now.

* Commands to directly set the priority of a texture object and to query the resident status of a texture object are included. I feel that binding a texture object would be an unacceptable burden for these management operations. These commands also allow queries and operations on lists of texture objects, which should improve efficiency.

* GenTexturesEXT does not return a success/failure boolean because it should never fail in practice.

New Procedures and Functions

```
    void GenTexturesEXT(sizei n,
                uint* textures);

    void DeleteTexturesEXT(sizei n,
                   const uint* textures);

    void BindTextureEXT(enum target,
                uint texture);

    void PrioritizeTexturesEXT(sizei n,
                       const uint* textures,
                       const clampf* priorities);

    boolean AreTexturesResidentEXT(sizei n,
                         const uint* textures,
                         boolean* residences);

    boolean IsTextureEXT(uint texture);
```

New Tokens

Accepted by the <pname> parameters of TexParameteri, TexParameterf, TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

```
TEXTURE_PRIORITY_EXT          0x8066
```

Accepted by the <pname> parameters of GetTexParameteriv and GetTexParameterfv:

```
TEXTURE_RESIDENT_EXT          0x8067
```

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

```
TEXTURE_1D_BINDING_EXT        0x8068
TEXTURE_2D_BINDING_EXT        0x8069
TEXTURE_3D_BINDING_EXT        0x806A
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

Add the following discussion to section 3.8 (Texturing). In addition to the default textures TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT, it is possible to create named 1, 2, and 3-dimensional texture objects. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by binding an unused name to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT. This binding is accomplished by calling BindTextureEXT with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, and <texture> set to the name of the new texture object. When a texture object is bound to a target, the previous binding for that target is automatically broken.

When a texture object is first bound it takes the dimensionality of its target. Thus, a texture object first bound to TEXTURE_1D is 1-dimensional; a texture object first bound to TEXTURE_2D is 2-dimensional, and a texture object first bound to TEXTURE_3D_EXT is 3-dimensional. The state of a 1-dimensional texture object immediately after it is first bound is equivalent to the state of the default TEXTURE_1D at GL initialization. Likewise, the state of a 2-dimensional or 3-dimensional texture object immediately after it is first bound is equivalent to the state of the default TEXTURE_2D or TEXTURE_3D_EXT at GL initialization. Subsequent bindings of a texture object have no effect on its state. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object to a target of different dimensionality.

While a texture object is bound, GL operations on the target to which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object. If texture mapping of the dimensionality of the target to which a texture object is bound is active, the bound texture object is used.

By default when an OpenGL context is created, TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT have 1, 2, and 3-dimensional textures associated with them. In order that access to these default textures not be lost, this extension treats them as though their names were all zero. Thus the default 1-dimensional texture is operated on, queried, and applied as TEXTURE_1D while zero is bound to TEXTURE_1D. Likewise, the default 2-dimensional texture is operated on, queried, and applied as TEXTURE_2D while zero is bound to TEXTURE_2D, and the default 3-dimensional texture is operated on, queried, and applied as TEXTURE_3D_EXT while zero is bound to TEXTURE_3D_EXT.

Texture objects are deleted by calling DeleteTexturesEXT with <textures> pointing to a list of <n> names of texture object to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is freed. If a texture object that is currently bound is deleted, the binding reverts to zero. DeleteTexturesEXT ignores names that do not correspond to textures objects, including zero.

GenTexturesEXT returns <n> texture object names in <textures>. These names are chosen in an unspecified manner, the only condition being that only names that were not in use immediately prior to the call to GenTexturesEXT are considered. Names returned by GenTexturesEXT are marked as used (so that they are not returned by subsequent calls to GenTexturesEXT), but they are associated with a texture object only after they are first bound (just as if the name were unused).

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently being treated as a part of the working set is said to be resident. AreTexturesResidentEXT returns TRUE if all of the <n> texture objects named in <textures> are resident, FALSE otherwise. If FALSE is returned, the residence of each texture object is returned in <residences>. Otherwise the contents of the <residences> array are not changed. If any of the names in <textures> is not the name of a texture object, FALSE is returned, the error INVALID_VALUE is generated, and the contents of <residences> are indeterminate. The resident status of a single bound texture object can also be queried by calling GetTexParameteriv or GetTexParameterfv with <target> set to the target to which the texture object is bound, and <pname> set to TEXTURE_RESIDENT_EXT. This is the only way that the resident status of a default texture can be queried.

Applications guide the OpenGL implementation in determining which texture objects should be resident by specifying a priority for each texture object. PrioritizeTexturesEXT sets the priorities of the <n> texture objects in <textures> to the values in <priorities>. Each priority value is clamped to the range [0.0, 1.0] before it is assigned. Zero indicates the lowest priority, and hence the least likelihood of being resident. One indicates the highest priority, and hence the greatest likelihood of being resident. The priority of a single bound texture object can also be changed by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to the target to which the texture object is bound, <pname> set to TEXTURE_PRIORITY_EXT, and <param> or <params> specifying the new priority value (which is clamped to [0.0,1.0] before being assigned). This is the only way that the priority of a default texture can be specified. (PrioritizeTexturesEXT silently ignores attempts to prioritize nontextures, and texture zero.)

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

BindTextureEXT and PrioritizeTexturesEXT are included in display lists. All other commands defined by this extension are not included in display lists.

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

IsTextureEXT returns TRUE if <texture> is the name of a valid texture object. If <texture> is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, IsTextureEXT returns FALSE.

Because the query values of TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D_EXT are already defined as booleans indicating whether these textures are enabled or disabled, another mechanism is required to query the binding associated with each of these texture targets. The name of the texture object currently bound to TEXTURE_1D is returned in <params> when GetIntegerv is called with <pname> set to TEXTURE_1D_BINDING_EXT. If no texture object is currently bound to TEXTURE_1D, zero is returned. Likewise, the name of the texture object bound to TEXTURE_2D or TEXTURE_3D_EXT is returned in <params> when GetIntegerv is called with <pname> set to TEXTURE_2D_BINDING_EXT or TEXTURE_3D_BINDING_EXT. If no texture object is currently bound to TEXTURE_2D or to TEXTURE_3D_EXT, zero is returned.

A texture object comprises the image arrays, priority, border color, filter modes, and wrap modes that are associated with that object. More explicitly, the state list

```
TEXTURE,
TEXTURE_PRIORITY_EXT
TEXTURE_RED_SIZE,
TEXTURE_GREEN_SIZE,
TEXTURE_BLUE_SIZE,
TEXTURE_ALPHA_SIZE,
TEXTURE_LUMINANCE_SIZE,
TEXTURE_INTENSITY_SIZE,
TEXTURE_WIDTH,
TEXTURE_HEIGHT,
TEXTURE_DEPTH_EXT,
TEXTURE_BORDER,
TEXTURE_COMPONENTS,
TEXTURE_BORDER_COLOR,
TEXTURE_MIN_FILTER,
TEXTURE_MAG_FILTER,
TEXTURE_WRAP_S,
TEXTURE_WRAP_T,
TEXTURE_WRAP_R_EXT
```

composes a single texture object.

When PushAttrib is called with TEXTURE_BIT enabled, the priorities, border colors, filter modes, and wrap modes of the currently bound texture objects are pushed, as well as the current texture bindings and enables. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture objects have their priorities, border colors, filter modes, and wrap modes restored to their pushed values.

Additions to the GLX Specification

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

GLX Protocol

Six new GL commands are added.

The following rendering command is sent to the server as part of a glXRender request:

```
BindTextureEXT
    2           12              rendering command length
    2           4117            rendering command opcode
    4           ENUM            target
    4           CARD32          texture
```

The following rendering command can be sent to the server as part of a glXRender request or as part of a glXRenderLarge request:

```
PrioritizeTexturesEXT
       2           8+(n*8)         rendering command length
       2           4118            rendering command opcode
       4           INT32           n
       n*4         LISTofCARD32    textures
       n*4         LISTofFLOAT32   priorities
```

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

```
       4           12+(n*8)        rendering command length
       4           4118            rendering command opcode
```

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

```
DeleteTexturesEXT
    1       CARD8             opcode (X assigned)
    1       16                GLX opcode (glXVendorPrivate)
    2       4+n               request length
    4       12                vendor specific opcode
    4       GLX_CONTEXT_TAG   context tag
    4       INT32             n
    n*4     CARD32            textures

   GenTexturesEXT
    1       CARD8             opcode (X assigned)
    1       17                GLX opcode (glXVendorPrivateWithReply)
    2       4                 request length
    4       13                vendor specific opcode
    4       GLX_CONTEXT_TAG   context tag
    4       INT32       n
  =>
    1       1                reply
    1                        unused
    2       CARD16            sequence number
    4       n                 reply length
    24                        unused
    4*n     LISTofCARD32      textures
```

```
        AreTexturesResidentEXT
         1       CARD8             opcode (X assigned)
         1       17                GLX opcode (glXVendorPrivateWithReply)
         2       4+n               request length
         4       11                vendor specific opcode
         4       GLX_CONTEXT_TAG   context tag
         4       INT32             n
         4*n     LISTofCARD32      textures
      =>
          1       1                reply
          1                        unused
          2      CARD16            sequence number
          4      (n+p)/4           reply length
          4       BOOL32           return_value
          20                       unused
          n       LISTofBOOL       residences
          p                        unused, p=pad(n)

        IsTextureEXT
         1       CARD8             opcode (X assigned)
         1       17                GLX opcode (glXVendorPrivateWithReply)
         2       4                 request length
         4       14                vendor specific opcode
         4       GLX_CONTEXT_TAG   context tag
         4       CARD32            textures
      =>
          1       1                reply
          1                        unused
          2      CARD16            sequence number
          4      0                 reply length
          4      BOOL32            return_value
          20                       unused
```

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, then all references to 3D textures in this specification are invalid.

Errors

INVALID_VALUE is generated if GenTexturesEXT parameter <n> is negative.

INVALID_VALUE is generated if DeleteTexturesEXT parameter <n> is negative.

INVALID_ENUM is generated if BindTextureEXT parameter <target> is not TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is TEXTURE_1D, and parameter <texture> is the name of a 2-dimensional or 3-dimensional texture object.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is TEXTURE_2D, and parameter <texture> is the name of a 1-dimensional or 3-dimensional texture object.

INVALID_OPERATION is generated if BindTextureEXT parameter <target> is

TEXTURE_3D_EXT, and parameter <texture> is the name of a 1-dimensional or 2-dimensional texture object.

INVALID_VALUE is generated if PrioritizeTexturesEXT parameter <n> negative.

INVALID_VALUE is generated if AreTexturesResidentEXT parameter <n> is negative.

INVALID_VALUE is generated by AreTexturesResidentEXT if any of the names in <textures> is zero, or is not the name of a texture.

INVALID_OPERATION is generated if any of the commands defined in this extension is executed between the execution of Begin and the corresponding execution of End.

New State

Get Value	Get Command	Type	Initial Value	Attribute
TEXTURE_1D	IsEnabled	B	FALSE	texture/enable
TEXTURE_2D	IsEnabled	B	FALSE	texture/enable
TEXTURE_3D_EXT	IsEnabled	B	FALSE	texture/enable
TEXTURE_1D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_2D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_3D_BINDING_EXT	GetIntegerv	Z+	0	texture
TEXTURE_PRIORITY_EXT	GetTexParameterfv	n x Z+	1	texture
TEXTURE_RESIDENT_EXT	AreTexturesResidentEXT	n x B	unknown	-
TEXTURE	GetTexImage	n x levels x I	null	-
TEXTURE_RED_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_GREEN_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_BLUE_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_ALPHA_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_LUMINANCE_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_INTENSITY_SIZE_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_WIDTH	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_HEIGHT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_DEPTH_EXT	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_4DSIZE_SGIS	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_BORDER	GetTexLevelParameteriv	n x levels x Z+	0	-
TEXTURE_COMPONENTS (1D and 2D)	GetTexLevelParameteriv	n x levels x Z42	1	-
TEXTURE_COMPONENTS (3D and 4D)	GetTexLevelParameteriv	n x levels x Z38	LUMINANCE	-
TEXTURE_BORDER_COLOR	GetTexParameteriv	n x C	0, 0, 0, 0	texture
TEXTURE_MIN_FILTER	GetTexParameteriv	n x Z7	NEAREST_MIPMAP_LINEAR	texture
TEXTURE_MAG_FILTER	GetTexParameteriv	n x Z3	LINEAR	texture
TEXTURE_WRAP_S	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_T	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_R_EXT	GetTexParameteriv	n x Z2	REPEAT	texture
TEXTURE_WRAP_Q_SGIS	GetTexParameteriv	n x Z2	REPEAT	texture

New Implementation Dependent State

None

Name

EXT_vertex_array

Name Strings

GL_EXT_vertex_array

Version

$Date: 1995/10/03 05:39:58 $ $Revision: 1.16 $ FINAL

Number

30

Dependencies

None

Overview

This extension adds the ability to specify multiple geometric primitives with very few subroutine calls. Instead of calling an OpenGL procedure to pass each individual vertex, normal, or color, separate arrays of vertexes, normals, and colors are prespecified, and are used to define a sequence of primitives (all of the same type) when a single call is made to DrawArraysEXT. A stride mechanism is provided so that an application can choose to keep all vertex data staggered in a single array, or sparsely in separate arrays. Single-array storage may optimize performance on some implementations.

This extension also supports the rendering of individual array elements, each specified as an index into the enabled arrays.

Issues

*   Should arrays for material parameters be provided? If so, how?

A: No. Let's leave this to a separate extension, and keep this extension lean.

*   Should a FORTRAN interface be specified in this document?

*   It may not be possible to implement GetPointervEXT in FORTRAN. If not, should we eliminate it from this proposal?

A: Leave it in.

*   Should a stride be specified by DrawArraysEXT which, if non-zero, would override the strides specified for the individual arrays? This might improve the efficiency of single-array transfers.

A: No, it's not worth the effort and complexity.

*   Should entry points for byte vertexes, byte indexes, and byte texture coordinates be added in this extension?

A: No, do this in a separate extension, which defines byte support for arrays and for the current procedural interface.

* Should support for meshes (not strips) of rectangles be provided?

A: No. If this is necessary, define a separate quad_mesh extension that supports both immediate mode and arrays. (Add QUAD_MESH_EXT as an token accepted by Begin and DrawArraysEXT. Add QuadMeshLengthEXT to specify the length of the mesh.)

Reasoning

* DrawArraysEXT requires that VERTEX_ARRAY_EXT be enabled so that future extensions can support evaluation as well as direct specification of vertex coordinates.

* This extension does not support evaluation. It could be extended to provide such support by adding arrays of points to be evaluated, and by adding enables to indicate that the arrays are to be evaluated. I think we may choose to add an array version of EvalMesh, rather than extending the operation of DrawArraysEXT, so I'd rather wait on this one.

* <size> is specified before <type> to match the order of the information in immediate mode commands, such as Vertex3f. (first 3, then f)

* It seems reasonable to allow attribute values to be undefined after DrawArraysEXT executes. This avoids implementation overhead in the case where an incomplete primitive is specified, and will allow optimization on multiprocessor systems. I don't expect this to be a burden to programmers.

* It is not an error to call VertexPointerEXT, NormalPointerEXT, ColorPointerEXT, IndexPointerEXT, TexCoordPointerEXT, or EdgeFlagPointerEXT between the execution of Begin and the corresponding execution of End. Because these commands will typically be implemented on the client side with no protocol, testing for between-Begin-End status requires that the client track this state, or that a round trip be made. Neither is desirable.

* Arrays are enabled and disabled individually, rather than with a single mask parameter, for two reasons. First, we have had trouble allocating bits in masks, so eliminating a mask eliminates potential trouble down the road. We may eventually require a larger number of array types than there are bits in a mask. Second, making the enables into state eliminates a parameter in ArrayElementEXT, and may allow it to execute more efficiently. Of course this state model may result in programming errors, but OpenGL is full of such hazards anyway!

* ArrayElementEXT is provided to support applications that construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because each call specifies only a single vertex, it is possible for an application to explicitly

specify per-primitive attributes, such as a single normal per individual triangle.

- The <count> parameters are added to the *PointerEXT commands to allow implementations to cache array data, and in particular to cache the transformed results of array data that are rendered repeatedly by ArrayElementEXT. Implementations that do not wish to perform such caching can ignore the <count> parameter.

- The <first> parameter of DrawArraysEXT allows a single set of arrays to be used repeatedly, possibly improving performance.

New Procedures and Functions

```
    void ArrayElementEXT(int i);

    void DrawArraysEXT(enum mode,
                 int first,
                 sizei count);

    void VertexPointerEXT(int size,
                  enum type,
                  sizei stride,
                  sizei count,
                  const void* pointer);

    void NormalPointerEXT(enum type,
                  sizei stride,
                  sizei count,
                  const void* pointer);

    void ColorPointerEXT(int size,
                 enum type,
                 sizei stride,
                 sizei count,
                 const void* pointer);

    void IndexPointerEXT(enum type,
                 sizei stride,
                 sizei count,
                 const void* pointer);

    void TexCoordPointerEXT(int size,
                    enum type,
                    sizei stride,
                    sizei count,
                    const void* pointer);

    void EdgeFlagPointerEXT(sizei stride,
                    sizei count,
                    const Boolean* pointer);

    void GetPointervEXT(enum pname,
                 void** params);
```

New Tokens

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

VERTEX_ARRAY_EXT	0x8074
NORMAL_ARRAY_EXT	0x8075
COLOR_ARRAY_EXT	0x8076
INDEX_ARRAY_EXT	0x8077
TEXTURE_COORD_ARRAY_EXT	0x8078
EDGE_FLAG_ARRAY_EXT	0x8079

Accepted by the <type> parameter of VertexPointerEXT, NormalPointerEXT, ColorPointerEXT, IndexPointerEXT, and TexCoordPointerEXT:

DOUBLE_EXT	0x140A

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

VERTEX_ARRAY_SIZE_EXT	0x807A
VERTEX_ARRAY_TYPE_EXT	0x807B
VERTEX_ARRAY_STRIDE_EXT	0x807C
VERTEX_ARRAY_COUNT_EXT	0x807D
NORMAL_ARRAY_TYPE_EXT	0x807E
NORMAL_ARRAY_STRIDE_EXT	0x807F
NORMAL_ARRAY_COUNT_EXT	0x8080
COLOR_ARRAY_SIZE_EXT	0x8081
COLOR_ARRAY_TYPE_EXT	0x8082
COLOR_ARRAY_STRIDE_EXT	0x8083
COLOR_ARRAY_COUNT_EXT	0x8084
INDEX_ARRAY_TYPE_EXT	0x8085
INDEX_ARRAY_STRIDE_EXT	0x8086
INDEX_ARRAY_COUNT_EXT	0x8087
TEXTURE_COORD_ARRAY_SIZE_EXT	0x8088
TEXTURE_COORD_ARRAY_TYPE_EXT	0x8089
TEXTURE_COORD_ARRAY_STRIDE_EXT	0x808A
TEXTURE_COORD_ARRAY_COUNT_EXT	0x808B
EDGE_FLAG_ARRAY_STRIDE_EXT	0x808C
EDGE_FLAG_ARRAY_COUNT_EXT	0x808D

Accepted by the <pname> parameter of GetPointervEXT:

VERTEX_ARRAY_POINTER_EXT	0x808E
NORMAL_ARRAY_POINTER_EXT	0x808F
COLOR_ARRAY_POINTER_EXT	0x8090
INDEX_ARRAY_POINTER_EXT	0x8091
TEXTURE_COORD_ARRAY_POINTER_EXT	0x8092
EDGE_FLAG_ARRAY_POINTER_EXT	0x8093

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

Array Specification

Individual array pointers and associated data are maintained for an array of vertexes, an array of normals, an array of colors, an array of color indexes, an array of texture coordinates, and an array of edge flags. The data associated with each array specify the data type of the values in the array, the number of values per element in the array (e.g. vertexes of 2, 3, or 4 coordinates), the byte stride from one array element to the next, and the number of elements (counting from the first) that are static. Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for any rendering. When an array is specified, the pointer and associated data are saved as client-side state, and static elements may be cached by the implementation. Non-static (dynamic) elements are never accessed until ArrayElementEXT or DrawArraysEXT is issued.

VertexPointerEXT specifies the location and data format of an array of vertex coordinates. <pointer> specifies a pointer to the first coordinate of the first vertex in the array. <type> specifies the data type of each coordinate in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <size> specifies the number of coordinates per vertex, and must be 2, 3, or 4. <stride> specifies the byte offset between pointers to consecutive vertexes. If <stride> is zero, the vertex data are understood to be tightly packed in the array. <count> specifies the number of vertexes, counting from the first, that are static.

NormalPointerEXT specifies the location and data format of an array of normals. <pointer> specifies a pointer to the first coordinate of the first normal in the array. <type> specifies the data type of each coordinate in the array, and must be one of BYTE, SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types byte, short, int, float, and double respectively. It is understood that each normal comprises three coordinates. <stride> specifies the byte offset between pointers to consecutive normals. If <stride> is zero, the normal data are understood to be tightly packed in the array. <count> specifies the number of normals, counting from the first, that are static.

ColorPointerEXT specifies the location and data format of an array of color components. <pointer> specifies a pointer to the first component of the first color element in the array. <type> specifies the data type of each component in the array, and must be one of BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE_EXT, implying GL data types byte, ubyte, short, ushort, int, uint, float, and double respectively. <size> specifies the number of components per color, and must be 3 or 4. <stride> specifies the byte offset between pointers to consecutive colors. If <stride> is zero, the color data are understood to be tightly packed in the array. <count> specifies the number of colors, counting from the first, that are static.

IndexPointerEXT specifies the location and data format of an array of color indexes. <pointer> specifies a pointer to the first index in the array. <type> specifies the data type of each index in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <stride> specifies the byte offset between pointers to consecutive indexes. If

<stride> is zero, the index data are understood to be tightly packed in the array. <count> specifies the number of indexes, counting from the first, that are static.

TexCoordPointerEXT specifies the location and data format of an array of texture coordinates. <pointer> specifies a pointer to the first coordinate of the first element in the array. <type> specifies the data type of each coordinate in the array, and must be one of SHORT, INT, FLOAT, or DOUBLE_EXT, implying GL data types short, int, float, and double respectively. <size> specifies the number of coordinates per element, and must be 1, 2, 3, or 4. <stride> specifies the byte offset between pointers to consecutive elements of coordinates. If <stride> is zero, the coordinate data are understood to be tightly packed in the array. <count> specifies the number of texture coordinate elements, counting from the first, that are static.

EdgeFlagPointerEXT specifies the location and data format of an array of boolean edge flags. <pointer> specifies a pointer to the first flag in the array. <stride> specifies the byte offset between pointers to consecutive edge flags. If <stride> is zero, the edge flag data are understood to be tightly packed in the array. <count> specifies the number of edge flags, counting from the first, that are static.

The table below summarizes the sizes and data types accepted (or understood implicitly) by each of the six pointer-specification commands.

Command	Sizes	Types
VertexPointerEXT	2,3,4	short, int, float, double
NormalPointerEXT	3	byte, short, int, float, double
ColorPointerEXT	3,4	byte, short, int, float, double, ubyte, ushort, uint
IndexPointerEXT	1	short, int, float, double
TexCoordPointerEXT	1,2,3,4	short, int, float, double
EdgeFlagPointerEXT	1	boolean

Rendering the Arrays

By default all the arrays are disabled, meaning that they will not be accessed when either ArrayElementEXT or DrawArraysEXT is called. An individual array is enabled or disabled by calling Enable or Disable with <cap> set to appropriate value, as specified in the table below:

Array Specification Command	Enable Token
VertexPointerEXT	VERTEX_ARRAY_EXT
NormalPointerEXT	NORMAL_ARRAY_EXT
ColorPointerEXT	COLOR_ARRAY_EXT
IndexPointerEXT	INDEX_ARRAY_EXT
TexCoordPointerEXT	TEXTURE_COORD_ARRAY_EXT
EdgeFlagPointerEXT	EDGE_FLAG_ARRAY_EXT

When ArrayElementEXT is called, a single vertex is drawn, using vertex and attribute data taken from location <i> of the enabled arrays. The semantics of ArrayElementEXT are defined in the C-code below:

