3D*labs*®

Software Release Note

Windows NT 3.51 & 4.0 Display & OpenGL Drivers for GLINT & PERMEDIA

OpenGL Extension Appendices

This note collects together appendices for various extensions supported by the Windows NT 3.51 & Windows NT 4.0 Display Driver and OpenGL Installable Client Driver for the GLINT 300SX/500TX reference board (Montserrat), the 300SX/500TX/MX + DELTA board (Racer) and the PERMEDIA reference boards with and without Delta.

Change History

Issue	Date	Change
1	10-Dec-96	New Release
2	13-Jun-97	MX/P2 Release

Document r20ext.doc

3D*labs Inc.*
181, Metro Drive Suite 520
San Jose, CA 95110
United States

Tel: (408) 436 3455
Fax: (408) 436 3458

3D*labs Ltd.*
Meadlake Place
Thorpe Lea Road. Egham
Surrey, TW20 8HE
United Kingdom

Tel: +44 (0) 1784 470555
Fax: +44 (0) 1784 470699

APPENDIX A
OpenGL Overlay Planes Extension for Windows NT

Hock San Lee

hockl@microsoft.com

Overview

Overlay and underlay planes are used in many high-end 3-D graphics applications. This extension allows such applications to use the hardware overlay and underlay planes in Windows NT 3.51.

In Windows NT, the pixel formats describe the pixel configurations of a graphics device. Each pixel format includes a description of the depth and other characteristics of the main color buffers and also tells what additional buffers (depth, accumulation, stencil, auxiliary) are associated with the main plane. The pixel format is hereby extended to include information about the overlay and underlay buffers that are available through the pixel format.

A layer plane has at least a front-left color buffer, and may optionally include right and/or back color buffers. It may share depth, accumulation, and stencil buffers with the main plane.

A layer plane specific rendering context is used to render into the layer buffers. GDI drawing is not available in layer planes.

The layer color buffers are managed as part of a normal window similar to the main color buffers. Each window has its own layer color buffers. Multiple windows with overlay and/or underlay planes can be displayed at the same time. The extension requires no additional window management and modifications to the existing clip change notification mechanism of the DDI. This reduces complexity and allows the extension to be supported in Windows NT 3.51 with minimal changes. The extension does not support free floating overlay windows that can move over any window in the main drawing plane. Such free floating windows would require additional overlay window management or access serialization in applications. The extension also does not support hardware pop-up plane that has no transparent color because it would obscure underlying planes in the window at all times.

Each layer plane in a window has an associated palette. The palette of a color-index layer plane can be set by the application, while the palette of an RGBA color plane is fixed. It is the responsibility of the application to realize the palette when the window is in the foreground. In addition, it has a transparent pixel color or index that allows the underlying planes to show through the layer plane.

The state of a rendering context can be copied to another rendering context in a different layer plane. Display lists can also be shared by rendering contexts in different layer planes.

New Procedures and Functions

```
BOOL    wglDescribeLayerPlane(hdc, iPixelFormat, iLayerPlane, nBytes, plpd)
HDC     hdc;
int     iPixelFormat;
int     iLayerPlane;
UINT    nBytes;
LPLAYERPLANEDESCRIPTOR plpd;

int     wglSetLayerPaletteEntries(hdc, iLayerPlane, iStart, cEntries, pcr)
HDC     hdc;
int     iLayerPlane;
int     iStart;
int     cEntries;
CONST COLORREF *pcr;

int     wglGetLayerPaletteEntries(hdc, iLayerPlane, iStart, cEntries, pcr)
```

```
HDC     hdc;
int     iLayerPlane;
int     iStart;
int     cEntries;
COLORREF *pcr;

BOOL    wglRealizeLayerPalette(hdc, iLayerPlane, bRealize)
HDC     hdc;
int     iLayerPlane;
BOOL    bRealize;

BOOL    wglSwapLayerBuffers(hdc, fuPlanes)
HDC     hdc;
UINT    fuPlanes;

HGLRC wglCreateLayerContext(hdc, iLayerPlane)
HDC     hdc;
int     iLayerPlane;

BOOL    wglCopyContext (hglrcSrc, hglrcDst, mask)
HGLRC hglrcSrc;
HGLRC hglrcDst;
UINT    mask;
```