```
void ArrayElementEXT (int i) {
    byte* p;
    if (NORMAL_ARRAY_EXT) {
      if (normal_stride == 0)
          p = (byte*)normal_pointer + i * 3 * sizeof(normal_type);
      else
          p = (byte*)normal_pointer + i * normal_stride;
      Normal3<normal_type>v ((normal_type*)p);
    }
    if (COLOR_ARRAY_EXT) {
      if (color_stride == 0)
          p = (byte*)color_pointer +
            i * color_size * sizeof(color_type);
      else
          p = (byte*)color_pointer + i * color_stride;
      Color<color_size><color_type>v ((color_type*)p);
    }
    if (INDEX_ARRAY_EXT) {
      if (index_stride == 0)
          p = (byte*)index_pointer + i * sizeof(index_type);
      else
          p = (byte*)index_pointer + i * index_stride;
      Index<index_type>v ((index_type*)p);
    }
    if (TEXTURE_COORD_ARRAY_EXT) {
      if (texcoord_stride == 0)
          p = (byte*)texcoord_pointer +
            i * texcoord_size * sizeof(texcoord_type);
      else
          p = (byte*)texcoord_pointer + i * texcoord_stride;
      TexCoord<texcoord_size><texcoord_type>v ((texcoord_type*)p);
    }
    if (EDGE_FLAG_ARRAY_EXT) {
      if (edgeflag_stride == 0)
          p = (byte*)edgeflag_pointer + i * sizeof(boolean);
      else
          p = (byte*)edgeflag_pointer + i * edgeflag_stride;
      EdgeFlagv ((boolean*)p);
    }
    if (VERTEX_ARRAY_EXT) {
      if (vertex_stride == 0)
          p = (byte*)vertex_pointer +
            i * vertex_size * sizeof(vertex_type);
      else
          p = (byte*)vertex_pointer + i * vertex_stride;
      Vertex<vertex_size><vertex_type>v ((vertex_type*)p);
    }
}
```

ArrayElementEXT executes even if VERTEX_ARRAY_EXT is not enabled. No drawing occurs in this case, but the attributes corresponding to enabled arrays are modified.

When DrawArraysEXT is called, <count> sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with element <first>. <mode> specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. Accepted values for <mode> are POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, and POLYGON. If VERTEX_ARRAY_EXT is not enabled, no geometric primitives are generated.

The semantics of DrawArraysEXT are defined in the C-code below:

```
void DrawArraysEXT(enum mode, int first, sizei count) {
    int i;
    if (count < 0)
      /* generate INVALID_VALUE error and abort */
    else {
      Begin (mode);
      for (i=0; i < count; i++)
          ArrayElementEXT(first + i);
      End ();
    }
}
```

The ways in which the execution of DrawArraysEXT differs from the semantics indicated in the pseudo-code above are:

1. Vertex attributes that are modified by DrawArraysEXT have an unspecified value after DrawArraysEXT returns. For example, if COLOR_ARRAY_EXT is enabled, the value of the current color is undefined after DrawArraysEXT executes. Attributes that aren’t modified remain well defined.

2. Operation of DrawArraysEXT is atomic with respect to error generation. If an error is generated, no other operations take place.

Although it is not an error to respecify an array between the execution of Begin and the corresponding execution of End, the result of such respecification is undefined. Static array data may be read and cached by the implementation at any time. If static array data are modified by the application, the results of any subsequently issued ArrayElementEXT or DrawArraysEXT commands are undefined.

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

ArrayElementEXT and DrawArraysEXT are included in display lists. When either command is entered into a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client side state, their values affect display lists when the lists are created, not when the lists are executed.

Array specification commands VertexPointerEXT, NormalPointerEXT, ColorPointerEXT, IndexPointerEXT, TexCoordPointerEXT, and EdgeFlagPointerEXT specify client side state, and are therefore not included in display lists. Likewise Enable and Disable, when called with <cap> set to VERTEX_ARRAY_EXT, NORMAL_ARRAY_EXT, COLOR_ARRAY_EXT, INDEX_ARRAY_EXT, TEXTURE_COORD_ARRAY_EXT, or EDGE_FLAG_ARRAY_EXT, are not included in display lists. GetPointervEXT returns state information, and so is not included in display lists.

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

GetPointervEXT returns in <param> the array pointer value specified by <pname>. Accepted values for <pname> are VERTEX_ARRAY_POINTER_EXT, NORMAL_ARRAY_POINTER_EXT, COLOR_ARRAY_POINTER_EXT, INDEX_ARRAY_POINTER_EXT, TEXTURE_COORD_ARRAY_POINTER_EXT, and EDGE_FLAG_ARRAY_POINTER_EXT.

All array data are client side state, and are not saved or restored by PushAttrib and PopAttrib.

Additions to the GLX Specification

None

GLX Protocol

A new rendering command is added; it can be sent to the server as part of a glXRender request or as part of a glXRenderLarge request:

The DrawArraysEXT command consists of three sections, in the following order: (1) header information, (2) a list of array information, containing the type and size of the array values for each enabled array and (3) a list of vertex data. Each element in the list of vertex data contains information for a single vertex taken from the enabled arrays.

```
DrawArraysEXT
    2           16+(12*m)+(s*n)     rendering command length
    2           4116                rendering command opcode
    4           CARD32              n (number of array elements)
    4           CARD32              m (number of enabled arrays)
    4           ENUM                mode    /* GL_POINTS etc */
    12*m        LISTofARRAY_INFO
    s*n         LISTofVERTEX_DATA
```

Where s = ns + cs + is + ts + es + vs + np + cp + ip + tp + ep + vp. (See description below, under VERTEX_DATA.) Note that if an array is disabled then no information is sent for it. For example, when the normal array is disabled, there is no ARRAY_INFO record for the normal array and ns and np are both zero.

Note that the list of ARRAY_INFO is unordered: since the ARRAY_INFO record contains the array type, the arrays in the list may be stored in any order. Also, the VERTEX_DATA list is a packed list of vertices. For each vertex, data is retrieved from the enabled arrays, and stored in the list.

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

```
    4           20+(12*m)+(s*n) rendering command length
    4           4116            rendering command opcode

ARRAY_INFO
    4            ENUM                data type
          0x1400  i=1          BYTE
          0x1401  i=1          UNSIGNED_BYTE
          0x1402  i=2          SHORT
          0x1403  i=2          UNSIGNED_SHORT
          0x1404  i=4          INT
          0x1405  i=4          UNSIGNED_INT
          0x1406  i=4          FLOAT
          0x140A  i=8          DOUBLE_EXT
    4            INT32               j (number of values in array element)
    4            ENUM                array type
          0x8074  j=2/3/4      VERTEX_ARRAY_EXT
          0x8075  j=3          NORMAL_ARRAY_EXT
          0x8076  j=3/4        COLOR_ARRAY_EXT
          0x8077  j=1          INDEX_ARRAY_EXT
          0x8078  j=1/2/3/4    TEXTURE_COORD_ARRAY_EXT
          0x8079  j=1          EDGE_FLAG_ARRAY_EXT
```

For each array, the size of an array element is i*j. Some arrays (e.g., the texture coordinate array) support different data sizes; for these arrays, the size, j, is specified when the array is defined.

```
VERTEX_DATA
 if the normal array is enabled:

    ns           LISTofBYTE          normal array element
    np                               unused, np=pad(ns)

    if the color array is enabled:

    cs           LISTofBYTE          color array element
    cp                               unused, cp=pad(cs)

    if the index array is enabled:

    is           LISTofBYTE          index array element
    ip                               unused, ip=pad(is)

    if the texture coord array is enabled:

    ts           LISTofBYTE          texture coord array element
    tp                               unused, tp=pad(ts)

    if the edge flag array is enabled:

    es           LISTofBYTE          edge flag array element
    ep                               unused, ep=pad(es)

    if the vertex array is enabled:

    vs           LISTofBYTE          vertex array element
    vp                               unused, vp=pad(vs)
```

where ns, cs, is, ts, es, vs is the size of the normal, color, index, texture, edge and vertex array elements and np, cp, ip, tp, ep, vp is the padding for the normal, color, index, texture, edge and vertex array elements, respectively.

Errors

INVALID_OPERATION is generated if DrawArraysEXT is called between the execution of Begin and the corresponding execution of End.

INVALID_ENUM is generated if DrawArraysEXT parameter <mode> is not POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, or POLYGON.

INVALID_VALUE is generated if DrawArraysEXT parameter <count> is negative.

INVALID_VALUE is generated if VertexPointerEXT parameter <size> is not 2, 3, or 4.

INVALID_ENUM is generated if VertexPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if VertexPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if NormalPointerEXT parameter <type> is not BYTE, SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if NormalPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if ColorPointerEXT parameter <size> is not 3 or 4.

INVALID_ENUM is generated if ColorPointerEXT parameter <type> is not BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if ColorPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if IndexPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if IndexPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if TexCoordPointerEXT parameter <size> is not 1, 2, 3, or 4.

INVALID_ENUM is generated if TexCoordPointerEXT parameter <type> is not SHORT, INT, FLOAT, or DOUBLE_EXT.

INVALID_VALUE is generated if TexCoordPointerEXT parameter <stride> or <count> is negative.

INVALID_VALUE is generated if EdgeFlagPointerEXT parameter <stride> or <count> is negative.

INVALID_ENUM is generated if GetPointervEXT parameter <pname> is not VERTEX_ARRAY_POINTER_EXT, NORMAL_ARRAY_POINTER_EXT, COLOR_ARRAY_POINTER_EXT, INDEX_ARRAY_POINTER_EXT, TEXTURE_COORD_ARRAY_POINTER_EXT, or EDGE_FLAG_ARRAY_POINTER_EXT.

New State

Get Value	Get Command	Type	Initial Value	Attrib
VERTEX_ARRAY_EXT	IsEnabled	B	False	client
VERTEX_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
VERTEX_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
VERTEX_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
VERTEX_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
VERTEX_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
NORMAL_ARRAY_EXT	IsEnabled	B	False	client
NORMAL_ARRAY_TYPE_EXT	GetIntegerv	Z5	FLOAT	client
NORMAL_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
NORMAL_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
NORMAL_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
COLOR_ARRAY_EXT	IsEnabled	B	False	client
COLOR_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
COLOR_ARRAY_TYPE_EXT	GetIntegerv	Z8	FLOAT	client
COLOR_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
COLOR_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
COLOR_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
INDEX_ARRAY_EXT	IsEnabled	B	False	client
INDEX_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
INDEX_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
INDEX_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
INDEX_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
TEXTURE_COORD_ARRAY_EXT	IsEnabled	B	False	client
TEXTURE_COORD_ARRAY_SIZE_EXT	GetIntegerv	Z+	4	client
TEXTURE_COORD_ARRAY_TYPE_EXT	GetIntegerv	Z4	FLOAT	client
TEXTURE_COORD_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
TEXTURE_COORD_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
TEXTURE_COORD_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client
EDGE_FLAG_ARRAY_EXT	IsEnabled	B	False	client
EDGE_FLAG_ARRAY_STRIDE_EXT	GetIntegerv	Z+	0	client
EDGE_FLAG_ARRAY_COUNT_EXT	GetIntegerv	Z+	0	client
EDGE_FLAG_ARRAY_POINTER_EXT	GetPointervEXT	Z+	0	client

New Implementation Dependent State

None

Name

EXT_vertex_weighting

Name Strings

GL_EXT_vertex_weighting

Notice

Status

Shipping (version 1.0)

Version

NVIDIA Date: May 25, 2000

Number

188

Dependencies

None

Written based on the wording of the OpenGL 1.2 specification but not dependent on it.

Overview

The intent of this extension is to provide a means for blending geometry based on two slightly differing modelview matrices. The blending is based on a vertex weighting that can change on a per-vertex basis. This provides a primitive form of skinning.

A second modelview matrix transform is introduced. When vertex weighting is enabled, the incoming vertex object coordinates are transformed by both the primary and secondary modelview matrices; likewise, the incoming normal coordinates are transformed by the inverses of both the primary and secondary modelview matrices. The resulting two position coordinates and two normal coordinates are blended based on the per-vertex vertex weight and then combined by addition. The transformed, weighted, and combined vertex position and normal are then used by OpenGL as the eye-space position and normal for lighting, texture coordinate, generation, clipping, and further vertex transformation.

Issues

Should the extension be written to extend to more than two vertex weights and modelview matrices?

RESOLUTION: NO. Supports only one vertex weight and two modelview matrices. If more than two is useful, that can be handled with another extension.

Should the weighting factor be GLclampf instead of GLfloat?

RESOLUTION: GLfloat. Though the value of a weighting factors outside the range of zero to one (and even weights that do not add to one) is dubious, there is no reason to limit the implementation to values between zero and one.

Should the weights and modelview matrices be labeled 1 & 2 or 0 & 1?

RESOLUTION: 0 & 1. This is consistent with the way lights and texture units are named in OpenGL. Make GL_MODELVIEW0_EXT be an alias for GL_MODELVIEW. Note that the GL_MODELVIEW0_EXT+1 will not be GL_MODELVIEW1_EXT as is the case with GL_LIGHT0 and GL_LIGHT1.

Should there be a way to simultaneously Rotate, Translate, Scale, LoadMatrix, MultMatrix, etc. the two modelview matrices together?

RESOLUTION: NO. The application must use MatrixMode and repeated calls to keep the matrices in sync if desired.

Should the secondary modelview matrix stack be as deep as the primary matrix stack or can they be different sizes?

RESOLUTION: Must be the SAME size. This wastes a lot of memory that will be probably never be used (the modelview matrix stack must have at least 32 entries), but memory is cheap.

The value returned by MAX_MODELVIEW_STACK_DEPTH applies to both modelview matrices.

Should there be any vertex array support for vertex weights.

RESOLUTION: YES.

Should we have a VertexWeight2fEXT that takes has two weight values?

RESOLUTION: NO. The weights are always vw and 1-vw.

What is the "correct" way to blend matrices, particularly when wo is not one or the modelview matrix is projective?

RESOLUTION: While it may not be 100% correct, the extension blends the vertices based on transforming the object coordinates by both M0 and M1, but the resulting w coordinate comes from simply transforming the object coordinates by M0 and extracting the w.

Another option would be to simply blend the two sets of eye coordinates without any special handling of w. This is harder.

Another option would be to divide by w before blending the two sets of eye coordinates. This is awkward because if the weight is 1.0 with vertex weighting enabled, the result is not the same as disabling vertex weighting since EYE_LINEAR texgen is based of of the non-perspective corrected eye coordinates.

As specified, the normal weighting and combination is performed on unnormalized normals. Would the math work better if the normals were normalized before weighting and combining?

RESOLUTION: Vertex weighting of normals is after the GL_RESCALE_NORMAL step and before the GL_NORMALIZE step.

As specified, feedback and selection should apply vertex weighting if enabled. Yuck, that would mean that we need software code for vertex weighting.

RESOLUTION: YES, it should work with feedback and selection.

Sometimes it would be useful to mirror changes in both modelview matrices. For example, the viewing transforms are likely to be different, just the final modeling transforms would be different. Should there be an API support for mirroring transformations into both matrices?

RESOLUTION: NO. Such support is likely to complicate the matrix management in the OpenGL. Applications can do a Get matrix from modelview0 and then a LoadMatrix into modelview1 manually if they need to mirror things.

I also worry that if we had a mirrored matrix mode, it would double the transform concatenation work if used naively.

Many of the changes to the two modelview matrices will be the same. For example, the initial view transform loaded into each will be the same. Should there be a way to "mirror" changes to both modelview matrices?

RESOLUTION: NO. Mirroring matrix changes would complicate the driver’s management of matrices. Also, I am worried that naive users would mirror all transforms and lead to lots of redundant matrix concatenations. The most efficient way to handle the slight differences between the modelview matrices is simply to GetFloat the primary matrix, LoadMatrix the values in the secondary modelview matrix, and then perform the "extra" transform to the secondary modelview matrix.

Ideally, a glCopyMatrix(GLenum src, GLenum dst) type OpenGL command could make this more efficient. There are similiar cases where you want the modelview matrix mirrored in the texture matrix. This is not the extension to solve this minor problem.

The post-vertex weighting normal is unlikely to be normalized. Should this extension automatically enable normalization?

RESOLUTION: NO. Normalization should operate as specified. The user is responsible for enabling GL_RESCALE_NORMAL or GL_NORMALIZE as needed.

You could imagine cases where the application only sent vertex weights of either zero or one and pre-normalized normals so that GL_NORMALIZE would not strictly be required.

Note that the vertex weighting of transformed normals occurs BEFORE normalize and AFTER rescaling. See the issue below for why this can make a difference.

How does vertex weighting interact with OpenGL 1.2’s GL_RESCALE_NORMAL enable?

RESOLUTION: Vertex weighting of transformed normals occurs BEFORE normalize and AFTER rescaling.

OpenGL 1.2 permits normal rescaling to behave just like normalize and because normalize immediately follows rescaling, enabling rescaling can be implementied by simply always enabling normalize.

Vertex weighting changes this. If one or both of the modelview matrices has a non-uniform scale, it may be useful to enable rescaling and normalize and this operates differently than simply enabling normalize. The difference is that rescaling occurs before the normal vertex weighting.

An implementation that truly treated rescaling as a normalize would support both a pre-weighting normalize and a post-weighting normalize. Arguably, this is a good thing.

For implementations that perform simply rescaling and not a full normalize to implement rescaling, the rescaling factor can be concatenated into each particular inverse modelview matrix.

New Procedures and Functions

```
void VertexWeightfEXT(float weight);

void VertexWeightfvEXT(float *weight);

void VertexWeightPointerEXT(int size, enum type,
                            sizei stride, void *pointer);
```

New Tokens

Accepted by the <target> parameter of Enable:

VERTEX_WEIGHTING_EXT	0x8509

Accepted by the <mode> parameter of MatrixMode:

MODELVIEW0_EXT	0x1700	(alias to MODELVIEW enumerant)
MODELVIEW1_EXT	0x850A	

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

VERTEX_WEIGHTING_EXT		
MODELVIEW0_EXT		
MODELVIEW1_EXT		
MODELVIEW0_MATRIX_EXT	0x0BA6	(alias to MODELVIEW_MATRIX)
MODELVIEW1_MATRIX_EXT	0x8506	
CURRENT_VERTEX_WEIGHT_EXT	0x850B	
VERTEX_WEIGHT_ARRAY_EXT	0x850C	
VERTEX_WEIGHT_ARRAY_SIZE_EXT	0x850D	
VERTEX_WEIGHT_ARRAY_TYPE_EXT	0x850E	
VERTEX_WEIGHT_ARRAY_STRIDE_EXT	0x850F	
MODELVIEW0_STACK_DEPTH_EXT	0x0BA3	(alias to MODELVIEW_STACK_DEPTH)
MODELVIEW1_STACK_DEPTH_EXT	0x8502	

Accepted by the <pname> parameter of GetPointerv:

VERTEX_WEIGHT_ARRAY_POINTER_EXT	0x8510

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

-- Section 2.6. 2nd paragraph changed:

"Each vertex is specified with two, three, or four coordinates. In addition, a current normal, current texture coordinates, current color, and current vertex weight may be used in processing each vertex."

-- Section 2.6. New paragraph after the 3rd paragraph:

"A vertex weight is associated with each vertex. When vertex weighting is enabled, this weight is used as a blending factor to blend the position and normals transformed by the primary and secondary modelview matrix transforms. The vertex weighting functionality takes place completely in the "vertex / normal transformation" stage of Figure 2.2."

-- Section 2.6.3. First paragraph changed to

"The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, and texture coordinates (Vertex, Color, VertexWeightEXT, Index, Normal, TexCoord)..."

-- Section 2.7. New paragraph after the 4th paragraph:

"The current vertex weight is set using

```
void VertexWeightfEXT(float weight);
void VertexWeightfvEXT(float *weight);
```

This weight is used when vertex weighting is enabled."

-- Section 2.7. The last paragraph changes from

"... and one floating-point value to store the current color index."

to:

"... one floating-point number to store the vertex weight, and one floating-point value to store the current color index."

-- Section 2.8. Change 1st paragraph to say:

"The client may specify up to seven arrays: one each to store edge flags, texture coordinates, colors, color indices, vertex weights, normals, and vertices. The commands"

Add to functions listed following first paragraph:

```
void VertexWeightPointerEXT(int size, enum type,
                            sizei stride, void *pointer);
```

Add to table 2.4 (p. 22):

Command	Sizes	Types
VertexWeightPointerEXT	1	float

Starting with the second paragraph on p. 23, change to add VERTEX_WEIGHT_ARRAY_EXT:

"An individual array is enabled or disabled by calling one of

```
void EnableClientState(enum array)
void DisableClientState(enum array)
```

with array set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY, INDEX_ARRAY, VERTEX_ARRAY_WEIGHT_EXT, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, color, secondary color, color index, normal, or vertex array, respectively.

The ith element of every enabled array is transferred to the GL by calling

```
void ArrayElement(int i)
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element i. For the vertex array, the corresponding command is Vertex<size><type>v, where <size> is one of [2,3,4], and <type> is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, color, secondary color, color index, and normal arrays are EdgeFlagv, TexCoord<size><type>v, Color<size><type>v, Index<type>v, VertexWeightfvEXT, and Normal<type>v, respectively..."

Change pseudocode on p. 27 to disable vertex weight array for canned interleaved array formats. After the lines

```
DisableClientState(EDGE_FLAG_ARRAY);
DisableClientState(INDEX_ARRAY);
```

insert the line

```
        DisableClientState(VERTEX_WEIGHT_ARRAY_EXT);
```

Substitute "seven" for every occurrence of "six" in the final paragraph on p. 27.

-- Section 2.10. Change the sentence:

"The model-view matrix is applied to these coordinates to yield eye coordinates."

to:

"The primary modelview matrix is applied to these coordinates to yield eye coordinates. When vertex weighting is enabled, a secondary modelview matrix is also applied to the vertex coordinates, the result of the two modelview transformations are weighted by its respective vertex weighting factor and combined by addition to yield the true eye coordinates. Vertex weighting is enabled or disabled using Enable and Disable (see section 2.10.3) with an argument of VERTEX_WEIGHTING_EXT."

Change the 4th paragraph to:

"If vertex weighting is disabled and a vertex in object coordinates is given by (xo yo zo wo)’ and the primary model-view matrix is M0, then the vertex’s eye coordinates are found as

```
   (xe ye ze we)’  =  M0 (xo yo zo wo)’
```

If vertex weighting is enabled, then the vertex’s eye coordinates are found as

```
   (xe0 ye0 ze0 we0)’  =  M0 (xo yo zo wo)’

   (xe1 ye1 ze1 we1)’  =  M1 (xo yo zo wo)’

   (xe,ye,ze)’ = vw*(xe0,ye0,ze0)’ + (1-vw) * (xe1,ye1,ze1)’

   we = we0
```

where M1 is the secondary modelview matrix and vw is the current vertex weight."

-- Section 2.10.2 Change the 1st paragraph to say:

"The projection matrix and the primary and secondary modelview matrices are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

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

which takes one of the four pre-defined constants TEXTURE, MODELVIEW0, MODELVIEW1, or PROJECTION (note that MODELVIEW is an alias for MODELVIEW0). TEXTURE is described later. If the current matrix is MODELVIEW0, then matrix operations apply to the primary modelview matrix; if MODELVIEW1, then matrix operations apply to the secondary modelview matrix; if PROJECTION, then they apply to the projection matrix."

Change the 9th paragraph to say:

"There is a stack of matrices for each of the matrix modes. For the MODELVIEW0 and MODELVIEW1 modes, the stack is at least 32 (that is, there is a stack of at least 32 modelview matrices). ..."

Change the last paragraph to say:

"The state required to implement transformations consists of a four-valued integer indicating the current matrix mode, a stack of at least two 4x4 matrices for each of PROJECTION and TEXTURE with associated stack pointers, and two stacks of at least 32 4x4 matrices with an associated stack pointer for MODELVIEW0 and MODELVIEW1. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW0."

-- Section 2.10.3 Change the 2nd and 7th paragraphs to say:

"For a modelview matrix M, the normal for this matrix is transformed to eye coordinates by:

```
   (nx’ ny’ nz’ q’) = (nx ny nz q) * M^-1
```

where, if (x y z w)’ are the associated vertex coordinates, then

```
       /  0,                      w= 0
       |
   q = |  -(nx ny nz) (x y z)’                             (2.1)
       |  --------------------,  w != 0
       \          w
```

Implementations may choose instead to transform (x y z)’ to eye coordinates using

```
   (nx’ ny’ nz’) = (nx ny nz) * Mu^-1
```

Where Mu is the upper leftmost 3x3 matrix taken from M.

Rescale multiplies the transformed normals by a scale factor

```
   ( nx" ny" nz" ) = f (nx’ ny’ nz’)
```

If rescaling is disabled, then f = 1. If rescaling is enabled, then f is computed as (mij denotes the matrix element in row i and column j of M^-1, numbering the topmost row of the matrix as row 1 and the leftmost column as column 1

```
                         1
          f =    ---------------------------
                 sqrt(m31^2 + m32^2 + m33^2)
```

Note that if the normals sent to GL were unit length and the model-view matrix uniformly scales space, the rescale make sthe transformed normals unit length.

Alternatively, an implementation may chose f as

```
                          1
          f =     --------------------------
                  sqrt(nx'^2 + ny'^2 + nz'^2)
```

recomputing f for each normal. This makes all non-zero length normals unit length regardless of their input length and the nature of the modelview matrix.

After rescaling, the final transformed normal used in lighting, nf, depends on whether vertex weighting is enabled or not.

When vertex weighting is disabled, nf is computed as

```
       nf = m * ( nx"0  ny"0  nz"0 )
```

where (nx"0 ny"0 nz"0) is the normal transformed as described above using the primary modelview matrix for M.

If normalization is enabled m=1. Otherwise

```
                          1
          m =     ------------------------------
                  sqrt(nx"0^2 + ny"0^2 + nz"0^2)
```

However when vertex weighting is enabled, the normal is transformed twice as described above, once by the primary modelview matrix and again by the secondary modelview matrix, weighted using the current per-vertex weight, and normalized. So nf is computed as

```
       nf = m * ( nx"w  ny"w  nz"w )
```

where nw is the weighting normal computed as

```
       nw = vw * ( nx"0  ny"0  nz"0 ) + (1-vw) * (nx"1 ny"1 nz"1)
```

where (nx"0 ny"0 nz"0) is the normal transformed as described above using the primary modelview matrix for M, and (nx"1 ny"1 nz"1) is the normal transformed as described above using the secondary modelview matrix for M, and vw is the current pver-vertex weight."

-- Section 2.12. Changes the 3rd paragraph:

"The coordinates are treated as if they were specified in a Vertex command. The x, y, z, and w coordinates are transformed by the current primary modelview and perspective matrices. These coordinates, along with current values, are used to generate a color and texture coordinates just as done for a vertex, except that vertex weighting is always treated as if it is disabled."

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

A new GL rendering command is added. The following command is sent to the server as part of a glXRender request:

```
VertexWeightfvEXT
    2  8        rendering command length
    2  4135     rendering command opcode
    4  FLOAT32  weight0
```

To support vertex arrays, the DrawArrays rendering command (sent via a glXRender or glXRenderLarge request) is amended as follows:

The list of arrays listed for the third element in the ARRAY_INFO structure is amended to include:

```
0x850c         j=1        VERTEX_WEIGHT_ARRAY_EXT
```

The VERTEX_DATA description is amended to include:

```
If the vertex weight array is enabled:
ws            LISTofBYTE        vertex weight array element
wp                              unused, wp=pad(ws)
```

with the following paragraph amended to read:

"where ns, cs, is, ts, es, vs, ws is the size of the normal, color, index, texture, edge, vertex, and vertex weight array elements and np, cp, ip, tp, ep, vp, wp is the padding for the normal, color, index, texture, edge, vertex, and vertex weight array elements, respectively."

Errors

The current vertex weight can be updated at any time. In particular WeightVertexEXT can be called between a call to Begin and the corresponding call to End.

INVALID_VALUE is generated if VertexWeightPointerEXT parameter <size> is not 1.

INVALID_ENUM is generated if VertexWeightPointerEXT parameter <type> is not FLOAT.

INVALID_VALUE is generated if VertexWeightPointerEXT parameter <stride> is negative.

New State

(table 6.5, p196)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
CURRENT_VERTEX_WEIGHT_EXT	F	GetFloatv	1	Current vertex weight	2.8	current

(table 6.6, p197)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
VERTEX_WEIGHT_ARRAY_EXT	B	IsEnabled	False	Vertex weight enable	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_SIZE_EXT	Z+	GetIntegerv	1	Weights per vertex	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_TYPE_EXT	Z1	GetIntegerv	FLOAT	Type of weights	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_STRIDE_EXT	Z	GetIntegerv	0	Stride between weights	2.8	vertex-array
VERTEX_WEIGHT_ARRAY_POINTER_EXT	Y	GetPointerv	0	Pointer to vertex weight array	2.8	vertex-array

(table 6.7, p198)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
MODELVIEW0_MATRIX_EXT	32*xM4	GetFloatv	Identity	Primary modelview stack	2.10.2	-
MODELVIEW1_MATRIX_EXT	32*xM4	GetFloatv	Identity	Secondary modelview stack	2.10.2	-
MODELVIEW0_STACK_DEPTH_EXT	Z+	GetIntegerv	1	Primary modelview stack depth	2.10.2	-
MODELVIEW1_STACK_DEPTH_EXT	Z+	GetIntegerv	1	Secondary modelview stack depth	2.10.2	-
MATRIX_MODE	Z4	GetIntegerv	MODELVIEW0	Current matrix mode	2.10.2	transform
VERTEX_WEIGHTING_EXT	B	IsEnabled	False	Vertex weighting on/off	2.10.2	transform/enable

NOTE: MODELVIEW_MATRIX is an alias for MODELVIEW0_MATRIX_EXT
MODELVIEW_STACK_DEPTH is an alias for MODELVIEW0_STACK_DEPTH_EXT

New Implementation Dependent State

None

Revision History

12/16/2000 amended to include GLX protocol for vertex arrays
5/25/2000 added missing MODELVIEW#_MATRIX tokens values

Name

NV_blend_square

Name Strings

GL_NV_blend_square

Version

Date: 8/7/1999 Version: 1.0

Number

194

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

It is useful to be able to multiply a number by itself in the blending stages -- for example, in certain types of specular lighting effects where a result from a dot product needs to be taken to a high power.

This extension provides four additional blending factors to permit this and other effects: SRC_COLOR and ONE_MINUS_SRC_COLOR for source blending factors, and DST_COLOR and ONE_MINUS_DST_COLOR for destination blending factors.

New Procedures and Functions

None

New Tokens

None

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

Two lines are added to each of tables 4.1 and 4.2:

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs, Gs, Bs, As)	*NEW*
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)	*NEW*
DST_COLOR	(Rd, Gd, Bd, Ad)	
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)	
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR	(Rc, Gc, Bc, Ac)	
ONE_MINUS_CONSTANT_COLOR	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	
CONSTANT_ALPHA	(Ac, Ac, Ac, Ac)	
ONE_MINUS_CONSTANT_ALPHA	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	
SRC_ALPHA_SATURATE	(f, f, f, 1)	

Table 4.1: Values controlling the source blending function and the source blending values they compute. f = min(As, 1 - Ad).

Value	Blend Factors	
ZERO	(0, 0, 0, 0)	
ONE	(1, 1, 1, 1)	
SRC_COLOR	(Rs, Gs, Bs, As)	
ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)	
DST_COLOR	(Rd, Gd, Bd, Ad)	*NEW*
ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)	*NEW*
SRC_ALPHA	(As, As, As, As) / Ka	
ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As) / Ka	
DST_ALPHA	(Ad, Ad, Ad, Ad) / Ka	
ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad) / Ka	
CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)	
ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)	
CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)	
ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)	

Table 4.2: Values controlling the destination blending function and the destination blending values they compute.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

None

New State

(table 6.15, page 205)

Get Value	Type	Get Command	Initial Value	Sec	Attribute
BLEND_SRC	Z15	GetIntegerv	ONE	4.1.6	color-buffer
BLEND_DST	Z14	GetIntegerv	ZERO	4.1.6	color-buffer

NOTE: the only change is that Z13 changes to Z15 and Z12 changes to Z14

New Implementation Dependent State

None

Name

NV_fence

Name Strings

GL_NV_fence

Contact

John Spitzer, NVIDIA Corporation (jspitzer ’at’ nvidia.com)

Notice

IP Status

NVIDIA Proprietary.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation. The receipt or possession of this document does not convey any express or implied rights to reproduce, disclose, distribute or prepare deivative works its contents or to manufacture, use, sell or import anything that it may describe in whole or in part. The document is provided as is with no express or implied representation or warranty of any kind as the accuracy of the information, its fitness for a particular purpose or otherwise.

Status

Shipping as of June 8, 2000 (version 1.0)

Version

July 25, 2000 (version 1.0)
$Date$ $Revision$

Number

??

Dependencies

None

Overview

The goal of this extension is provide a finer granularity of synchronizing GL command completion than offered by standard OpenGL, which offers only two mechanism for synchronization: Flush and Finish. Since Flush merely assures the user that the commands complete in a finite (though undetermined) amount of time, it is, thus, of only modest utility. Finish, on the other hand, stalls CPU execution until all pending GL commands have completed. This extension offers a middle ground - the ability to "finish" a subset of the command stream, and the ability to determine whether a given command has completed or not.

This extension introduces the concept of a "fence" to the OpenGL command stream. Once the fence is inserted into the command stream, it can be queried for a given condition - typically, its completion. Moreover, the application may also request a partial Finish -- that is, all commands prior to the fence will be forced to complete until control is returned to the calling process. These new mechanisms allow for synchronization between the host CPU and the GPU, which may be accessing the same resources (typically memory).

This extension is useful in conjunction with NV_vertex_array_range to determine when vertex information has been pulled from the vertex array range. Once a fence has been tested TRUE or finished, all vertex indices issued before the fence must have been pulled. This ensures that the vertex data memory corresponding to the issued vertex indices can be safely modified (assuming no other outstanding vertex indices are issued subsequent to the fence).

Issues

Do we need an IsFenceNV command?

RESOLUTION: Yes. Not sure who would use this, but it’s in there. Semantics currently follow the texture object definition -- that is, calling IsFenceNV before SetFenceNV will return FALSE.

Are the fences sharable between multiple contexts?

RESOLUTION: No.

Potentially this could change with a subsequent extension.

What other conditions will be supported?

Only ALL_COMPLETED_NV will be supported initially. Future extensions may wish to implement additional fence conditions.

What is the relative performance of the calls?

Execution of a SetFenceNV is not free, but will not trigger a Flush or Finish.

Is the TestFenceNV call really necessary? How often would this be used compared to the FinishFenceNV call (which also flushes to ensure this happens in finite time)?

It is conceivable that a user may use TestFenceNV to decide which portion of memory should be used next without stalling the CPU. An example of this would be a scenario where a single AGP buffer is used for both static (unchanged for multiple frames) and dynamic (changed every frame) data. If the user has written dynamic data to all banks dedicated to dynamic data, and still has more dynamic objects to write, the user would first want to check if the first dynamic object has completed, before writing into the buffer. If the object has not completed, instead of stalling the CPU with a FinishFenceNV call, it would possibly be better to start overwriting static objects instead.

What should happen if TestFenceNV is called for a name before SetFenceNV is called?

We should probably generate an error, and return TRUE. This follows the semantics for texture object names before they are bound, in that they acquire their state upon binding. We will arbitrarily return TRUE for consistency.

What should happen if FinishFenceNV is called for a name before SetFenceNV is called?

RESOLUTION: Generate an INVALID_OPERATION error because the fence id does not exist yet. SetFenceNV must be called to create a fence.

Do we need a mechanism to query which condition a given fence was set with?

RESOLUTION: Yes, use glGetFenceivNV with FENCE_CONDITION_NV.

Should we allow these commands to be compiled within display list? Which ones? How about within Begin/End pairs?

RESOLUTION: DeleteFencesNV, GenFencesNV, TestFenceNV, and IsFenceNV are executed immediately while FinishFenceNV and SetFenceNV are compiled. Do not allow any of these commands within Begin/End pairs.

Can fences be used as a form of performance monitoring?

Yes, with some caveats. By setting and testing or finishing fences, developers can measure the GPU latency for completing GL operations. For example, developers might do the following:

```
start = getCurrentTime();
updateTextures();
glSetFenceNV(TEXTURE_LOAD_FENCE, GL_ALL_COMPLETED_NV);
drawBackground();
glSetFenceNV(DRAW_BACKGROUND_FENCE, GL_ALL_COMPLETED_NV);
drawCharacters();
glSetFenceNV(DRAW_CHARACTERS_FENCE, GL_ALL_COMPLETED_NV);

glFinishFenceNV(TEXTURE_LOAD_FENCE);
textureLoadEnd = getCurrentTime();

glFinishFenceNV(DRAW_BACKGROUND_FENCE);
drawBackgroundEnd = getCurrentTime();

glFinishFenceNV(DRAW_CHARACTERS_FENCE);
drawCharactersEnd = getCurrentTime();

printf("texture load time = %d\n", textureLoadEnd - start);
printf("draw background time = %d\n", drawBackgroundEnd - textureLoadEnd);
printf("draw characters time = %d\n", drawCharacters - drawBackgroundEnd);
```

Note that there is a small amount of overhead associated with inserting each fence into the GL command stream. Each fence causes the GL command stream to momentarily idle (idling the entire GPU pipeline). The significance of this idling should be small if there are a small nuber of fences and large amount of intervening commands.

If the time between two fences is zero or very near zero, it probably means that a GPU-CPU synchronization such as a glFinish probably occurred. A glFinish is an explicit GPU-CPU synchronization, but sometimes implicit GPU-CPU synchronizations are performed by the driver.

New Procedures and Functions

```
void GenFencesNV(sizei n, uint *fences);

void DeleteFencesNV(sizei n, const uint *fences);

void SetFenceNV(uint fence, enum condition);

boolean TestFenceNV(uint fence);

void FinishFenceNV(uint fence);

boolean IsFenceNV(uint fence);

void GetFenceivNV(uint fence, enum pname, int *params);
```

New Tokens

Accepted by the <condition> parameter of SetFenceNV:

ALL_COMPLETED_NV	0x84F2

Accepted by the <pname> parameter of GetFenceivNV:

FENCE_STATUS_NV	0x84F3
FENCE_CONDITION_NV	0x84F4

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

Add to the end of Section 5.4 "Display Lists"

"DeleteFencesNV, GenFencesNV, GetFenceivNV, TestFenceNV, and IsFenceNV are not complied into display lists but are executed immediately."

After the discussion of Flush and Finish (Section 5.5) add a description of the fence operations:

"5.X Fences

The command

```
void SetFenceNV(uint fence, enum condition);
```

sets a fence within the GL command stream, and assigns the fence a status of FALSE and a condition as set by the condition argument. The condition argument must be ALL_COMPLETED_NV. Once the fence’s condition is satisfied within the command stream, its state is changed to TRUE. For a condition of ALL_COMPLETED_NV, this is completion of the fence command. No other state is affected by execution of the fence command. A fence’s state can be queried by calling the command

```
boolean TestFenceNV(uint fence);
```

The command

```
void FinishFenceNV(uint fence);
```

forces all GL commands prior to the fence to satisfy the condition set within SetFenceNV, which, in this spec, is always completion. FinishFenceNV does not return until all effects from these commands on GL client and server state and the framebuffer are fully realized.

The fence must first be created before it can be used. The command

```
void GenFencesNV(sizei n, uint *fences);
```

returns n previously unused fence names in fences. These names are marked as used, for the purposes of GenFencesNV only, but acquire boolean state only when they have been set.

Fences are deleted by calling

```
void DeleteFencesNV(sizei n, const uint *fences);
```

fences contains n names of fences to be deleted. After a fence is deleted, it has no state, and its name is again unused. Unused names in fences are silently ignored.

If the fence passed to TestFenceNV or FinishFenceNV is not the name of a fence, the error INVALID_OPERATION is generated. In this case, TestFenceNV will return TRUE, for the sake of consistency.

State must be maintained to indicate which fence integers are currently used or set. In the initial state, no indices are in use. When a fence integer is set, the condition and status of the fence are also maintained. The status is a boolean. The condition is the value last set as the condition by SetFenceNV.

Once the status of a fence has been finished (via FinishFenceNV) or tested and the returned status is TRUE (via either TestFenceNV or GetFenceivNV querying the FENCE_STATUS_NV), the status remains TRUE until the next SetFenceNV of the fence."

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

Insert new section after Section 6.1.10 "Minmax Query"

"6.1.11 Fence Query

The command

```
boolean IsFenceNV(uint fence);
```

return TRUE if texture is the name of a fence. If fence is not the name of a fence, or if an error condition occurs, IsFenceNV returns FALSE. A name returned by GenFencesNV, but not yet set via SetFenceNV, is not the name of a fence.

The command

```
void GetFenceivNV(uint fence, enum pname, int *params)
```

obtains the indicated fence state for the specified fence in the array params. pname must be either FENCE_STATUS_NV or FENCE_CONDITION_NV. The INVALID_OPERATION error is generated if the named fence does not exist."

Additions to the GLX Specification

None

GLX Protocol

Seven new GL commands are added.

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

SetFenceNV

2	12	rendering command length
2	????	rendering command opcode
4	CARD32	fence
4	CARD32	condition

FinishFenceNV

2	8	rendering command length
2	????	rendering command opcode
4	CARD32	fence

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

DeleteFencesNV

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

GenFencesNV

1	CARD8	opcode (X assigned)
1	17	GLX opcode (glXVendorPrivateWithReply)
2	4	request length
4	????	vendor specific opcode
4	GLX_CONTEXT_TAG	context tag
4	INT32	n

```
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           n               reply length
        24                          unused
        n*4         LISTofCARD322   fences

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

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

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

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

```
        4           INT32           params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofINT32     params
```

Note that polling with TestFenceNV when using indirect GLX rendering will be considerably less efficient than using FinishFenceNV because TestFenceNV is an X protocol round-trip while FinishFenceNV synchronizes the GLX command stream without an X protocol round-trip.

Errors

INVALID_VALUE is generated if GenFencesNV parameter <n> is negative.

INVALID_VALUE is generated if DeleteFencesNV parameter <n> is negative.

INVALID_OPERATION is generated if the fence used in TestFenceNV or FinishFenceNV is not the name of a fence.

INVALID_ENUM is generated if the condition used in SetFenceNV is not ALL_COMPLETED_NV.

INVALID_OPERATION is generated if any of the commands defined in this extension is executed between the execution of Begin and the corresponding execution of End.

INVALID_OPERATION is generated if the named fence in GetFenceivNV does not exist.

New State

Table 6.X. Fence Objects.

Get value	Type	Get command	Initial value	Description	Section	Attribute
FENCE_STATUS_NV	B	GetFenceivNV	determined by 1st SetFenceNV	Fence status	5.X	-
FENCE_CONDITION_NV	Z1	GetFenceivNV	determined by 1st SetFenceNV	Fence condition	5.X	-

New Implementation Dependent State

None

GeForce Implementation Details

This section describes implementation-defined limits for GeForce:

SetFenceNV calls are not free. They should be used prudently, and a "good number" of commands should be sent between calls to SetFenceNV. Each fence insertion will cause the GPU’s command processing to go momentarily idle. Testing or finishing a fence may require an one or more somewhat expensive uncached reads.

Do not leave a fence untested or unfinished for an extremely large interval of intervening fences. If more than approximately 2 billion (specifically 2^31-1) intervening fences are inserted into the GL command stream before a fence is tested or finished, said fence may indicate an incorrect status. Note that certain GL operations involving display lists, compiled vertex arrays, and textures may insert fences implicitly for internal driver use.

In practice, this limitation is unlikely to be a practical limitation if fences are finished or tested within a few frames of their insertion into the GL command stream.

Revision History

None

Name

NV_fog_distance

Name Strings

GL_NV_fog_distance

Notice

IP Status

NVIDIA Proprietary.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation. The receipt or possession of this document does not convey any express or implied rights to reproduce, disclose, distribute or prepare deivative works its contents or to manufacture, use, sell or import anything that it may describe in whole or in part. The document is provided as is with no express or implied representation or warranty of any kind as the accuracy of the information, its fitness for a particular purpose or otherwise.

Status

Shipping (version 1.0)

Version

NVIDIA Date: July 27, 2000

Number

192

Dependencies

Written based on the wording of the OpenGL 1.2 specification.

Overview

Ideally, the fog distance (used to compute the fog factor as described in Section 3.10) should be computed as the per-fragment Euclidean distance to the fragment center from the eye. In practice, implementations "may choose to approximate the eye-coordinate distance from the eye to each fragment center by abs(ze). Further, [the fog factor] f need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are."

This extension provides the application specific control over how OpenGL computes the distance used in computing the fog factor.

The extension supports three fog distance modes: "eye plane absolute", where the fog distance is the absolute planar distance from the eye plane (i.e., OpenGL’s standard implementation allowance as cited above);

"eye plane", where the fog distance is the signed planar distance from the eye plane; and "eye radial", where the fog distance is computed as a Euclidean distance. In the case of the eye radial fog distance mode, the distance may be computed per-vertex and then interpolated per-fragment.

The intent of this extension is to provide applications with better control over the tradeoff between performance and fog quality. The "eye planar" modes (signed or absolute) are straightforward to implement with good performance, but scenes are consistently under-fogged at the edges of the field of view. The "eye radial" mode can provide for more accurate fog at the edges of the field of view, but this assumes that either the eye radial fog distance is computed per-fragment, or if the fog distance is computed per-vertex and then interpolated per-fragment, then the scene must be sufficiently tessellated.

Issues

What should the default state be?

IMPLEMENTATION DEPENDENT.

The EYE_PLANE_ABSOLUTE_NV mode is the most consistent with the way most current OpenGL implementations are implemented without this extension, but because this extension provides specific control over a capability that core OpenGL is intentionally lax about, the default fog distance mode is left implementation dependent. We would not want a future OpenGL implementation that supports fast EYE_RADIAL_NV fog distance to be stuck using something less.

Advice: If an implementation can provide fast per-pixel EYE_RADIAL_NV support, then EYE_RADIAL_NV is the ideal default, but if not, then EYE_PLANE_ABSOLUTE_NV is the most reasonable default mode.

How does this extension interact with the EXT_fog_coord extension?

If FOG_COORDINATE_SOURCE_EXT is set to FOG_COORDINATE_EXT, then the fog distance mode is ignored. However, the fog distance mode is used when the FOG_COORDINATE_SOURCE_EXT is set to FRAGMENT_DEPTH_EXT. Essentially, when the EXT_fog_coord functionality is enabled, the fog distance is supplied by the user-supplied fog-coordinate so no automatic fog distance computation is performed.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameters of Fogf, Fogi, Fogfv, Fogiv, GetBooleanv, GetIntegerv, GetFloatv, and GetDoublev:

```
    FOG_DISTANCE_MODE_NV                0x855A
```

When the <pname> parameter of Fogf, Fogi, Foggv, and Fogiv, is

FOG_DISTANCE_MODE_NV, then the value of <param> or the value pointed to by <params> may be:

```
EYE_RADIAL_NV                      0x855B
EYE_PLANE
EYE_PLANE_ABSOLUTE_NV              0x855C
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Section 3.10 "Fog"

Add to the end of the 3rd paragraph:

"If pname is FOG_DISTANCE_MODE_NV, then param must be, or params must point to an integer that is one of the symbolic constants EYE_PLANE_ABSOLUTE_NV, EYE_PLANE, or EYE_RADIAL_NV and this symbolic constant determines how the fog distance should be computed."

Replace the 4th paragraph beginning "An implementation may choose to approximate ..." with:

"When the fog distance mode is EYE_PLANE_ABSOLUTE_NV, the fog distance z is approximated by abs(ze) [where ze is the Z component of the fragment’s eye position]. When the fog distance mode is EYE_PLANE, the fog distance z is approximated by ze. When the fog distance mode is EYE_RADIAL_NV, the fog distance z is computed as the Euclidean distance from the center of the fragment in eye coordinates to the eye position. Specifically:

```
  z  =  sqrt( xe*xe + ye*ye + ze*ze );
```

In the EYE_RADIAL_NV fog distance mode, the Euclidean distance is permitted to be computed per-vertex, and then interpolated per-fragment."

Change the last paragraph to read:

"The state required for fog consists of a three valued integer to select the fog equation, a three valued integer to select the fog distance mode, three floating-point values d, e, and s, and RGBA fog color and a fog color index, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_MODE is EXP, FOG_DISTANCE_NV is implementation defined, d = 1.0, e = 1.0, and s = 0.0; Cf = (0,0,0,0) and if = 0."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when Fog is called with a <pname> of FOG_DISTANCE_MODE_NV and the value of <param> or what is pointed to by <params> is not one of EYE_PLANE_ABSOLUTE_NV, EYE_PLANE, or EYE_RADIAL_NV.

New State

(table 6.8, p198) add the entry:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
FOG_DISTANCE_MODE_NV	Z3	GetIntegerv	implementation dependent	Determines how fog distance is computed	3.10	fog

New Implementation State

None

Name

NV_light_max_exponent

Name Strings

GL_NV_light_max_exponent

Notice

Version

May 20, 1999

Number

189

Dependencies

None

Overview

Default OpenGL does not permit a shininess or spot exponent over 128.0. This extension permits implementations to support and advertise a maximum shininess and spot exponent beyond 128.0.

Note that extremely high exponents for shininess and/or spot light cutoff will require sufficiently high tessellation for acceptable lighting results.

Paul Deifenbach’s thesis suggests that higher exponents are necessary to approximate BRDFs with per-vertex ligthing and multiple passes.

New Procedures and Functions

None

New Tokens

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

MAX_SHININESS_NV	0x8504
MAX_SPOT_EXPONENT_NV	0x8505

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

In Table 2.7, change the srm range entry to read:

"(range: [0.0, value of MAX_SHININESS_NV])"

In Table 2.7, change the srli range entry to read:

"(range: [0.0, value of MAX_SPOT_EXPONENT_NV])"

Add to the end of the second paragraph in Section 2.13.2:

"The values of MAX_SHININESS_NV and MAX_SPOT_EXPONENT_NV are implementation dependent, but must be equal or greater than 128."

Additions to Chapter 3 of the GL Specification (Rasterization)

None

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

None.

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_VALUE is generated by Material if enum is SHININESS and the shininess param is greater than the MAX_SHININESS_NV.

INVALID_VALUE is generated by Material if enum is SPOT_EXPONENT and the shininess param is greater than the MAX_SPOT_EXPONENT_NV.

New State

None.

New Implementation Dependent State

(table 6.24, p214) add the following entries:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_SHININESS_NV	Z+	GetIntegerv	128	Maximum shininess for specular lighting	2.13.2	-
MAX_SPOT_EXPONENT_NV	Z+	GetIntegerv	128	Maximum exponent for spot lights	2.13.2	-

NVIDIA Implementation Details

NVIDIA’s Release 4 drivers incorrectly and accidently advertised this extension with an "EXT" prefix instead of an "NV" prefix. Release 5 and later drivers correctly advertise this extension with an "NV" extension.

Revision History

5/20/00 - earlier versions of this specification had the incorrect enumerant values which did not match NVIDIA’s driver implementation.

Name

NV_register_combiners

Name Strings

GL_NV_register_combiners

Notice

Copyright NVIDIA Corporation, 1999, 2000.

IP Status

NVIDIA Proprietary.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation as designated in the document. The receipt or possession of this document does not convey any express or implied rights to reproduce, disclose, distribute or prepare deivative works its contents or to manufacture, use, sell or import anything that it may describe in whole or in part. The document is provided as is with no express or implied representation or warranty of any kind as the accuracy of the information, its fitness for a particular purpose or otherwise.

Status

Shipping (version 1.1)

Version

NVIDIA Date: July 25, 2000 (version 1.1)
$Date$ $Revision$

Number

191

Dependencies

ARB_multitexture, assuming the value of MAX_ACTIVE_TEXTURES_ARB is at least 2.

Written based on the wording of the OpenGL 1.2 specification with the ARB_multitexture appendix E.

Overview

NVIDIA’s next-generation graphics processor and its derivative designs support an extremely configurable mechanism know as "register combiners" for computing fragment colors.

The register combiner mechanism is a significant redesign of NVIDIA’s original TNT combiner mechanism as introduced by NVIDIA’s RIVA TNT graphics processor. Familiarity with the TNT combiners will help the reader appreciate the greatly enhanced register combiners functionality (see the NV_texture_env_combine4 OpenGL extension specification for this background). The register combiner mechanism has the following enhanced functionality:

The numeric range of combiner computations is from [-1,1] (instead of TNT’s [0,1] numeric range),

The set of available combiner inputs is expanded to include the secondary color, fog color, fog factor, and a second combiner constant color (TNT’s available combiner inputs consist of only zero, a single combiner constant color, the primary color, texture 0, texture 1, and, in the case of combiner 1, the result of combiner 0).

Each combiner variable input can be independently scaled and biased into several possible numeric ranges (TNT can only complement combiner inputs).

Each combiner stage computes three distinct outputs (instead TNT’s single combiner output).

The output operations include support for computing dot products (TNT has no support for computing dot products).

After each output operation, there is a configurable scale and bias applied (TNT’s combiner operations builds in a scale and/or bias into some of its combiner operations).

Each input variable for each combiner stage is fetched from any entry in a combiner register set. Moreover, the outputs of each combiner stage are written into the register set of the subsequent combiner stage (TNT could only use the result from combiner 0 as a possible input to combiner 1; TNT lacks the notion of an input/output register set).

The register combiner mechanism supports at least two general combiner stages and then a special final combiner stage appropriate for applying a color sum and fog computation (TNT provides two simpler combiner stages, and TNT’s color sum and fog stages are hard-wired and not subsumed by the combiner mechanism as in register combiners).

The register combiners fit into the OpenGL pipeline as a rasterization processing stage operating in parallel to the traditional OpenGL texture environment, color sum, AND fog application. Enabling this extension bypasses OpenGL’s existing texture environment, color sum, and fog application processing and instead use the register combiners. The combiner and texture environment state is orthogonal so modifying combiner state does not change the traditional OpenGL texture environment state and the texture environment state is ignored when combiners are enabled.

OpenGL application developers can use the register combiner mechanism for very sophisticated shading techniques. For example, an approximation of Blinn’s bump mapping technique can be achieved with the combiner mechanism. Additionally, multi-pass shading models that require several passes with unextended OpenGL 1.2 functionality can be implemented in several fewer passes with register combiners.

Issues

Should we expose the full register combiners mechanism?

RESOLUTION: NO. We ignore small bits of NV10 hardware functionality. The texture LOD input is ignored. We also ignore the inverts on input to the EF product.

Do we provide full gets for all the combiner state?

RESOLUTION: YES.

Do we parameterize combiner input and output updates to avoid enumerant explosions?

RESOLUTION: YES. To update a combiner stage input variable, you need to specify the <stage>, <portion>, and <variable>. To update a combiner stage output operation, you need to specify the <stage> and <portion>. This does mean that we need to add special Get routines that are likewise parameterized. Hence, GetCombinerInputParameter*, GetCombinerOutputParameter*, and GetFinalCombinerInputParameter*.

Is the register combiner functionality a super-set of the TNT combiner functionality?

Yes, but only in the sense of being a computational super-set. All computations performed with the TNT combiners can be performed with the register combiners, but the sequence of operations necessary to configure an identical computational result can be quite different.

For example, the TNT combiners have an operation that includes a final complement operation. The register combiners can perform range mappings only on inputs, but not on outputs. The register combiners can mimic the TNT operation with a post-operation complement only by taking pains to complement on input any uses of the output in later combiner stages.

What this does mean is that NV10’s hardware functionality will permit support for both the NV_register_combiners AND NV_texture_env_combine4 extensions.

Note the existance of an "speclit" input complement bit supported by NV10 (but not accessible through the NV_register_combiners extensions).

Should we say anything about the precision of the combiner computations?

RESOLUTION: NO. The spec is written as if the computations are done on floating point values ranging from -1.0 to 1.0 (clamping is specified where this range is exceeded). The fact that NV10 does the computations as 9-bit signed fixed point is not mentioned in the spec. This permits a future design to support more precision or use a floating pointing representation.

What should the initial combiner state be?

RESOLUTION: See tables NV_register_combiners.4 and NV_register_combiners.5. The default state has one general combiner stage active that modulates the incoming color with texture 0. The final combiner is setup initially to implement OpenGL 1.2's standard color sum and fog stages.

What should happen to the TEXTURE0_ARB and TEXTUER1_ARB inputs if one or both textures are disabled?

RESOLUTION: The value of these inputs is undefined.

What do the TEXTURE0_ARB and TEXTURE1_ARB inputs correspond to? Does the number correspond to the absolute texture unit number or is the number based on how many textures are enabled (ie, TEXTURE_ARB0 would correspond to the 2nd texture unit if the 2nd unit is enabled, but the 1st is disabled).

RESOLUTION: The absolute texture unit.

This should be a lot less confusing to the programmer than having the texture inputs switch textures if texture 0 is disabled.

Note that the proposed hardware actually determines the TEXTURE0 and TEXTURE1 input based on which texture is enabled. This means it is up to the ICD to properly update the combiner state when just one texture is enabled. Since we will already have to do this to track the standard OpenGL texture environment for ARB_multitexture, we can do it for this extension too.

Should the combiners state be PushAttrib/PopAttrib'ed along with the texture state?

RESOLUTION: YES.

Should we advertise the LOD fractional input to the combiners?

RESOLUTION: NO.

There will be a performance impact when two combiner stages are enabled versus just one stage. Should we mention that somewhere?

RESOLUTION: NO. (But it is worth mentioning in this issues section.)

Should the scale and bias for the CombinerOutputNV be indicated by enumerants or specified outright as floats?

RESOLUTION: ENUMERANTS. While some future combiners might support an arbitrary scale & bias specified as floats, NV10 just does the enumerated options.

Should a dot product be computed in parralel with the sum of products?

RESOLUTION: NO. Language has been added ot the CombinerOutputNV discussion saying that if either <abDotProduct> or <cdDotProduct>

is true, then <sumOutput> must be GL_DISCARD.

The rationale for this is that we want to minimize the number of adders that are required to ease a transition to floating point.

New Procedures and Functions

```
    void CombinerParameterfvNV(GLenum pname,
                               const GLfloat *params);

    void CombinerParameterivNV(GLenum pname,
                               const GLint *params);

    void CombinerParameterfNV(GLenum pname,
                              GLfloat param);

    void CombinerParameteriNV(GLenum pname,
                              GLint param);

    void CombinerInputNV(GLenum stage,
                         GLenum portion,
                         GLenum variable,
                         GLenum input,
                         GLenum mapping,
                         GLenum componentUsage);

    void CombinerOutputNV(GLenum stage,
                          GLenum portion,
                          GLenum abOutput,
                          GLenum cdOutput,
                          GLenum sumOutput,
                          GLenum scale,
                          GLenum bias,
                          GLboolean abDotProduct,
                          GLboolean cdDotProduct,
                          GLboolean muxSum);

    void FinalCombinerInputNV(GLenum variable,
                              GLenum input,
                              GLenum mapping,
                              GLenum componentUsage);

    void GetCombinerInputParameterfvNV(GLenum stage,
                                       GLenum portion,
                                       GLenum variable,
                                       GLenum pname,
                                       GLfloat *params);

    void GetCombinerInputParameterivNV(GLenum stage,
                                       GLenum portion,
                                       GLenum variable,
                                       GLenum pname,
                                       GLint *params);
    void GetCombinerOutputParameterfvNV(GLenum stage,
                                        GLenum portion,
                                        GLenum pname,
                                        GLfloat *params);

    void GetCombinerOutputParameterivNV(GLenum stage,
                                        GLenum portion,
                                        GLenum pname,
                                        GLint *params);

    void GetFinalCombinerInputParameterfvNV(GLenum variable,
                                            GLenum pname,
                                            GLfloat *params);

    void GetFinalCombinerInputParameterivNV(GLenum variable,
                                            GLenum pname,
                                            GLfloat *params);
```

New Tokens

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

REGISTER_COMBINERS_NV	0x8522

Accepted by the <stage> parameter of CombinerInputNV, CombinerOutputNV, GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, and GetCombinerOutputParameterivNV:

COMBINER0_NV	0x8550
COMBINER1_NV	0x8551
COMBINER2_NV	0x8552
COMBINER3_NV	0x8553
COMBINER4_NV	0x8554
COMBINER5_NV	0x8555
COMBINER6_NV	0x8556
COMBINER7_NV	0x8557

Accepted by the <variable> parameter of CombinerInputNV, GetCombinerInputParameterfvNV, and GetCombinerInputParameterivNV:

VARIABLE_A_NV	0x8523
VARIABLE_B_NV	0x8524
VARIABLE_C_NV	0x8525
VARIABLE_D_NV	0x8526

Accepted by the <variable> parameter of FinalCombinerInputNV, GetFinalCombinerInputParameterfvNV, and GetFinalCombinerInputParameterivNV:

VARIABLE_A_NV	
VARIABLE_B_NV	
VARIABLE_C_NV	
VARIABLE_D_NV	
VARIABLE_E_NV	0x8527
VARIABLE_F_NV	0x8528
VARIABLE_G_NV	0x8529

Accepted by the <input> parameter of CombinerInputNV:

ZERO		(not new)
CONSTANT_COLOR0_NV	0x852A	
CONSTANT_COLOR1_NV	0x852B	
FOG		(not new)
PRIMARY_COLOR_NV	0x852C	
SECONDARY_COLOR_NV	0x852D	
SPARE0_NV	0x852E	
SPARE1_NV	0x852F	
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)

Accepted by the <mapping> parameter of CombinerInputNV:

UNSIGNED_IDENTITY_NV	0x8536
UNSIGNED_INVERT_NV	0x8537
EXPAND_NORMAL_NV	0x8538
EXPAND_NEGATE_NV	0x8539
HALF_BIAS_NORMAL_NV	0x853A
HALF_BIAS_NEGATE_NV	0x853B
SIGNED_IDENTITY_NV	0x853C
SIGNED_NEGATE_NV	0x853D

Accepted by the <input> parameter of FinalCombinerInputNV:

ZERO		(not new)
CONSTANT_COLOR0_NV		
CONSTANT_COLOR1_NV		
FOG		(not new)
PRIMARY_COLOR_NV		
SECONDARY_COLOR_NV		
SPARE0_NV		
SPARE1_NV		
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)
E_TIMES_F_NV	0x8531	
SPARE0_PLUS_SECONDARY_COLOR_NV	0x8532	

Accepted by the <mapping> parameter of FinalCombinerInputNV:

UNSIGNED_IDENTITY_NV
UNSIGNED_INVERT_NV

Accepted by the <scale> parameter of CombinerOutputNV:

NONE		(not new)
SCALE_BY_TWO_NV	0x853E	
SCALE_BY_FOUR_NV	0x853F	
SCALE_BY_ONE_HALF_NV	0x8540	

Accepted by the <bias> parameter of CombinerOutputNV:

NONE		(not new)
BIAS_BY_NEGATIVE_ONE_HALF_NV	0x8541	

Accepted by the <abOutput>, <cdOutput>, and <sumOutput> parameter of CombinerOutputNV:

DISCARD_NV	0x8530	
PRIMARY_COLOR_NV		
SECONDARY_COLOR_NV		
SPARE0_NV		
SPARE1_NV		
TEXTURE0_ARB		(see ARB_multitexture)
TEXTURE1_ARB		(see ARB_multitexture)

Accepted by the <pname> parameter of GetCombinerInputParameterfvNV and GetCombinerInputParameterivNV:

COMBINER_INPUT_NV	0x8542
COMBINER_MAPPING_NV	0x8543
COMBINER_COMPONENT_USAGE_NV	0x8544

Accepted by the <pname> parameter of GetCombinerOutputParameterfvNV and GetCombinerOutputParameterivNV:

COMBINER_AB_DOT_PRODUCT_NV	0x8545
COMBINER_CD_DOT_PRODUCT_NV	0x8546
COMBINER_MUX_SUM_NV	0x8547
COMBINER_SCALE_NV	0x8548
COMBINER_BIAS_NV	0x8549
COMBINER_AB_OUTPUT_NV	0x854A
COMBINER_CD_OUTPUT_NV	0x854B
COMBINER_SUM_OUTPUT_NV	0x854C

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

CONSTANT_COLOR0_NV
CONSTANT_COLOR1_NV

Accepted by the <pname> parameter of CombinerParameterfvNV, CombinerParameterivNV, CombinerParameterfNV, CombinerParameteriNV, GetBooleanv, GetDoublev, GetFloatv, and GetIntegerv:

NUM_GENERAL_COMBINERS_NV	0x854E
COLOR_SUM_CLAMP_NV	0x854F

```
    Accepted by the <pname> parameter of GetFinalCombinerInputParameterfvNV
    and GetFinalCombinerInputParameterivNV:

        COMBINER_INPUT_NV
        COMBINER_MAPPING_NV
        COMBINER_COMPONENT_USAGE_NV

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

        MAX_GENERAL_COMBINERS_NV           0x854D
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

-- Figure 3.1 "Rasterization" (page 58)

+ Change the "Texturing" block to say "Texture Fetching".

+ Insert a new block between "Texture Fetching" and "Color Sum". Name the new block "Texture Environment Application".

+ Insert a new block after "Texture Fetching". Name the new block "Register Combiners Application".

+ The output of the "Texture Fetching" stage feeds to both "Texture Environment Application" and "Register Combiners Application".

+ The input for "Color Sum" comes from "Texture Environment Application".

+ The output to "Fragments" is switched (controlled by Disable/Enable REGISTER_COMBINERS_NV) between the output of "Fog" and "Register Combiners Application".

Essentially, when register combiners are enabled, the entire standard texture environment application, color sum, and fog blocks are replaced with the single register combiners block. [Note that this is different from how the NV_texture_env_combine4 extension works; that extension controls the texture environment application block, but still uses the standard color sum and fog blocks.]

-- NEW Section 3.8.12 "Register Combiners Application"

"In parallel to the texture application, color sum, and fog processes described in sections 3.8.10, 3.9, and 3.10, register combiners provide a means of computing fcoc, the final combiner output color, for each fragment generated by rasterization.

The register combiners consist of two or more general combiner stages arranged in a fixed sequence ordered by each combiner stage’s number. An implementation supports a maximum number of general combiners stages, which may be queried by calling GetIntegerv with the symbolic constant MAX_GENERAL_COMBINERS_NV. Implementations must support at least two general combiner stages. The general combiner stages are named COMBINER0_NV, COMBINER1_NV, and so on.

Each general combiner in the sequence receives its inputs and computes its outputs in an identical manner. At the end of the sequence of general combiner stages, there is a final combiner stage that operates in a different manner than the general combiner stages. The general combiner operation is described first, followed by a description of the final combiner operation.

Each combiner stage (the general combiner stages and the final combiner stage) has an associated combiner register set. Each combiner register set contains <n> RGBA vectors with components ranging from -1.0 to 1.0 where <n> is 8 plus the maximum number of active textures supported (that is, the implementation’s value for MAX_ACTIVE_TEXTURES_ARB). The combiner register set entries are listed in the table NV_register_combiners.1.

[Table NV_register_combiners.1]

Register Name	Initial Value	Reference	Output Status
ZERO	0	-	read only
CONSTANT_COLOR0_NV	ccc0	Section 3.8.12.1	read only
CONSTANT_COLOR1_NV	ccc1	Section 3.8.12.1	read only
FOG	Cf	Section 3.10	read only
PRIMARY_COLOR_NV	cpri	Section 2.13.1	read/write
SECONDARY_COLOR_NV	csec	Section 2.13.1	read/write
SPARE0_NV	see below	Section 3.8.12	read/write
SPARE1_NV	undefined	Section 3.8.12	read/write
TEXTURE0_ARB	CT0	Figure E.2	read/write
TEXTURE1_ARB	CT1	Figure E.2	read/write
TEXTURE<i>_ARB	CT<i>	Figure E.2	read/write

The register set of COMBINER0_NV, the first combiner stage, is initialized as described in table NV_register_combiners.1.

The initial value of the alpha portion of register SECONDARY_COLOR_NV is undefined. The initial value of the alpha portion of register SPARE0_NV is the alpha component of texture 0 if texturing is enabled for texture 0; however, the initial value of the RGB portion SPARE0_NV is undefined. The initial value of the SPARE1_NV register is undefined. The initial of registers TEXTURE0_ARB, TEXTURE1_ARB, and TEXTURE<i>_ARB are undefined if texturing is not enabled for textures 0, 1, and <i>, respectively.

3.8.12.1 Combiner Parameters

Combiner parameters are specified by

```
CombinerParameterfvNV(GLenum pname, const GLfloat *params);
CombinerParameterivNV(GLenum pname, const GLint *params);
CombinerParameterfNV(GLenum pname, GLfloat param);
CombinerParameteriNV(GLenum pname, GLint param);
```

<pname> is a symbolic constant indicating which parameter is to be

set as described in the table NV_register_combiners.2:

[Table NV_register_combiners.2]

Parameter	Name	Number of values	Type
ccc0	CONSTANT_COLOR0_NV	4	color
ccc1	CONSTANT_COLOR1_NV	4	color
ngc	NUM_GENERAL_COMBINERS_NV	1	positive integer
csc	COLOR_SUM_CLAMP_NV	1	boolean

<params> is a pointer to a group of values to which to set the indicated parameter. <param> is simply the indicated parameter. The number of values pointed to depends on the parameter being set as shown in the table above. Color parameters specified with CombinerParameter*NV are converted to floating-point values (if specified as integers) as indicated by Table 2.6 for signed integers. The floating-point color values are then clamped to the range [0,1].

The values ccc0 and ccc1 named by CONSTANT_COLOR0_NV and CONSTANT_COLOR1_NV are constant colors available for inputs to the combiner stages. The value ngc named by NUM_GENERAL_COMBINERS_NV is a positive integer indicating how many general combiner stages are active, that is, how many general combiner stages a fragment should be processed by. Setting ngc to a value less than one or greater than the value of MAX_GENERAL_COMBINERS_NV generates an INVALID_VALUE error. The value csc named by COLOR_SUM_CLAMP_NV is a boolean described in section 3.8.12.3.

3.8.12.2 General Combiner Stage Operation

The command

```
    CombinerInputNV(GLenum stage,
                    GLenum portion,
                    GLenum variable,
                    GLenum input,
                    GLenum mapping,
                    GLenum componentUsage);
```

controls the assignment of all the general combiner input variables. For the RGB combiner portion, these are Argb, Brgb, Crgb, and Drgb; and for the combiner alpha portion, these are Aa, Ba, Ca, and Da. The <stage> parameter is a symbolic constant of the form COMBINER<i>_NV, indicating that general combiner stage <i> is to be updated. The constant COMBINER<i>_NV = COMBINER0_NV + <i> where <i> is in the range 0 to <k>-1 and <k> is the implementation dependent value of MAX_COMBINERS_NV. The <portion> parameter may be either RGB or ALPHA and determines whether the RGB color vector or alpha scalar portion of the specified combiner stage is updated. The <variable> parameter may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, or VARIABLE_D_NV and determines which respective variable of the specified combiner stage and combiner stage portion is updated.

The <input>, <mapping>, and <componentUsage> parameters specify the assignment of a value for the input variable indicated by

<stage>, <portion>, and <variable>. The <input> parameter may be one of the register names from table NV_register_combiners.1.

The <componentUsage> parameter may be one of RGB, ALPHA, or BLUE.

When the <portion> parameter is RGB, a <componentUsage> parameter of RGB indicates that the RGB portion of the indicated register should be assigned to the RGB portion of the combiner input variable, while an ALPHA <componentUsage> parameter indicates that the alpha portion of the indicated register should be replicated across the RGB portion of the combiner input variable.

When the <portion> parameter is ALPHA, the <componentUsage> parameter of ALPHA indicates that the alpha portion of the indicated register should be assigned to the alpha portion of the combiner input variable, while a BLUE <componentUsage> parameter indicates that the blue component of the indicated register should be assigned to the alpha portion of the combiner input variable.

When the <portion> parameter is ALPHA, a <componentUsage> parameter of RGB generates an INVALID_OPERATION error. When the <portion> parameter is RGB, a <componentUsage> parameter of BLUE generates an INVALID_OPERATION error.

When the <portion> parameter is ALPHA, an <input> parameter of FOG generates an INVALID_OPERATION error. The alpha component of the fog register is only available in the final combiner. The alpha component of the fog register is the fragment’s fog factor when fog is enabled; otherwise, the alpha component of the fog register is one.

Before the value in the register named by <input> is assigned to the specified input variable, a range mapping is performed based on <mapping>. The mapping may be one of the tokens from the table NV_register_combiners.3.

[Table NV_register_combiners.3]

Mapping Name	Mapping Function
UNSIGNED_IDENTITY_NV	max(0.0, e)
UNSIGNED_INVERT_NV	1.0 - min(max(e, 0.0), 1.0)
EXPAND_NORMAL_NV	2.0 * max(0.0, e) - 1.0
EXPAND_NEGATE_NV	-2.0 * max(0.0, e) + 1.0
HALF_BIAS_NORMAL_NV	max(0.0, e) - 0.5
HALF_BIAS_NEGATE_NV	-max(0.0, e) + 0.5
SIGNED_IDENTITY_NV	e
SIGNED_NEGATE_NV	-e

Based on the <mapping> parameter, the mapping function in the table above is evaluated for each element <e> of the input vector before assigning the result to the specified input variable. Note that the mapping for the RGB and alpha portion of each input variable is distinct.

Each general combiner stage computes the following ten expressions based on the values assigned to the variables Argb, Brgb, Crgb, Drgb, Aa, Ba, Ca, and Da as determined by the combiner state set by CombinerInputNV.

["gcc" stands for general combiner computation.]

```
   gcc1rgb = [ Argb[r]*Brgb[r], Argb[g]*Brgb[g], Argb[b]*Brgb[b] ]

   gcc2rgb = [ Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b],
               Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b],
               Argb[r]*Brgb[r] + Argb[g]*Brgb[g] + Argb[b]*Brgb[b] ]

   gcc3rgb = [ Crgb[r]*Drgb[r], Crgb[g]*Drgb[g], Crgb[b]*Drgb[b] ]

   gcc4rgb = [ Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b],
               Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b],
               Crgb[r]*Drgb[r] + Crgb[g]*Drgb[g] + Crgb[b]*Drgb[b] ]

   gcc5rgb = gcc1rgb + gcc3rgb

   gcc6rgb = gcc1rgb or gcc3rgb              [see below]

   gcc1a   = Aa * Ba

   gcc2a   = Ca * Da

   gcc3a   = gcc1a + gcc2a

   gcc4a   = gcc1a or gcc2a                  [see below]
```

The computation of gcc6rgb and gcc4a involves a special "or" operation. This operation evaluates to the right-hand operand if the alpha component of the combiner’s SPARE0_NV register is less than 0.5; otherwise, the operation evaluates to the left-hand operand.

The command

```
    CombinerOutputNV(GLenum stage,
                     GLenum portion,
                     GLenum abOutput,
                     GLenum cdOutput,
                     GLenum sumOutput,
                     GLenum scale,
                     GLenum bias,
                     GLboolean abDotProduct,
                     GLboolean cdDotProduct,
                     GLboolean muxSum);
```

controls the general combiner output operation including designating the register set locations where results of the general combiner’s three computations are written. The <stage> and <portion> parameters take the same values as the respective parameters for CombinerInputNV.

If the <portion> parameter is ALPHA, specifying a non-FALSE value for either of the parameters <abDotProduct> or <cdDotProduct>, generates an INVALID_VALUE error.

If the <abDotProduct> or <cdDotProduct> parameter is non-FALSE, the value of the <sumOutput> parameter must be GL_DISCARD_NV; otherwise, generate an INVALID_OPERATION error.

The <scale> parameter must be one of NONE, SCALE_BY_TWO_NV, SCALE_BY_FOUR_NV, or SCALE_BY_ONE_HALF_NV and specifies the value of the combiner stage’s portion scale, either cscalergb or cscalea depending on the <portion> parameter, to 1.0, 2.0, 4.0, or 0.5, respectively.

The <bias> parameter must be either NONE or BIAS_BY_NEGATIVE_ONE_HALF_NV and specifies the value of the combiner stage’s portion bias, either cbiasrgb or cbiasa depending on the <portion> parameter, to 0.0 or -0.5, respectively. If <scale> is either SCALE_BY_ONE_HALF_NV or SCALE_BY_FOUR_NV, a <bias> of BIAS_BY_NEGATIVE_ONE_HALF_NV generates an INVALID_OPERATION error.

```
If the <abDotProduct> parameter is FALSE, then

   if <portion> is RGB,     out1rgb = max(min(gcc1rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,   out1a   = max(min((gcc1a + cbiasa) * cscalea, 1), -1)

otherwise <portion> must be RGB and

   out1rgb = max(min((gcc2rgb + cbiasrgb) * cscalergb, 1), -1)

If the <cdDotProduct> parameter is FALSE, then

   if <portion> is RGB,      out2rgb = max(min((gcc3rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out2a   = max(min((gcc2a + cbiasa) * cscalea, 1), -1)

otherwise <portion> must be RGB so

   out2rgb = max(min((gcc4rgb + cbiasrgb) * cscalergb, 1), -1)

If the <muxSum> parameter is FALSE, then

   if <portion> is RGB,      out3rgb = max(min((gcc5rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out3a   = max(min((gcc3a + cbiasa) * cscalea, 1), -1)

otherwise

   if <portion> is RGB,      out3rgb = max(min((gcc6rgb + cbiasrgb) * cscalergb, 1), -1)
   if <portion> is ALPHA,    out3a   = max(min((gcc4a + cbiasa) * cscalea, 1), -1)
```

out1rgb, out2rgb, and out3rgb are written to the RGB portion of combiner stage’s registers named by <abOutput>, <cdOutput>, and <sumOutput> respectively. out1a, out2a, and out3a are written to the alpha portion of combiner stage’s registers named by <abOutput>, <cdOutput>, and <sumOutput> respectively. The parameters <abOutput>, <cdOutput>, and <sumOutput> must be either DISCARD_NV or one of the register names from table NV_register_combiners.1 that has an output status of read/write. If an output is set to DISCARD_NV, that output is not written to any register. The error INVALID_OPERATION is generated if <abOutput>, <cdOutput>, and <sumOutput> do not all name unique register names (though multiple outputs to DISCARD_NV are legal).

When the general combiner stage’s register set is written based on the computed outputs, the updated register set is copied to the register set of the subsequent combiner stage in the combiner sequence. Copied undefined values are likewise undefined.

The subsequent combiner stage following the last active general combiner stage, indicated by the general combiner stage’s number being equal to ngc-1, in the sequence is the final combiner stage. In other words, the number of general combiner stages each fragment is transformed by is determined by the value of NUM_GENERAL_COMBINERS_NV.

3.8.12.3 Final Combiner Stage Operation

The final combiner stage operates differently from the general combiner stages. While a general combiner stage updates its register set and passes the register set to the next combiner stage, the final combiner outputs an RGBA color fcoc, the final combiner output color. The final combiner stage is capable of applying the standard OpenGL color sum and fog operations, but has the configurability to be used for other purposes.

The command

```
   FinalCombinerInputNV(GLenum variable,
                        GLenum input,
                        GLenum mapping,
                        GLenum componentUsage);
```

controls the assignment of all the final combiner input variables. The variables A, B, C, D, E, and F are RGB vectors. The variable G is an alpha scalar. The <variable> parameter may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, VARIABLE_D_NV, VARIABLE_E_NV, VARIABLE_F_NV, and VARIABLE_G_NV, and determines which respective variable of the final combiner stage is updated.

The <input>, <mapping>, and <componentUsage> parameters specify the assignment of a value for the input variable indicated by <variable>.

The <input> parameter may be any one of the register names from table NV_register_combiners.1 or be one of two pseudo-register names, either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV. The value of E_TIMES_F_NV is the product of the value of variable E times the value of variable F. The value of SPARE0_PLUS_SECONDARY_COLOR_NV is the value the SPARE0_NV register mapped using the UNSIGNED_IDENITY_NV input mapping plus the value of the SECONDARY_COLOR_NV register mapped using the UNSIGNED_IDENTITY_NV input mapping. If csc, the color sum clamp, is non-FALSE, the value of SPARE0_PLUS_SECONDARY_COLOR_NV is first clamped to the range [0,1]. The alpha component of E_TIMES_F_NV and SPARE0_PLUS_SECONDARY_COLOR_NV is always zero.

When <variable> is one of VARIABLE_E_NV, VARIABLE_F_NV, or VARIABLE_G_NV and <input> is either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV, generate an INVALID_OPERATION error. When <variable> is VARIABLE_A_NV and <input> is SPARE0_PLUS_SECONDARY_COLOR_NV, generate an INVALID_OPERATION error.

The <componentUsage> parameter may be one of RGB, BLUE, ALPHA (with certain restrictions depending on the <variable> and <input>

as described below).

When the <variable> parameter is not VARIABLE_G_NV, a <componentUsage> parameter of RGB indicates that the RGB portion of the indicated register should be assigned to the RGB portion of the combiner input variable, while an ALPHA <componentUsage> parameter indicates that the alpha portion of the indicated register should be replicated across the RGB portion of the combiner input variable.

When the <variable> parameter is VARIABLE_G_NV, a <componentUsage> parameter of ALPHA indicates that the alpha component of the indicated register should be assigned to the alpha portion of the G input variable, while a BLUE <componentUsage> parameter indicates that the blue component of the indicated register should be assigned to the alpha portion of the G input variable.

The INVALID_OPERATION error is generated when <componentUsage> is BLUE and <variable> is not VARIABLE_G_NV. The INVALID_OPERATION error is generated when <componentUsage> is RGB and <variable> is VARIABLE_G_NV.

The INVALID_OPERATION error is generated when both the <input> parameter is either E_TIMES_F_NV or SPARE0_PLUS_SECONDARY_COLOR_NV and the <componentUsage> parameter is ALPHA or BLUE.

Before the value in the register named by <input> is assigned to the specified input variable, a range mapping is performed based on <mapping>. The mapping may be either UNSIGNED_IDENTITY_NV or UNSIGNED_INVERT_NV and operates as specified in table NV_register_combiners.3.

The final combiner stage computes the following expression based on the values assigned to the variables A, B, C, D, E, F, and G as determined by the combiner state set by FinalCombinerInputNV

```
  fcoc = [ min(ab[r] + iac[r] + D[r], 1.0),
           min(ab[g] + iac[g] + D[g], 1.0),
           min(ab[b] + iac[b] + D[b], 1.0),
           G ]
```

where

```
  ab   = [ A[r]*B[r], A[g]*B[g], A[b]*B[b] ]
  iac  = [ (1.0 -A [r])*C[r], (1.0 - A[g])*C[g], (1.0 - A[b])*C[b] ]
```

3.8.12.4 Required State

The state required for the register combiners is a bit indicating whether register combiners are enabled or disabled, an integer indicating how many general combiners are active, a bit indicating whether or not the color sum clamp to 1 should be performed, two RGBA constant colors, <n> sets of general combiner stage state where <n> is the value of MAX_GENERAL_COMBINERS_NV, and the final combiner stage state. The per-stage general combiner state consists of the RGB input portion state and the alpha input portion state. Each portion (RGB and alpha) of the per-stage general combiner state consists of: four integers indicating the input register for the four variables A, B, C, and D; four integers to indicate each variable’s range mapping; four bits to indicate whether to use the alpha component of the input for each variable; a bit indicating whether the AB dot product should be output; a bit indicating whether the CD dot product should be output; a bit indicating whether the sum or mux output should be output; two integers to maintain the output scale and bias enumerants; three integers to maintain the output register set names. The final combiner stage state consists of seven integers to indicate the input register for the seven variables A, B, C, D, E, F, and G; seven integers to indicate each variable’s range mapping; and seven bits to indicate whether to use the alpha component of the input for each variable.

The general combiner per-stage state is initialized as described in table NV_register_combiners.4.

[Table NV_register_combiners.4]

Portion	Variable	Input	Component Usage	Mapping
RGB	A	PRIMARY_COLOR_NV	RGB	UNSIGNED_IDENTITY_NV
RGB	B	TEXTURE#_ARB	RGB	UNSIGNED_IDENTITY_NV
RGB	C	ZERO	RGB	UNSIGNED_IDENTITY_NV
RGB	D	ZERO	RGB	UNSIGNED_IDENTITY_NV
alpha	A	PRIMARY_COLOR_NV	ALPHA	UNSIGNED_IDENTITY_NV
alpha	B	TEXTURE#_ARB	ALPHA	UNSIGNED_IDENTITY_NV
alpha	C	ZERO	ALPHA	UNSIGNED_IDENTITY_NV
alpha	D	ZERO	ALPHA	UNSIGNED_IDENTITY_NV

where # is the general combiner stage number.

The final combiner stage state is initialized as described in table NV_register_combiners.5.

[Table NV_register_combiners.5]

Variable	Input	Component Usage	Mapping
A	FOG	ALPHA	UNSIGNED_IDENTITY_NV
B	SPARE0_PLUS_SECONDARY_COLOR_NV	RGB	UNSIGNED_IDENTITY_NV
C	FOG	RGB	UNSIGNED_IDENTITY_NV
D	ZERO	RGB	UNSIGNED_IDENTITY_NV
E	ZERO	RGB	UNSIGNED_IDENTITY_NV
F	ZERO	RGB	UNSIGNED_IDENTITY_NV
G	SPARE0_NV	ALPHA	UNSIGNED_IDENTITY_NV"

-- NEW Section 3.8.11 "Antialiasing Application"

Insert the following paragraph BEFORE the section’s first paragraph:

"Register combiners are enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constant REGISTER_COMBINERS_NV. If the register combiners are enabled (and not in color index mode), the fragment’s color value is replaced with fcoc, the final combiner output color, computed in section 3.8.12; otherwise, the fragment’s color value is the result of the fog application in section 3.10."

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

-- Section 6.1.3 "Enumerated Queries"

Change the first two sentences (page 182) to say:

"Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, combiners, etc.) as well as a symbolic constant. These are"

Add to the bottom of the list of function prototypes (page 183):

```
  void GetCombinerInputParameterfvNV(GLenum stage, GLenum portion,
                                     GLenum variable,
                                     GLenum pname, const GLfloat *params);
  void GetCombinerInputParameterivNV(GLenum stage, GLenum portion,
                                     GLenum variable,
                                     GLenum pname, const GLint *params);
  void GetCombinerOutputParameterfvNV(GLenum stage, GLenum portion,
                                      GLenum pname, const GLfloat *params);
  void GetCombinerOutputParameterivNV(GLenum stage, GLenum portion,
                                      GLenum pname, GLint *params);
  void GetFinalCombinerInputParameterfvNV(GLenum variable, GLenum pname,
                                          const GLfloat *params);
  void GetFinalCombinerInputParameterivNV(GLenum variable, GLenum pname,
                                          const GLfloat *params);
```

Add the following paragraph to the end of the section (page 184):

"The GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, and GetCombinerOutputParameterivNV parameter <stage> may be one of COMBINER0_NV, COMBINER1_NV, and so on, indicating which general combiner stage to query. The GetCombinerInputParameterfvNV, GetCombinerInputParameterivNV, GetCombinerOutputParameterfvNV, and GetCombinerOutputParameterivNV parameter <portion> may be either RGB or ALPHA, indicating which portion of the general combiner stage to query. The GetCombinerInputParameterfvNV and GetCombinerInputParameterivNV parameter <variable> may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, or VARIABLE_D_NV, indicating which variable of the general combiner stage to query. The GetFinalCombinerInputParameterfvNV and GetFinalCombinerInputParameterivNV parameter <variable> may be one of VARIABLE_A_NV, VARIABLE_B_NV, VARIABLE_C_NV, VARIABLE_D_NV, VARIABLE_E_NV, VARIABLE_F_NV, or VARIABLE_G_NV."

Additions to the GLX Specification

None.

GLX Protocol

Thirteen new GL commands are added.

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

```
CombinerParameterfNV
    2           12                rendering command length
    2           4136              rendering command opcode
    4           ENUM              pname
    4           FLOAT32           param

CombinerParameterfvNV
    2           8+4*n             rendering command length
    2           4137              rendering command opcode
    4           ENUM              pname
                0x852A   n=4      GL_CONSANT_COLOR0_NV
                0x852B   n=4      GL_CONSANT_COLOR1_NV
                0x854E   n=1      GL_NUM_GENERAL_COMBINERS_NV
                0x854F   n=1      GL_COLOR_SUM_CLAMP_NV
                else     n=0
    4*n         LISTofFLOAT32     params

CombinerParameteriNV
    2           12                rendering command length
    2           4138              rendering command opcode
    4           ENUM              pname
    4           INT32             param

CombinerParameterivNV
    2           8+4*n             rendering command length
    2           4139              rendering command opcode
    4           ENUM              pname
                0x852A   n=4      GL_CONSANT_COLOR0_NV
                0x852B   n=4      GL_CONSANT_COLOR1_NV
                0x854E   n=1      GL_NUM_GENERAL_COMBINERS_NV
                0x854F   n=1      GL_COLOR_SUM_CLAMP_NV
                else     n=0
    4*n         LISTofINT32       params

CombinerInputNV
    2           28                rendering command length
    2           4140              rendering command opcode
    4           ENUM              stage
    4           ENUM              portion
    4           ENUM              variable
    4           ENUM              input
    4           ENUM              mapping
    4           ENUM              componentUsage
    CombinerOutputNV
        2           36              rendering command length
        2           4141            rendering command opcode
        4           ENUM            stage
        4           ENUM            portion
        4           ENUM            abOutput
        4           ENUM            cdOutput
        4           ENUM            sumOutput
        4           ENUM            scale
        4           ENUM            bias
        1           BOOL            abDotProduct
        1           BOOL            cdDotProduct
        1           BOOL            muxSum
        1           BOOL            unused

    FinalCombinerOutputNV
        2           20              rendering command length
        2           4142            rendering command opcode
        4           ENUM            variable
        4           ENUM            input
        4           ENUM            mapping
        4           ENUM            componentUsage
```

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

```
    GetCombinerInputParameterfvNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           7               request length
        4           1270            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            stage
        4           ENUM            portion
        4           ENUM            variable
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m = (n==1 ? 0 : n)
        4                           unused
        4           CARD32          unused

        if (n=1) this follows:

        4           FLOAT32         params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofFLOAT32   params
        GetCombinerInputParameterivNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           7               request length
            4           1271            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            portion
            4           ENUM            variable
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           INT32           params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofINT32     params

        GetCombinerOutputParameterfvNV
            1           CARD8           opcode (X assigned)
            1           17              GLX opcode (glXVendorPrivateWithReply)
            2           6               request length
            4           1272            vendor specific opcode
            4           GLX_CONTEXT_TAG context tag
            4           ENUM            stage
            4           ENUM            portion
            4           ENUM            pname
          =>
            1           1               reply
            1                           unused
            2           CARD16          sequence number
            4           m               reply length, m = (n==1 ? 0 : n)
            4                           unused
            4           CARD32          unused

            if (n=1) this follows:

            4           FLOAT32         params
            12                          unused

            otherwise this follows:

            16                          unused
            n*4         LISTofFLOAT32   params
```

```
GetCombinerOutputParameterivNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           6               request length
    4           1273            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            stage
    4           ENUM            portion
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           INT32           params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofINT32     params

GetFinalCombinerInputParameterfvNV
    1           CARD8           opcode (X assigned)
    1           17              GLX opcode (glXVendorPrivateWithReply)
    2           5               request length
    4           1274            vendor specific opcode
    4           GLX_CONTEXT_TAG context tag
    4           ENUM            variable
    4           ENUM            pname
  =>
    1           1               reply
    1                           unused
    2           CARD16          sequence number
    4           m               reply length, m = (n==1 ? 0 : n)
    4                           unused
    4           CARD32          unused

    if (n=1) this follows:

    4           FLOAT32         params
    12                          unused

    otherwise this follows:

    16                          unused
    n*4         LISTofFLOAT32   params
    GetFinalCombinerInputParameterivNV
        1           CARD8           opcode (X assigned)
        1           17              GLX opcode (glXVendorPrivateWithReply)
        2           5               request length
        4           1275            vendor specific opcode
        4           GLX_CONTEXT_TAG context tag
        4           ENUM            variable
        4           ENUM            pname
      =>
        1           1               reply
        1                           unused
        2           CARD16          sequence number
        4           m               reply length, m = (n==1 ? 0 : n)
        4                           unused
        4           CARD32          unused

        if (n=1) this follows:

        4           INT32           params
        12                          unused

        otherwise this follows:

        16                          unused
        n*4         LISTofINT32     params
```

Errors

INVALID_VALUE is generated when CombinerParameterfvNV or CombinerParameterivNV is called with <pname> set to NUM_GENERAL_COMBINERS and the value pointed to by <params> is less than one or greater or equal to the value of MAX_GENERAL_COMBINERS_NV.

INVALID_OPERATION is generated when CombinerInputNV is called with a <componentUsage> parameter of RGB and a <portion> parameter of ALPHA.

INVALID_OPERATION is generated when CombinerInputNV is called with a <componentUsage> parameter of BLUE and a <portion> parameter of RGB.

INVALID_OPERATION is generated When CombinerInputNV is called with a <componentUsage> parameter of ALPHA and an <input> parameter of FOG.

INVALID_VALUE is generated when CombinerOutputNV is called with a <portion> parameter of ALPHA, but a non-FALSE value for either of the parameters <abDotProduct> or <cdDotProduct>.

INVALID_OPERATION is generated when CombinerOutputNV is called with a <scale> of either SCALE_BY_TWO_NV or SCALE_BY_FOUR_NV and a <bias> of BIAS_BY_NEGATIVE_ONE_HALF_NV.

INVALID_OPERATION is generated when CombinerOutputNV is called such that <abOutput>, <cdOutput>, and <sumOutput> do not all name unique register names (though multiple outputs to DISCARD_NV are legal).

```
INVALID_OPERATION is generated when FinalCombinerOutputNV
is called where <variable> is one of VARIABLE_E_NV,
VARIABLE_F_NV, or VARIABLE_G_NV and <input> is E_TIMES_F_NV
or SPARE0_PLUS_SECONDARY_COLOR_NV.

INVALID_OPERATION is generated when FinalCombinerOutputNV
is called where <variable> is VARIABLE_A_NV and <input> is
SPARE0_PLUS_SECONDARY_COLOR_NV.

INVALID_OPERATION is generated when FinalCombinerInputNV is called
with VARIABLE_G_NV for <variable> and RGB for <componentUsage>.

INVALID_OPERATION is generated when FinalCombinerInputNV is called
with a value other than VARIABLE_G_NV for <variable> and BLUE for
<componentUsage>.

INVALID_OPERATION is generated when FinalCombinerInputNV is
called where the <input> parameter is either E_TIMES_F_NV or
SPARE0_PLUS_SECONDARY_COLOR_NV and the <componentUsage> parameter
is ALPHA.

INVALID_OPERATION is generated when CombinerOutputNV is called with
either <abDotProduct> or <cdDotProduct> assigned non-FALSE and
<sumOutput> is not GL_DISCARD_NV.
```

New State

-- (NEW table 6.29, after p217)

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
REGISTER_COMBINERS_NV texture/enable	B	IsEnabled	False	register combiners enable	3.8.11	
NUM_GENERAL_COMBINERS_NV	Z+	GetIntegerv	1	number of active combiner stages	3.8.12.1	texture
COLOR_SUM_CLAMP_NV	B	GetBooleanv	True	whether or not SPARE0_PLUS_ SECONDARY_ COLOR_NV clamps combiner stages	3.8.12.1	texture
CONSTANT_COLOR0_NV	C	GetFloatv	0,0,0,0	combiner constant color zero	3.8.12.1	texture
CONSTANT_COLOR1_NV	C	GetFloatv	0,0,0,0	combiner constant color one	3.8.12.1	texture
COMBINER_INPUT_NV	Z8x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	combiner input variables	3.8.12.2	texture
COMBINER_COMPONENT_USAGE_NV	Z3x#x2x4	GetCombinerInputParameter*NV	see 3.8.12.4	use alpha for combiner input	3.8.12.2	texture
COMBINER_MAPPING_NV	Z8x#x2x4	GetCombinerInputParameter*NV	UNSIGNED_IDENTITY_NV	complement combiner input	3.8.12.2	texture
COMBINER_AB_DOT_PRODUCT_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output AB dot product	3.8.12.3	texture
COMBINER_CD_DOT_PRODUCT_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output CD dot product	3.8.12.3	texture
COMBINER_MUX_SUM_NV	Bx#x2	GetCombinerOutputParameter*NV	False	output mux sum	3.8.12.3	texture
COMBINER_SCALE_NV	Z2x#x2	GetCombinerOutputParameter*NV	NONE	output scale	3.8.12.3	texture
COMBINER_BIAS_NV	Z2x#x2	GetCombinerOutputParameter*NV	NONE	output bias	3.8.12.3	texture
COMBINER_AB_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	DISCARD_NV	AB output register	3.8.12.3	texture
COMBINER_CD_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	DISCARD_NV	CD output register	3.8.12.3	texture
COMBINER_SUM_OUTPUT_NV	Z7x#x2	GetCombinerOutputParameter*NV	SPARE0_NV	sum output register	3.8.12.3	texture
COMBINER_INPUT_NV	Z10x7	GetFinalCombinerInputParameter*NV	see 3.8.12.4	final combiner input	3.8.12.4	texture
COMBINER_MAPPING_NV	Z2x7	GetFinalCombinerInputParameter*NV	UNSIGNED_IDENTITY_NV	final combiner input mapping	3.8.12.4	texture
COMBINER_COMPONENT_USAGE_NV	Z2x7	GetFinalCombinerInputParameter*NV	see 3.8.12.4	use alpha for final combiner input mapping	3.8.12.4	texture

[where # is the value of MAX_GENERAL_COMBINERS_NV]

New Implementation Dependent State

(table 6.24, p214) add the following entry:

Get Value	Type	Get Command	Minimum Value	Description	Sec	Attribute
MAX_GENERAL_COMBINERS_NV	Z+	GetIntegerv	2	Maximum num of general combiner stages	3.8.12	-

NVIDIA Implementation Details

The effective range of the RGB portion of the final combiner should be [0,4] if the color sum clamp is false. Exercising this range requires assigning SPARE0_PLUS_SECONDARY_COLOR_NV to the D variable and either B or C or both B and C. In practice this is a very unlikely configuration.

However due to a bug in the GeForce 256 and Quadro hardware, values generated above 2 in the RGB portion of the final combiner will be computed incorrectly. GeForce2 GTS and subsequent NVIDIA GPUs have fixed this bug.

Revision History

April 4, 2000 - Document that alpha component of the FOG register should be zero when fog is disabled. The Release 4 NVIDIA drivers have a bug where this is not always true (though it often still is). The bug is fixed in the Release 5 NVIDIA drivers.

June 8, 2000 - The alpha component of the FOG register is not available for use until the final combiner. The specification previously incorrectly stated:

> "INVALID_OPERATION is generated When CombinerInputNV is called with a <portion> parameter of ALPHA and an <input> parameter of FOG."

It is actually the <componentUsage> (not the <portion>) that should not be allowed to be ALPHA. The Release 4 NVIDIA drivers implemented the above incorrect error check. The Release 5 (and later) NVIDIA drivers (after June 8, 2000) have fixed this bug and correctly implement the error based on <componentUsage>.

The specification previously did not allow BLUE for the <componentUsage> of the G variable in the final combiner. This is now allowed in the Release 5 (and later) NVIDIA drivers (after June 8, 2000). The Release 4 NVIDIA drivers do not permit BLUE for the <componentUsage> of the G variable and generate an INVALID_OPERATION error if this is attempted. The Release 5 NVIDIA drivers (after June 8, 2000) have fixed this bug and permit BLUE for the <componentUsage> of the G variable.

Name

NV_texgen_emboss

Name Strings

GL_NV_texgen_emboss

Notice

IP Status

NVIDIA Proprietary.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation. The receipt or possession of this document does not convey any express or implied rights to reproduce, disclose, distribute or prepare deivative works its contents or to manufacture, use, sell or import anything that it may describe in whole or in part. The document is provided as is with no express or implied representation or warranty of any kind as the accuracy of the information, its fitness for a particular purpose or otherwise.

Status

Shipping (version 1.0)

Version

NVIDIA Date: July 25, 2000

Number

193

Dependencies

ARB_multitexture.

Written based on the wording of the OpenGL 1.2 specification and the ARB_multitexture extension.

Overview

This extension provides a new texture coordinate generation mode suitable for multitexture-based embossing (or bump mapping) effects.

Given two texture units, this extension generates the texture coordinates of a second texture unit (an odd-numbered texture unit) as a perturbation of a first texture unit (an even-numbered texture unit one less than the second texture unit). The perturbation is based on the normal, tangent, and light vectors. The normal vector is supplied by glNormal; the light vector is supplied as a direction vector to a specified OpenGL light’s position; and the tanget vector is supplied by the second texture unit’s current texture coordinate. The perturbation is also scaled by program-supplied scaling constants.

If both texture units are bound to the same texture representing a height field, by subtracting the difference between the resulting two filtered texels, programs can achieve a per-pixel embossing effect.

Issues

Can you do embossing on any texture unit?

NO. Just odd numbered units. This meets a constraint of the proposed hardware implementation, and because embossing takes two texture units anyway, it shouldn’t be a real limitation.

Can you just enable one coordinate of a texture unit for embossing?

Yes but NOT REALLY. The texture coordinate generation formula is specified such that only when ALL the coordinates are enabled and are using embossing, do you get the embossing computation. Otherwise, you get undefined values for texture coordinates enabled for texture coordinate generation and setup for embossing.

Does the light specified have to be enabled for embossing to work?

Yes, currently. But perhaps we could require implementations to enable a phantom light (the light colors would be black).

Could the emboss constant just be the reciprocal of the width and height of the texture units texture if that’s what the programmer will have it be most of the time?

NO. Too much work and there may be reasons for the programmer to control this.

OpenGL’s base texture environment functionality isn’t powerful enough to do the subtraction needed for embossing. Where would you get powerful enough texture environment functionality.

Another extension. Try NV_register_combiners.

What is the interpretation of CT?

For the purposes of embossing, CT should be thought of as the vertex’s tangent vector. This tangent vector indicates the direction on the "surface" where PCTs is not changing and PCTt is increasing.

Are the CT and PCT variables the user-supplied current texture coordinates?

YES. Except when the texture unit’s texture coordinate evaluator is enabled, then CT and PCT use the respective evaluated texture coordinates.

This extension specification’s language "Denote as CT the texture unit’s current texture coordinates" and "Denote as PCT the previous texture unit’s current texture coordinates" refers to the "current texture coordinates" OpenGL state which is the state specified via glTexCoord. Plus the exception for evaluators.

To be explicit, PCT is NOT the result of texgen or the texture matrix. Likewise, CT is NOT the result of texgen or the texture matrix. PCT and CT are the respective texture unit’s evaluated texture coordinate if the vertex is evaluated with texture coordinate evaluation enabled, otherwise if the vertex is generated via vertex arrays with the respective texture coordinate array enabled, the texture coordinate from the texture coordinate array, otherwise the respective current texture coordinate is used.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

```
EMBOSS_MAP_NV                       0x855F
```

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain EMBOSS_MAP_NV.

Accepted by the <pname> parameters of GetTexGendv, GetTexGenfv, GetTexGeniv, TexGend, TexGendv, TexGenf, TexGenfv, TexGeni, and TexGeniv:

```
EMBOSS_LIGHT_NV                     0x855D
EMBOSS_CONSTANT_NV                  0x855E
```

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, or EMBOSS_MAP_NV."

Add these paragraphs after the 4th paragraph:

"When used with a suitable texture, suitable explicit texture coordinates, a suitable (extended) texture environment, suitable lighting parameters, and suitable embossing parameters, calling TexGen with TEXTURE_GEN_MODE indicating EMBOSS_MAP_NV can simulate the lighting effect of embossing on a polygon. The error INVALID_ENUM occurs when the active texture unit has an even number.

The emboss constant and emboss light parameters for controlling the EMBOSS_MAP_NV mode are specified by calling TexGen with pname set to EMBOSS_CONSTANT_NV and EMBOSS_LIGHT_NV respectively.

When pname is EMBOSS_CONSTANT_NV, param or what params points to is a scalar value. An error INVALID_ENUM occurs if pname is EMBOSS_CONSTANT_NV and coord is R or Q. An error INVALID_ENUM also occurs if pname is EMBOSS_CONSTANT_NV and the active texture unit number is even.

When pname is EMBOSS_LIGHT_NV, param or what params points to is a symbolic constant of the form LIGHTi, indicating that light i is to have the specified parameter set. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and coord is R or Q. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and the active texture unit number is even. An error INVALID_ENUM occurs if pname is EMBOSS_LIGHT_NV and the value i for LIGHTi is negative or is greater than or equal to the value of MAX_LIGHTS.

If TEXTURE_GEN_MODE indicates EMBOSS_MAP_NV, the generation function for the coordinates S, T, R, and Q is computed as follows.

Denote as L the light direction vector from the vertex’s eye position to the position of the light specified by the coordinate’s EMBOSS_LIGHT_NV state (the direction vector is computed as described in Section 3.13.1).

Denote as N the current normal after transformation to eye coordinates.

Denote as CT the texture unit’s current texture coordinates transformed to eye coordinates by normal transformation (as described in Section 3.10.3) and normalized.

However, if the vertex is evaluated (as described in Section 5.1) and the texture unit’s texture coordinate map is enabled, use the texture unit’s evaluated texture coordinate to compute CT.

Denote as B the cross product of N and the <s,t,r> vector of CT.

```
  Bx = Ny*CTr - CTt*Nz
  By = Nz*CTs - CTr*Nx
  Bz = Nx*CTt - CTs*Ny
```

Denote as BN the normalized version of the vector B.

```
  BNx = Bx / sqrt(Bx*Bx + By*By + Bz*Bz);
  BNy = By / sqrt(Bx*Bx + By*By + Bz*Bz);
  BNz = Bz / sqrt(Bx*Bx + By*By + Bz*Bz);
```

Denote as T the cross product of B and N.

```
  Tx = BNy*Nz - Ny*BNz
  Ty = BNz*Nx - Nz*BNx
  Tz = BNx*Ny - Nx*BNy
```

Observe that BN and T are orthonormal.

Denote as PCT the previous texture unit’s current texture coordinates. If the number of the texture unit for the texture coordinates being generated is n, then the previous texture unit is texture unit number n-1. Note that n is restricted to be odd.

However, if the vertex is evaluated (as described in Section 5.1) and the previous texture unit's texture coordinate map is enabled, use the previous texture unit's evaluated texture coordinate to compute PCT.

Denote Ks as the S coordinate's EMBOSS_CONSTANT_NV state. Denote Kt as the T coordinate's EMBOSS_CONSTANT_NV state. These constants should typically be set to the reciprocal of the width and height respectively of the texture map used for embossing.

Denote E as follows:

```
  Es = PCTs + Ks * (Lx*BNx + Ly*BNy + Lz*BNz) * PCTq
  Et = PCTt - Kt * (Lx*Tx + Ly*Ty + Lz*Tz) * PCTq
  Er = PCTr
  Eq = PCTq
```

Then the value assigned to an s, t, r, and q coordinates are Es, Et, Er, and Eq respectively. However, for this assignment to occur, the following three conditions must be met. First, all the texture coordinate generation modes of all the texture coordinates (S, T, R, and Q) of the texture unit must be set to EMBOSS_MAP_NV. Second, all the texture coordinate generation modes of the texture unit must be enabled. Third, the EMBOSS_LIGHT_NV parameters of coordinates S and T must be identical and the light and lighting must be enabled. If these conditions are not met, the values of all coordinates in the texture unit with the EMBOSS_MAP_NV mode are undefined."

The last paragraph's first sentence should be changed to:

"The state required for texture coordinate generation comprises a five-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR; also, an emboss constant and emboss light are required for each of the four coordinates.... The initial values for emboss constants and emboss lights are 1.0 and LIGHT0 respectively."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <pname> of TEXTURE_GEN_MODE, a <param> value or value of what <params> points to of EMBOSS_MAP_NV, and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_CONSTANT_NV and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_LIGHT_NV and the active texture unit is even.

INVALID_ENUM is generated when TexGen is called with a <coord> of R or Q when <pname> indicates EMBOSS_CONSTANT_NV.

INVALID_ENUM is generated when TexGen is called with a <coord> of R or Q when <pname> indicates EMBOSS_LIGHT_NV.

INVALID_ENUM is generated when TexGen is called with a <pname> of EMBOSS_LIGHT_NV and the value of i for the parameter LIGHTi is negative or is greater than or equal to the value of MAX_LIGHTS.

New State

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture
EMBOSS_CONSTANT_NV	4xR	GetTexGenfv	1.0	Scaling constant for emboss texgen	2.10.4	texture
EMBOSS_LIGHT_NV	4xZ8*	GetTexGeniv	LIGHT0	Light used for embossing.	2.10.4	texture

When ARB_multitexture is supported, the Type column is per-texture unit.

(the TEXTURE_GEN_MODE type changes from 4xZ3 to 4xZ5)

New Implementation State

None

Name

NV_texgen_reflection

Name Strings

GL_NV_texgen_reflection

Notice

Version

August 24, 1999

Number

179

Dependencies

Written based on the wording of the OpenGL 1.2 specification but not dependent on it.

Overview

This extension provides two new texture coordinate generation modes that are useful texture-based lighting and environment mapping. The reflection map mode generates texture coordinates (s,t,r) matching the vertex’s eye-space reflection vector. The reflection map mode is useful for environment mapping without the singularity inherent in sphere mapping. The normal map mode generates texture coordinates (s,t,r) matching the vertex’s transformed eye-space normal. The normal map mode is useful for sophisticated cube map texturing-based diffuse lighting models.

Issues

Should we place the normal/reflection vector in the (s,t,r) texture coordinates or (s,t,q) coordinates?

RESOLUTION: (s,t,r). Even if the proposed hardware uses "q" for the third component, the API should claim to support generation of (s,t,r) and let the texture matrix (through a concatenation with the user-supplied texture matrix) move "r" into "q".

Should you be able to have some texture coordinates computing REFLECTION_MAP_NV and others not? Same question with NORMAL_MAP_NV.

RESOLUTION: YES. This is the way that SPHERE_MAP works. It is not clear that this would ever be useful though.

Should something special be said about the handling of the q texture coordinate for this spec?

RESOLUTION: NO. But the following paragraph is useful for implementors concerned about the handling of q.

The REFLECTION_MAP_NV and NORMAL_MAP_NV modes are intended to supply reflection and normal vectors for cube map texturing hardware. When these modes are used for cube map texturing, the generated texture coordinates can be thought of as an reflection vector. The value of the q texture coordinate then simply scales the vector but does not change its direction. Because only the vector direction (not the vector magnitude) matters for cube map texturing, implementations are free to leave q undefined when any of the s, t, or r texture coordinates are generated using REFLECTION_MAP_NV or NORMAL_MAP_NV.

New Procedures and Functions

None

New Tokens

Accepted by the <param> parameters of TexGend, TexGenf, and TexGeni when <pname> parameter is TEXTURE_GEN_MODE:

```
NORMAL_MAP_NV                     0x8511
REFLECTION_MAP_NV                 0x8512
```

When the <pname> parameter of TexGendv, TexGenfv, and TexGeniv is TEXTURE_GEN_MODE, then the array <params> may also contain NORMAL_MAP_NV or REFLECTION_MAP_NV.

Additions to Chapter 2 of the 1.2 Specification (OpenGL Operation)

-- Section 2.10.4 "Generating Texture Coordinates"

Change the last sentence in the 1st paragraph to:

"If <pname> is TEXTURE_GEN_MODE, then either <params> points to or <param> is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP_NV, or NORMAL_MAP_NV."

Add these paragraphs after the 4th paragraph:

"If TEXTURE_GEN_MODE indicates REFLECTION_MAP_NV, compute the reflection vector r as described for the SPHERE_MAP mode. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = rx; the value assigned to a t coordinate is t = ry; and the value assigned to a r coordinate is r = rz. Calling TexGen with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV generates the error INVALID_ENUM.

If TEXTURE_GEN_MODE indicates NORMAL_MAP_NV, compute the normal vector n’ as described in section 2.10.3. Then the value assigned to an s coordinate (the first TexGen argument value is S) is s = nfx; the value assigned to a t coordinate is t = nfy; and the value assigned to a r coordinate is r = nfz. (The values nfx, nfy, and nfz are the components of nf.) Calling TexGen with a <coord>

of Q when <pname> indicates REFLECTION_MAP_NV generates the error INVALID_ENUM.

The last paragraph’s first sentence should be changed to:

"The state required for texture coordinate generation comprises a five-valued integer for each coordinate indicating coordinate generation mode, ..."

Additions to Chapter 3 of the 1.2 Specification (Rasterization)

None

Additions to Chapter 4 of the 1.2 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.2 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.2 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Errors

INVALID_ENUM is generated when TexGen is called with a <coord> of Q when <pname> indicates REFLECTION_MAP_NV or NORMAL_MAP_NV.

New State

(table 6.14, p204) change the entry for TEXTURE_GEN_MODE to:

Get Value	Type	Get Command	Initial Value	Description	Sec	Attribute
TEXTURE_GEN_MODE	4xZ5	GetTexGeniv	EYE_LINEAR	Function used for texgen (for s,t,r, and q)	2.10.4	texture

(the type changes from 4xZ3 to 4xZ5)

New Implementation State

None

Name

NV_texture_env_combine4

Name Strings

GL_NV_texture_env_combine4

Contact

Michael Gold, NVIDIA Corporation (gold ’at’ nvidia.com)

Notice

Copyright NVIDIA Corporation, 1999, 2000.

IP Status

NVIDIA Proprietary.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation. The receipt or possession of this document does not convey any express or implied rights to reproduce, disclose, distribute or prepare deivative works its contents or to manufacture, use, sell or import anything that it may describe in whole or in part. The document is provided as is with no express or implied representation or warranty of any kind as the accuracy of the information, its fitness for a particular purpose or otherwise.

Version

Last update: July 25, 2000
$Date: 1999/06/21 13:54:17 $ $Revision: 1.2 $

Number

195

Dependencies

EXT_texture_env_combine is required and is modified by this extension
ARB_multitexture affects the definition of this extension

Overview

New texture environment function COMBINE4_NV allows programmable texture combiner operations, including

ADD	Arg0 * Arg1 + Arg2 * Arg3
ADD_SIGNED_EXT	Arg0 * Arg1 + Arg2 * Arg3 - 0.5

where Arg0, Arg1, Arg2 and Arg3 are derived from

ZERO	the value 0
PRIMARY_COLOR_EXT	primary color of incoming fragment
TEXTURE	texture color of corresponding texture unit
CONSTANT_EXT	texture environment constant color
PREVIOUS_EXT	result of previous texture environment; on texture unit 0, this maps to PRIMARY_COLOR_EXT
TEXTURE<n>_ARB	texture color of the <n>th texture unit

In addition, the result may be scaled by 1.0, 2.0 or 4.0.

Issues

None

New Procedures and Functions

None

New Tokens

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is TEXTURE_ENV_MODE

COMBINE4_NV	0x8503

Accepted by the <pname> parameter of GetTexEnvfv, GetTexEnviv, TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <target> parameter value is TEXTURE_ENV

SOURCE3_RGB_NV	0x8583
SOURCE3_ALPHA_NV	0x858B
OPERAND3_RGB_NV	0x8593
OPERAND3_ALPHA_NV	0x859B

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE3_RGB_NV, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT, SOURCE2_ALPHA_EXT, or SOURCE3_ALPHA_NV

ZERO
TEXTURE<n>_ARB

where <n> is in the range 0 to NUMBER_OF_TEXTURE_UNITS_ARB-1.

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_RGB_EXT, OPERAND1_RGB_EXT, OPERAND2_RGB_EXT or OPERAND3_RGB_NV

SRC_COLOR
ONE_MINUS_SRC_COLOR
SRC_ALPHA
ONE_MINUS_SRC_ALPHA

Accepted by the <params> parameter of TexEnvf, TexEnvi, TexEnvfv, and TexEnviv when the <pname> parameter value is OPERAND0_ALPHA_EXT, OPERAND1_ALPHA_EXT, OPERAND2_ALPHA_EXT, or OPERAND3_ALPHA_NV

SRC_ALPHA
ONE_MINUS_SRC_ALPHA

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

None

Additions to Chapter 3 of the GL Specification (Rasterization)

Added to subsection 3.8.9, before the paragraph describing the state requirements:

If the value of TEXTURE_ENV_MODE is COMBINE4_NV, the form of the texture function depends on the values of COMBINE_RGB_EXT and COMBINE_ALPHA_EXT, according to table 3.21. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE_EXT and ALPHA_SCALE, respectively. The results are clamped to [0,1]. If the value of COMBINE_RGB_EXT or COMBINE_ALPHA_EXT is not one of the listed values, the result is undefined.

COMBINE_RGB_EXT or COMBINE_ALPHA_EXT	Texture Function
ADD	Arg0 * Arg1 + Arg2 * Arg3
ADD_SIGNED_EXT	Arg0 * Arg1 + Arg2 * Arg3 - 0.5

Table 3.21: COMBINE4_NV texture functions

The arguments Arg0, Arg1, Arg2 and Arg3 are determined by the values of SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT. In the following two tables, Ct and At are the filtered texture RGB and alpha values; Cc and Ac are the texture environment RGB and alpha values; Cf and Af are the RGB and alpha of the primary color of the incoming fragment; and Cp and Ap are the RGB and alpha values resulting from the previous texture environment. On texture environment 0, Cp and Ap are identical to Cf and Af, respectively. Ct<n> and At<n> are the filtered texture RGB and alpha values from the texture bound to the <n>th texture unit. If the <n>th texture unit is disabled, the value of each component is 1. The relationship is described in tables 3.22 and 3.23.

SOURCE<n>_RGB_EXT	OPERAND<n>_RGB_EXT	Argument
ZERO	SRC_COLOR	0
	ONE_MINUS_SRC_COLOR	1
	SRC_ALPHA	0
	ONE_MINUS_SRC_ALPHA	1
TEXTURE	SRC_COLOR	Ct
	ONE_MINUS_SRC_COLOR	(1-Ct)
	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_COLOR	Cc
	ONE_MINUS_SRC_COLOR	(1-Cc)
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_COLOR	Cf
	ONE_MINUS_SRC_COLOR	(1-Cf)
	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_COLOR	Cp
	ONE_MINUS_SRC_COLOR	(1-Cp)
	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)
TEXTURE<n>_ARB	SRC_COLOR	Ct<n>
	ONE_MINUS_SRC_COLOR	(1-Ct<n>)
	SRC_ALPHA	At<n>
	ONE_MINUS_SRC_ALPHA	(1-At<n>)

Table 3.22: Arguments for COMBINE_RGB_EXT functions

SOURCE<n>_ALPHA_EXT	OPERAND<n>_ALPHA_EXT	Argument
ZERO	SRC_ALPHA	0
	ONE_MINUS_SRC_ALPHA	1
TEXTURE	SRC_ALPHA	At
	ONE_MINUS_SRC_ALPHA	(1-At)
CONSTANT_EXT	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	(1-Ac)
PRIMARY_COLOR_EXT	SRC_ALPHA	Af
	ONE_MINUS_SRC_ALPHA	(1-Af)
PREVIOUS_EXT	SRC_ALPHA	Ap
	ONE_MINUS_SRC_ALPHA	(1-Ap)
TEXTURE<n>_ARB	SRC_ALPHA	At<n>
	ONE_MINUS_SRC_ALPHA	(1-At<n>)

Table 3.23: Arguments for COMBINE_ALPHA_EXT functions

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

None

Additions to Chapter 5 of the GL Specification (Special Functions)

None

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

None

Additions to the GLX Specification

None

GLX Protocol

None

Errors

INVALID_ENUM is generated if <params> value for SOURCE0_RGB_EXT, SOURCE1_RGB_EXT, SOURCE2_RGB_EXT, SOURCE3_RGB_NV, SOURCE0_ALPHA_EXT, SOURCE1_ALPHA_EXT, SOURCE2_ALPHA_EXT or SOURCE3_ALPHA_NV is not one of ZERO, TEXTURE, CONSTANT_EXT, PRIMARY_COLOR_EXT, PREVIOUS_EXT or TEXTURE<n>_ARB, where <n> is in the range 0 to NUMBER_OF_TEXTURE_UNITS_ARB-1.

INVALID_ENUM is generated if <params> value for OPERAND0_RGB_EXT, OPERAND1_RGB_EXT, OPERAND2_RGB_EXT or OPERAND3_RGB_NV is not one of SRC_COLOR, ONE_MINUS_SRC_COLOR, SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

INVALID_ENUM is generated if <params> value for OPERAND0_ALPHA_EXT OPERAND1_ALPHA_EXT, OPERAND2_ALPHA_EXT, or OPERAND3_ALPHA_NV is not one of SRC_ALPHA or ONE_MINUS_SRC_ALPHA.

Modifications to EXT_texture_env_combine

This extension relaxes the restrictions on SOURCE<n>_RGB_EXT, SOURCE<n>_ALPHA_EXT, OPERAND<n>_RGB_EXT and OPERAND<n>_ALPHA_EXT for use with EXT_texture_env_combine. All params specified by Table 3.22 and Table 3.23 are valid.

Dependencies on ARB_multitexture

If ARB_multitexture is not implemented, all references to TEXTURE<n>_ARB and NUMBER_OF_TEXTURE_UNITS_ARB are deleted.

New State

Get Value	Get Command	Type	Initial Value	Attribute
SOURCE3_RGB_NV	GetTexEnviv	n x Z5+n	ZERO	texture
SOURCE3_ALPHA_NV	GetTexEnviv	n x Z5+n	ZERO	texture
OPERAND3_RGB_NV	GetTexEnviv	n x Z2	ONE_MINUS_SRC_COLOR	texture
OPERAND3_ALPHA_NV	GetTexEnviv	n x Z2	ONE_MINUS_SRC_ALPHA	texture

New Implementation Dependent State

None

NVIDIA Implementation Details

```
    Because of a hardware limitation, TNT, TNT2, GeForce, and Quadro
    treat "scale by 4.0" with the COMBINE_RGB_EXT or COMBINE_ALPHA_EXT
    mode of ADD_SIGNED_EXT as "scale by 2.0".
```

Name

NV_vertex_array_range

Name Strings

GL_NV_vertex_array_range

Contact

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

Notice

IP Status

NVIDIA Proprietary.

This document is protected by copyright and contains information proprietary to NVIDIA Corporation. The receipt or possession of this document does not convey any express or implied rights to reproduce, disclose, distribute or prepare deivative works its contents or to manufacture, use, sell or import anything that it may describe in whole or in part. The document is provided as is with no express or implied representation or warranty of any kind as the accuracy of the information, its fitness for a particular purpose or otherwise.

Status

Shipping (version 1.0)

Version

July 25, 2000

Number

190

Dependencies

None

Overview

The goal of this extension is to permit extremely high vertex processing rates via OpenGL vertex arrays even when the CPU lacks the necessary data movement bandwidth to keep up with the rate at which the vertex engine can consume vertices. CPUs can keep up if they can just pass vertex indices to the hardware and let the hardware "pull" the actual vertex data via Direct Memory Access (DMA). Unfortunately, the current OpenGL 1.1 vertex array functionality has semantic constraints that make such an approach hard. Hence, the vertex array range extension.

This extension provides a mechanism for deferring the pulling of vertex array elements to facilitate DMAed pulling of vertices for fast, efficient vertex array transfers. The OpenGL client need only pass vertex indices to the hardware which can DMA the actual index's vertex data directly out of the client address space.

The OpenGL 1.1 vertex array functionality specifies a fairly strict coherency model for when OpenGL extracts vertex data from a vertex array and when the application can update the in memory vertex array data. The OpenGL 1.1 specification says "Changes made to array data between the execution of Begin and the corresponding execution of End may affect calls to ArrayElement that are made within the same Begin/End period in non-sequential ways. That is, a call to ArrayElement that precedes a change to array data may access the changed data, and a call that follows a change to array data may access the original data."

This means that by the time End returns (and DrawArrays and DrawElements return since they have implicit Ends), the actual vertex array data must be transferred to OpenGL. This strict coherency model prevents us from simply passing vertex element indices to the hardware and having the hardware "pull" the vertex data out (which is often long after the End for the primitive has returned to the application).

Relaxing this coherency model and bounding the range from which vertex array data can be pulled is key to making OpenGL vertex array transfers faster and more efficient.

The first task of the vertex array range extension is to relax the coherency model so that hardware can indeed "pull" vertex data from the OpenGL client's address space long after the application has completed sending the geometry primitives requiring the vertex data.

The second problem with the OpenGL 1.1 vertex array functionality is the lack of any guidance from the API about what region of memory vertices can be pulled from. There is no size limit for OpenGL 1.1 vertex arrays. Any vertex index that points to valid data in all enabled arrays is fair game. This makes it hard for a vertex DMA engine to pull vertices since they can be potentially pulled from anywhere in the OpenGL client address space.

The vertex array range extension specifies a range of the OpenGL client's address space where vertices can be pulled. Vertex indices that access any array elements outside the vertex array range are specified to be undefined. This permits hardware to DMA from finite regions of OpenGL client address space, making DMA engine implementation tractable.

The extension is specified such that an (error free) OpenGL client using the vertex array range functionality could no-op its vertex array range commands and operate equivalently to using (if slower than) the vertex array range functionality.

Because different memory types (local graphics memory, AGP memory) have different DMA bandwidths and caching behavior, this extension includes a window system dependent memory allocator to allocate cleanly the most appropriate memory for constructing a vertex array range. The memory allocator provided allows the application to tradeoff the desired CPU read frequency, CPU write frequency, and memory priority while still leaving it up to OpenGL implementation the exact memory type to be allocated.

Issues

How does this extension interact with the compiled_vertex_array extension?

I think they should be independent and not interfere with each other. In practice, if you use NV_vertex_array_range, you can surpass the performance of compiled_vertex_array

Should some explanation be added about what happens when an OpenGL application updates its address space in regions overlapping with the currently configured vertex array range?

RESOLUTION: I think the right thing is to say that you get non-sequential results. In practice, you’ll be using an old context DMA pointing to the old pages.

If the application change’s its address space within the vertex array range, the application should call glVertexArrayRangeNV again. That will re-make a new vertex array range context DMA for the application’s current address space.

If we are falling back to software transformation, do we still need to abide by leaving "undefined" vertices outside the vertex array range? For example, pointers that are not 32-bit aligned would likely cause a fall back.

RESOLUTION: No. The fact that vertex is "undefined" means we can do anything we want (as long as we send a vertex and do not crash) so it is perfectly fine for the software puller to grab vertex information not available to the hardware puller.

Should we give a programmer a sense of how big a vertex array range they can specify?

RESOLUTION: No. Just document it if there are limitations. Probably very hardware and operating system dependent.

Is it clear enough that language about ArrayElement also applies to DrawArrays and DrawElements?

Maybe not, but OpenGL 1.1 spec is clear that DrawArrays and DrawElements are defined in terms of ArrayElement.

Should glFlush be the same as glVertexArrayRangeFlush?

RESOLUTION: No. A glFlush is cheaper than a glVertexArrayRangeFlush though a glVertexArrayRangeFlushNV should do a flush.

If any the data for any enabled array for a given array element index falls outside of the vertex array range, what happens?

RESOLUTION: An undefined vertex is generated.

What error is generated in this case?

I don’t know yet. We should make sure the hardware really does let us know when vertices are undefined.

Note that this is a little weird for OpenGL since most errors in OpenGL result in the command being ignored. Not in this case though.

Should this extension support an interface for allocating video and AGP memory?

RESOLUTION: YES. It seems like we should be able to leave the task of memory allocation to DirectDraw, but DirectDraw’s asynchronous unmapping behavior and having to hold locks to update DirectDraw surfaces makes that mechanism to cumbersome.

Plus the API is a lot easier if we do it ourselves.

How do we decide what type of memory to allocate for the application?

RESOLUTION: Usage hints. The application rates the read frequency (how often will they read the memory), the write frequency (how often will they write the memory), and the priority (how important is this memory relative to other uses for the memory such as texturing) on a scale of 1.0 to 0.0. Using these hints and the size of the memory requsted, the OpenGL implementation decides where to allocate the memory.

We try to not directly expose particular types of memory (AGP, local memory, cached/uncached, etc) so future memory types can be supported by merely updating the OpenGL implementation.

Should the memory allocator functionality be available be a part of the GL or window system dependent (GLX or WGL) APIs?

RESOLUTION: The window system dependent API.

The memory allocator should be considered a window system/ operating system dependent operation. This also permits memory to be allocated when no OpenGL rendering contexts exist yet.

New Procedures and Functions

```
void VertexArrayRangeNV(sizei length, void *pointer)
void FlushVertexArrayRangeNV(void)
```

New Tokens

Accepted by the <cap> parameter of EnableClientState, DisableClientState, and IsEnabled:

VERTEX_ARRAY_RANGE_NV	0x851D

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

VERTEX_ARRAY_RANGE_LENGTH_NV	0x851E
VERTEX_ARRAY_RANGE_VALID_NV	0x851F
MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV	0x8520

Accepted by the <pname> parameter of GetPointerv:

VERTEX_ARRAY_RANGE_POINTER_NV	0x8521

Additions to Chapter 2 of the 1.1 Specification (OpenGL Operation)

After the discussion of vertex arrays (Section 2.8) add a description of the vertex array range:

"The command

```
void VertexArrayRangeNV(sizei length, void *pointer)
```

specifies the current vertex array range. When the vertex array range is enabled and valid, vertex array vertex transfers from within the vertex array range are potentially faster. The vertex array range is a contiguous region of (virtual) address space for placing vertex arrays. The "pointer" parameter is a pointer to the base of the vertex array range. The "length" pointer is the length of the vertex array range in basic machine units (typically unsigned bytes).

The vertex array range address space region extends from "pointer" to "pointer + length - 1" inclusive. When specified and enabled, vertex array vertex transfers from within the vertex array range are potentially faster.

There is some system burden associated with establishing a vertex array range (typically, the memory range must be locked down). If either the vertex array range pointer or size is set to zero, the previously established vertex array range is released (typically, unlocking the memory).

The vertex array range may not be established for operating system dependent reasons, and therefore, not valid. Reasons that a vertex array range cannot be established include spanning different memory types, the memory could not be locked down, alignment restrictions are not met, etc.

The vertex array range is enabled or disabled by calling EnableClientState or DisableClientState with the symbolic constant VERTEX_ARRAY_RANGE_NV.

The vertex array range is either valid or invalid and this state can be determined by querying VERTEX_ARRAY_RANGE_VALID_NV. The vertex array range is valid when the following conditions are met:

- VERTEX_ARRAY_RANGE_NV is enabled.

- VERTEX_ARRAY is enabled.

- VertexArrayRangeNV has been called with a non-null pointer and non-zero size.

- The vertex array range has been established.

- An implementation-dependent validity check based on the pointer alignment, size, and underlying memory type of the vertex array range region of memory.

- An implementation-dependent validity check based on the current vertex array state including the strides, sizes, types, and pointer alignments (but not pointer value) for currently enabled vertex arrays.

- Other implementation-dependent validaity checks based on other OpenGL rendering state.

Otherwise, the vertex array range is not valid. If the vertex array range is not valid, vertex array transfers will not be faster.

When the vertex array range is valid, ArrayElement commands may generate undefined vertices if and only if any indexed elements of the enabled arrays are not within the vertex array range or if the index is negative or greater or equal to the implementation-dependent value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV. If an undefined vertex is generated, an INVALID_OPERATION error may or may not be generated.

The vertex array cohenecy model specifies when vertex data must be be extracted from the vertex array memory. When the vertex array range is not valid, (quoting the specification) ‘Changes made to array data between the execution of Begin and the corresponding execution of End may effect calls to ArrayElement that are made within the same Begin/End period in non-sequential ways. That is, a call to ArrayElement that precedes a change to array data may access the changed data, and a call that follows a change to array data may access the original data.’

When the vertex array range is valid, the vertex array coherency model is relaxed so that changes made to array data until the next "vertex array range flush" may affects calls to ArrayElement in non-sequential ways. That is a call to ArrayElement that precedes a change to array data (without an intervening "vertex array range flush") may access the changed data, and a call that follows a change (without an intervening "vertex array range flush") to array data may access original data.

A ’vertex array range flush’ occurs when one of the following operations occur:

- Finish returns.

- o FlushVertexArrayRangeNV returns.
- o VertexArrayRangeNV returns.
- o ClientStateDisable of VERTEX_ARRAY_RANGE_NV returns.
- o ClientStateEnable of VETEX_ARRAY_RANGE_NV returns.
- o Another OpenGL context is made current.

The client state required to implement the vertex array range consists of an enable bit, a memory pointer, an integer size, and a valid bit.

If the memory mapping of pages within the vertex array range changes, using the vertex array range may or may not result in undefined data being fetched from the vertex arrays when the vertex array range is enabled and valid. To ensure that the vertex array range reflects the address space’s current state, the application is responsible for calling VertexArrayRange again after any memory mapping changes within the vertex array range."llo

Additions to Chapter 5 of the 1.1 Specification (Special Functions)

Add to the end of Section 5.4 "Display Lists"

"VertexArrayRangeNV and FlushVertexArrayRangeNV are not complied into display lists but are executed immediately.

If a display list is compiled while VERTEX_ARRAY_RANGE_NV is enabled, the commands ArrayElement, DrawArrays, DrawElements, and DrawRangeElements are accumulated into a display list as if VERTEX_ARRAY_RANGE_NV is disabled."

Additions to the WGL interface:

"When establishing a vertex array range, certain types of memory may be more efficient than other types of memory. The commands

```
void *wglAllocateMemoryNV(sizei size,
                          float readFrequency,
                          float writeFrequency,
                          float priority)
void wglFreeMemoryNV(void *pointer)
```

allocate and free memory that may be more suitable for establishing an efficient vertex array range than memory allocated by other means. The wglAllocateMemoryNV command allocates <size> bytes of contiguous memory.

The <readFrequency>, <writeFrequency>, and <priority> parameters are usage hints that the OpenGL implementation can use to determine the best type of memory to allocate. These parameters range from 0.0 to 1.0. A <readFrequency> of 1.0 indicates that the application intends to frequently read the allocated memory; a <readFrequency> of 0.0 indicates that the application will rarely or never read the memory. A <writeFrequency> of 1.0 indicates that the application intends to frequently write the allocated memory; a <writeFrequency> of 0.0 indicates that the application will rarely write the memory. A <priority> parameter of 1.0 indicates that memory type should be the most efficient available memory, even at the expense of (for example) available texture memory; a <priority> of 0.0 indicates that the vertex array range does not require an efficient memory type (for example, so that more efficient memory is available for other purposes such as texture memory).

The OpenGL implementation is free to use the <size>, <readFrequency>, <writeFrequency>, and <priority> parameters to determine what memory type should be allocated. The memory types available and how the memory type is determined is implementation dependent (and the implementation is free to ignore any or all of the above parameters).

Possible memory types that could be allocated are uncached memory, write-combined memory, graphics hardware memory, etc. The intent of the wglAllocateMemoryNV command is to permit the allocation of memory for efficient vertex array range usage. However, there is no requirement that memory allocated by wglAllocateMemoryNV must be used to allocate memory for vertex array ranges.

If the memory cannot be allocated, a NULL pointer is returned (and no OpenGL error is generated). An implementation that does not support this extension’s memory allocation interface is free to never allocate memory (always return NULL).

The wglFreeMemoryNV command frees memory allocated with wglAllocateMemoryNV. The <pointer> should be a pointer returned by wglAllocateMemoryNV and not previously freed. If a pointer is passed to wglFreeMemoryNV that was not allocated via wglAllocateMemoryNV or was previously freed (without being reallocated), the free is ignored with no error reported.

The memory allocated by wglAllocateMemoryNV should be available to all other threads in the address space where the memory is allocated (the memory is not private to a single thread). Any thread in the address space (not simply the thread that allocated the memory) may use wglFreeMemoryNV to free memory allocated by itself or any other thread.

Because wglAllocateMemoryNV and wglFreeMemoryNV are not OpenGL rendering commands, these commands do not require a current context. They operate normally even if called within a Begin/End or while compiling a display list."

Additions to the GLX Specification

Same language as the "Additions to the WGL Specification" section except all references to wglAllocateMemoryNV and wglFreeMemoryNV should be replaced with glXAllocateMemoryNV and glXFreeMemoryNV respectively.

Additional language:

"OpenGL implementations using GLX indirect rendering should fail to set up the vertex array range (failing to set the vertex array valid bit so the vertex array range functionality is not usable). Additionally, glXAllocateMemoryNV always fails to allocate memory (returns NULL) when used with an indirect rendering context."

GLX Protocol

None

Errors

INVALID_OPERATION is generated if VertexArrayRange or FlushVertexArrayRange is called between the execution of Begin and the corresponding execution of End.

INVALID_OPERATION may be generated if an undefined vertex is generated.

New State

Get Value	Get Command	Type	Initial Value	Attrib
VERTEX_ARRAY_RANGE_NV	IsEnabled	B	False	vertex-array
VERTEX_ARRAY_RANGE_POINTER_NV	GetPointerv	Z+	0	vertex-array
VERTEX_ARRAY_RANGE_LENGTH_NV	GetIntegerv	Z+	0	vertex-array
VERTEX_ARRAY_RANGE_VALID_NV	GetBooleanv	B	False	vertex-array

New Implementation Dependent State

Get Value	Get Command	Type	Minimum Value
MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV	GetIntegerv	Z+	65535

NV10 Implementation Details

This section describes implementation-defined limits for NV10:

The value of MAX_VERTEX_ARRAY_RANGE_ELEMENT_NV is 65535.

This section describes bugs in the NV10 vertex array range. These bugs will be fixed in a future hardware release:

If VERTEX_ARRAY is enabled with a format of GL_SHORT and the vertex array range is valid, a vertex array vertex with an X, Y, Z, or W coordinate of -32768 is wrongly interpreted as zero. Example: the X,Y coordinate (-32768,-32768) is incorrectly read as (0,0) from the vertex array.

If TEXTURE_COORD_ARRAY is enabled with a format of GL_SHORT and the vertex array range is valid, a vertex array texture coord with an S, T, R, or Q coordinate of -32768 is wrongly interpreted as zero. Example: the S,T coordinate (-32768,-32768) is incorrectly read as (0,0) from the texture coord array.

This section describes the implementation-dependent validity checks for NV10.

- For the NV10 implementation-dependent validity check for the vertex array range region of memory to be true, all of the following must be true:

1. The <pointer> must be 32-byte aligned.

2. The underlying memory types must all be the same (all standard system memory -OR- all AGP memory -OR- all video memory).

- For the NV10 implementation-dependent validity check for the vertex array state to be true, all of the following must be true:

```
1.  ( VERTEX_ARRAY must be enabled -AND-
      The vertex array stride must be less than 256 -AND-
      ( ( The vertex array type must be FLOAT -AND-
          The vertex array stride must be a multiple of 4 bytes -AND-
          The vertex array pointer must be 4-byte aligned -AND-
          The vertex array size must be 2, 3, or 4 ) -OR-
        ( The vertex array type must be SHORT -AND-
          The vertex array stride must be a multiple of 4 bytes -AND-
          The vertex array pointer must be 4-byte aligned. -AND-
          The vertex array size must be 2 ) -OR-
        ( The vertex array type must be SHORT -AND-
          The vertex array stride must be a multiple of 8 bytes -AND-
          The vertex array pointer must be 8-byte aligned. -AND-
          The vertex array size must be 4 ) -OR-
        ( The vertex array type must be SHORT -AND-
          The vertex array stride must be a multiple of 8 bytes -AND-
          The vertex array pointer must be 8-byte aligned. )
          The vertex array stride must non-zero -AND-
          The vertex array size must be 3 ) ) )

2.  ( NORMAL_ARRAY must be disabled. ) -OR -
    ( NORMAL_ARRAY must be enabled -AND-
      The normal array size must be 3 -AND-
      The normal array stride must be less than 256 -AND-
      ( ( The normal array type must be FLOAT -AND-
          The normal array stride must be a multiple of 4 bytes -AND-
          The normal array pointer must be 4-byte aligned. ) -OR-
        ( The normal array type must be SHORT -AND-
          The normal array stride must be a multiple of 8 bytes -AND-
          The normal array stride must non-zero -AND-
          The normal array pointer must be 8-byte aligned. ) ) )

3.  ( COLOR_ARRAY must be disabled. ) -OR -
    ( COLOR_ARRAY must be enabled -AND-
      The color array type must be FLOAT or UNSIGNED_BYTE -AND-
      The color array stride must be a multiple of 4 bytes -AND-
      The color array stride must be less than 256 -AND-
      The color array pointer must be 4-byte aligned -AND-
   ( ( The color array size must be 3 -AND-
          The color array stride must non-zero ) -OR-
     ( The color array size must be 4 ) )

4.  ( SECONDARY_COLOR_ARRAY must be disabled. ) -OR -
    ( SECONDARY_COLOR_ARRAY must be enabled -AND-
      The secondary color array type must be FLOAT or UNSIGNED_BYTE -AND-
      The secondary color array stride must be a multiple of 4 bytes -AND-
      The secondary color array stride must be less than 256 -AND-
      The secondary color array pointer must be 4-byte aligned -AND-
   ( ( The secondary color array size must be 3 -AND-
          The secondary color array stride must non-zero ) -OR-
        ( The secondary color array size must be 4 ) )

   5.  For texture units zero and one:

       ( TEXTURE_COORD_ARRAY must be disabled. ) -OR -
       ( TEXTURE_COORD_ARRAY must be enabled -AND-
         The texture coord array stride must be less than 256 -AND-
         ( ( The texture coord array type must be FLOAT -AND-
             The texture coord array pointer must be 4-byte aligned. )
             The texture coord array stride must be a multiple of 4 bytes -AND-
             The texture coord array size must be 1, 2, 3, or 4 ) -OR-
           ( The texture coord array type must be SHORT -AND-
             The texture coord array pointer must be 4-byte aligned. )
             The texture coord array stride must be a multiple of 4 bytes -AND-
             The texture coord array stride must non-zero -AND-
             The texture coord array size must be 1 ) -OR-
           ( The texture coord array type must be SHORT -AND-
             The texture coord array pointer must be 4-byte aligned. )
             The texture coord array stride must be a multiple of 4 bytes -AND-
             The texture coord array size must be 2 ) -OR-
           ( The texture coord array type must be SHORT -AND-
             The texture coord array pointer must be 8-byte aligned. )
             The texture coord array stride must be a multiple of 8 bytes -AND-
             The texture coord array stride must non-zero -AND-
             The texture coord array size must be 3 ) -OR-
           ( The texture coord array type must be SHORT -AND-
             The texture coord array pointer must be 8-byte aligned. )
             The texture coord array stride must be a multiple of 8 bytes -AND-
             The texture coord array size must be 4 ) ) )

   6.  ( EDGE_FLAG_ARRAY must be disabled. )

   7.  ( VERTEX_WEIGHT_ARRAY_NV must be disabled. ) -OR -
       ( VERTEX_WEIGHT_ARRAY_NV must be enabled. -AND -
         The vertex weight array type must be FLOAT -AND-
         The vertex weight array size must be 1 -AND-
         The vertex weight array stride must be a multiple of 4 bytes -AND-
         The vertex weight array stride must be less than 256 -AND-
         The vertex weight array pointer must be 4-byte aligned )

   8.  ( FOG_COORDINATE_ARRAY must be disabled. )

o  For the NV10 implementation-dependent validity check based on
   other OpenGL rendering state is FALSE if any of the following are true:

   1.  ( COLOR_LOGIC_OP is enabled -AND-
         The logic op is not COPY )

   2.  ( LIGHT_MODEL_TWO_SIDE is true. )

   3.  Either texture unit is enabled and active with a texture
       with a non-zero border.

   4.  Several other obscure unspecified reasons.
```

Name

SGIS_texture_lod

Name Strings

GL_SGIS_texture_lod

Version

$Date: 1997/05/30 01:34:44 $ $Revision: 1.8 $

Number

24

Dependencies

EXT_texture is required
EXT_texture3D affects the definition of this extension
EXT_texture_object affects the definition of this extension
SGI_detail_texture affects the definition of this extension
SGI_sharpen_texture affects the definition of this extension

Overview

This extension imposes two constraints related to the texture level of detail parameter LOD, which is represented by the Greek character lambda in the GL Specification. One constraint clamps LOD to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the LOD parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

Note: because the shape of the mipmap array is always determined by the dimensions of the level 0 array, this array must be loaded for mipmapping to be active. If the level 0 array is specified with a null image pointer, however, no actual data transfer will take place. And a sufficiently tuned implementation might not even allocate space for a level 0 array so specified until true image data were presented.

Issues

* Should detail and sharpen texture operate when the level 0 image is not being used?

A: Sharpen yes, detail no.

* Should the shape of the mipmap array be determined by the dimensions of the level 0 array, regardless of the base level?

A: Yes, this is the better solution. Driving everything from the base level breaks the proxy query process, and allows mipmap arrays to be placed arbitrarily. The issues of requiring a level 0 array are partially overcome by the use of null-point loads, which avoid data transfer and, potentially, data storage allocation.

* With the arithmetic as it is, a linear filter might access an array past the limit specified by MAX_LEVEL or p. But the results of this access are not significant, because the blend will weight them as zero.

New Procedures and Functions

None

New Tokens

Accepted by the <pname> parameter of TexParameteri, TexParameterf, TexParameteriv, TexParameterfv, GetTexParameteriv, and GetTexParameterfv:

```
TEXTURE_MIN_LOD_SGIS           0x813A
TEXTURE_MAX_LOD_SGIS           0x813B
TEXTURE_BASE_LEVEL_SGIS        0x813C
TEXTURE_MAX_LEVEL_SGIS         0x813D
```

Additions to Chapter 2 of the 1.0 Specification (OpenGL Operation)

None

Additions to Chapter 3 of the 1.0 Specification (Rasterization)

GL Specification Table 3.7 is updated as follows:

Name	Type	Legal Values
TEXTURE_WRAP_S	integer	CLAMP, REPEAT
TEXTURE_WRAP_T	integer	CLAMP, REPEAT
TEXTURE_WRAP_R_EXT	integer	CLAMP, REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR, FILTER4_SGIS
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR, FILTER4_SGIS, LINEAR_DETAIL_SGIS, LINEAR_DETAIL_ALPHA_SGIS, LINEAR_DETAIL_COLOR_SGIS, LINEAR_SHARPEN_SGIS, LINEAR_SHARPEN_ALPHA_SGIS, LINEAR_SHARPEN_COLOR_SGIS
TEXTURE_BORDER_COLOR	4 floats	any 4 values in [0,1]
DETAIL_TEXTURE_LEVEL_SGIS	integer	any non-negative integer
DETAIL_TEXTURE_MODE_SGIS	integer	ADD, MODULATE
TEXTURE_MIN_LOD_SGIS	float	any value
TEXTURE_MAX_LOD_SGIS	float	any value
TEXTURE_BASE_LEVEL_SGIS	integer	any non-negative integer
TEXTURE_MAX_LEVEL_SGIS	integer	any non-negative integer

Table 3.7: Texture parameters and their values.

Base Array

Although it is not explicitly stated, it is the clear intention of the OpenGL specification that texture minification filters NEAREST and LINEAR, and all texture magnification filters, be applied to image array zero. This extension introduces a parameter, BASE_LEVEL, that explicitly specifies which array level is used for these filter operations. Base level is specified for a specific texture by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_BASE_LEVEL_SGIS, and <param> set to (or <params> pointing to) the desired value. The error INVALID_VALUE is generated if the specified BASE_LEVEL is negative.

Level of Detail Clamping

The level of detail parameter LOD is defined in the first paragraph of Section 3.8.1 (Texture Minification) of the GL Specification, where it is represented by the Greek character lambda. This extension redefines the definition of LOD as follows:

```
LOD'(x,y) = log_base_2 (Q(x,y))


       /  MAX_LOD LOD' > MAX_LOD
LOD = (   LOD'          LOD' >= MIN_LOD and LOD' <= MAX_LOD
       \  MIN_LOD LOD' < MIN_LOD
        \ undefined     MIN_LOD > MAX_LOD
```

The variable Q in this definition represents the Greek character rho, as it is used in the OpenGL Specification. (Recall that Q is computed based on the dimensions of the BASE_LEVEL image array.) MIN_LOD is the value of the per-texture variable TEXTURE_MIN_LOD_SGIS, and MAX_LOD is the value of the per-texture variable TEXTURE_MAX_LOD_SGIS.

Initially TEXTURE_MIN_LOD_SGIS and TEXTURE_MAX_LOD_SGIS are -1000 and 1000 respectively, so they do not interfere with the normal operation of texture mapping. These values are respecified for a specific texture by calling TexParameteri, TexParemeterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_MIN_LOD_SGIS or TEXTURE_MAX_LOD_SGIS, and <param> set to (or <params> pointing to) the new value. It is not an error to specify a maximum LOD value that is less than the minimum LOD value, but the resulting LOD values are not defined.

LOD is clamped to the specified range prior to any use. Specifically, the mipmap image array selection described in the Mipmapping Subsection of the GL Specification is based on the clamped LOD value. Also, the determination of whether the minification or magnification filter is used is based on the clamped LOD.

Mipmap Completeness

The GL Specification describes a "complete" set of mipmap image arrays as array levels 0 through p, where p is a well defined function of the dimensions of the level 0 image. This extension modifies the notion of completeness: instead of requiring that all arrays 0 through p meet the requirements, only arrays 0 and arrays BASE_LEVEL through MAX_LEVEL (or p, whichever is smaller) must meet these requirements. The specification of BASE_LEVEL was described above. MAX_LEVEL is specified by calling TexParameteri, TexParemeterf, TexParameteriv, or TexParameterfv with <target> set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT, <pname> set to TEXTURE_MAX_LEVEL_SGIS, and <param> set to (or <params> pointing to) the desired value. The error INVALID_VALUE is generated if the specified MAX_LEVEL is negative. If MAX_LEVEL is smaller than BASE_LEVEL, or if BASE_LEVEL is greater than p, the set of arrays is incomplete.

Array Selection

Magnification and non-mipmapped minification are always performed using only the BASE_LEVEL image array. If the minification filter is one that requires mipmapping, one or two array levels are selected using the equations in the table below, and the LOD value is clamped to a maximum value that insures that no array beyond the limits specified by MAX_LEVEL and p is accessed.

Minification Filter	Maximum LOD	Array level(s)
NEAREST_MIPMAP_NEAREST	M + 0.4999	floor(B + 0.5)
LINEAR_MIPMAP_NEAREST	M + 0.4999	floor(B + 0.5)
NEAREST_MIPMAP_LINEAR	M	floor(B), floor(B)+1
LINEAR_MIPMAP_LINEAR	M	floor(B), floor(B)+1

where:

M = min(MAX_LEVEL,p) - BASE_LEVEL
B = BASE_LEVEL + LOD

For NEAREST_MIPMAP_NEAREST and LINEAR_MIPMAP_NEAREST the specified image array is filtered according to the rules for NEAREST or LINEAR respectively. For NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR both selected arrays are filtered according to the rules for NEAREST or LINEAR, respectively. The resulting values are then blended as described in the Mipmapping section of the OpenGL specification.

Additional Filters

Sharpen filters (described in SGIS_sharpen_texture) operate on array levels BASE_LEVEL and BASE_LEVEL+1. If the minimum of MAX_LEVEL and p is not greater than BASE_LEVEL, then sharpen texture reverts to a LINEAR magnification filter. Detail filters (described in SGIS_detail_texture) operate only when BASE_LEVEL is zero.

Texture Capacity

In Section 3.8 the OpenGL specification states:

> "In order to allow the client to meaningfully query the maximum image array sizes that are supported, an implementation must not allow an image array of level one or greater to be created if a ‘complete’ set of image arrays consistent with the requested array could not be supported."

Given this extension’s redefinition of completeness, the above paragraph should be rewritten to indicate that all levels of the ‘complete’ set of arrays must be supportable. E.g.

> "In order to allow the client to meaningfully query the maximum image array sizes that are supported, an implementation must not allow an image array of level one or greater to be created if a ‘complete’ set of image arrays (all levels 0 through p) consistent with the requested array could not be supported."

Additions to Chapter 4 of the 1.0 Specification (Per-Fragment Operations and the Frame Buffer)

None

Additions to Chapter 5 of the 1.0 Specification (Special Functions)

None

Additions to Chapter 6 of the 1.0 Specification (State and State Requests)

None

Additions to the GLX Specification

None

Dependencies on EXT_texture

EXT_texture is required.

Dependencies on EXT_texture3D

If EXT_texture3D is not supported, references to 3D texture mapping and to TEXTURE_3D_EXT in this document are invalid and should be ignored.

Dependencies on EXT_texture_object

If EXT_texture_object is implemented, the state values named

TEXTURE_MIN_LOD_SGIS
TEXTURE_MAX_LOD_SGIS
TEXTURE_BASE_LEVEL_SGIS
TEXTURE_MAX_LEVEL_SGIS

are added to the state vector of each texture object. When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound textures have their LOD and LEVEL parameters restored to their pushed values.

Dependencies on SGIS_detail_texture

If SGIS_detail_texture is not supported, references to detail texture mapping in this document are invalid and should be ignored.

Dependencies on SGIS_sharpen_texture

If SGIS_sharpen_texture is not supported, references to sharpen texture mapping in this document are invalid and should be ignored.

Errors

INVALID_VALUE is generated if an attempt is made to set TEXTURE_BASE_LEVEL_SGIS or TEXTURE_MAX_LEVEL_SGIS to a negative value.

New State

```
                                                    Initial
    Get Value                Get Command         Type    Value   Attrib
    ---------                -----------         ----    ------  ------
    TEXTURE_MIN_LOD_SGIS     GetTexParameterfv   n x R   -1000   texture
    TEXTURE_MAX_LOD_SGIS     GetTexParameterfv   n x R    1000   texture
    TEXTURE_BASE_LEVEL_SGIS  GetTexParameteriv   n x R       0   texture
    TEXTURE_MAX_LEVEL_SGIS   GetTexParameteriv   n x R    1000   texture
```

New Implementation Dependent State

None

Name

    EXT_swap_control

Name Strings

    WGL_EXT_swap_control

Version

    Date: 1/27/1999   Revision: 1.3

Number

    172

Dependencies

    WGL_EXT_extensions_string is required.

Overview

    This extension allows an application to specify a minimum periodicity
    of color buffer swaps, measured in video frame periods.

New Procedures and Functions

```
BOOL wglSwapIntervalEXT(int interval)

int wglGetSwapIntervalEXT(void)
```

New Tokens

    None

Additions to Chapter 2 of the 1.2 GL Specification (OpenGL Operation)

    None

Additions to Chapter 3 of the 1.2 GL Specification (Rasterization)

    None

Additions to Chapter 4 of the 1.2 GL Specification (Per-Fragment Operations and the Framebuffer)

    None

Additions to Chapter 5 of the 1.2 GL Specification (Special Functions)

    None

Additions to Chapter 6 of the 1.2 GL Specification (State and State Requests)

    None

Additions to the WGL Specification

wglSwapIntervalEXT specifies the minimum number of video frame periods per buffer swap for the window associated with the current context. The interval takes effect when SwapBuffers or wglSwapLayerBuffer is first called subsequent to the wglSwapIntervalEXT call.

The parameter ’interval’ specifies the minimum number of video frames that are displayed before a buffer swap will occur.

A video frame period is the time required by the monitor to display a full frame of video data. In the case of an interlaced monitor, this is typically the time required to display both the even and odd fields of a frame of video data. An interval set to a value of 2 means that the color buffers will be swapped at most every other video frame.

If ’interval’ is set to a value of 0, buffer swaps are not synchronized to a video frame. The ’interval’ value is silently clamped to the maximum implementation-dependent value supported before being stored.

The swap interval is not part of the render context state. It cannot be pushed or popped. The current swap interval for the window associated with the current context can be obtained by calling wglGetSwapIntervalEXT. The default swap interval is 1.

Because there is no way to extend wgl, this call is defined in the ICD and can be called by obtaining the address with wglGetProcAddress. Because this is not a GL extension, it is not included in the GL_EXTENSIONS string.

Errors

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

ERROR_INVALID_DATA	The ’interval’ parameter is negative.

New State

None

New Implementation Dependent State

None