New Tokens and Data Structures

```
typedef struct tagLAYERPLANEDESCRIPTOR { // lpd
    WORD        nSize;
    WORD        nVersion;
    DWORD       dwFlags;
    BYTE        iPixelType;
    BYTE        cColorBits;
    BYTE        cRedBits;
    BYTE        cRedShift;
    BYTE        cGreenBits;
    BYTE        cGreenShift;
    BYTE        cBlueBits;
    BYTE        cBlueShift;
    BYTE        cAlphaBits;
    BYTE        cAlphaShift;
    BYTE        cAccumBits;
    BYTE        cAccumRedBits;
    BYTE        cAccumGreenBits;
```

```
    BYTE        cAccumBlueBits;
    BYTE        cAccumAlphaBits;
    BYTE        cDepthBits;
    BYTE        cStencilBits;
    BYTE        cAuxBuffers;
    BYTE        iLayerPlane;
    BYTE        bReserved;
    COLORREF    crTransparent;
} LAYERPLANEDESCRIPTOR;
```

Accepted by the dwFlags field of LAYERPLANEDESCRIPTOR:

LPD_SUPPORT_OPENGL
LPD_SUPPORT_GDI
LPD_DOUBLEBUFFER
LPD_STEREO
LPD_SWAP_EXCHANGE
LPD_SWAP_COPY
LPD_TRANSPARENT
LPD_SHARE_DEPTH
LPD_SHARE_STENCIL
LPD_SHARE_ACCUM

Accepted by the iPixelType field of LAYERPLANEDESCRIPTOR:

LPD_TYPE_RGBA
LPD_TYPE_COLORINDEX

Accepted by the dwFlags field of PIXELFORMATDESCRIPTOR:

PFD_SWAP_LAYER_BUFFERS

Accepted by the fuPlanes parameter of wglSwapLayerBuffers:

WGL_SWAP_MAIN_PLANE
WGL_SWAP_OVERLAYi, where i is between 1 and 15.
WGL_SWAP_UNDERLAYi, where i is between 1 and 15.

Pixel Format Descriptor

A pixel format identifies a particular buffer configuration of a graphics device, and is described by the pixel format descriptor. The pixel format descriptor is extended to include the number of overlay and underlay planes in the pixel format. Two pixel formats that have identical main planes but different layer planes are now two distinct pixel formats. Previously, they were both identified in a single pixel format when overlay was not supported.

Each pixel format can include up to 15 overlay planes and up to 15 underlay planes. All pixel formats must include the main color plane.

If a pixel format includes an underlay plane, the pixel format must provide a transparent pixel color or index, depending on the pixel type, that allows the pixels of the underlying plane to show through the main color plane. All other layer planes except the lowest numbered underlay plane must provide a transparent pixel color or index, depending on the pixel type of the overlay or underlay plane.

A window can be set up to use a particular pixel format. All planes identified in the pixel format, including the overlay and underlay planes, are part of the regular window. Applications can create multiple windows each with a different layer plane configuration. These windows can be displayed at the same time. There is no additional management for these windows other than the normal window management. The layer planes share the same main window region and are updated at the same time as the main window. Free floating overlay windows that can move over any window in the main drawing plane are not supported.

The depth, accumulation and stencil buffers may be shared by all layer planes in a window.

Changes to PIXELFORMATDESCRIPTOR

dwFlags	Add the following flag PFD_SWAP_LAYER_BUFFERS Indicates whether the device can swap individual layer planes other than all layer planes as a group if the pixel format includes double-buffered overlay or underlay planes. If this flag is not set, the wglSwapLayerBuffers function is not supported.
iLayerType	Ignored.
bReserved	Count of up to 15 overlay planes (bit 0-3) and up to 15 underlay planes (bit 4-7)
dwLayerMask	Ignored.
dwVisibleMask	Specifies the transparent color or index if there is an underlay plane. If the pixel type is RGBA, it is a transparent RGB color. If the pixel type is color index, it is a transparent index.
dwDamageMask	Ignored.

Examining Layer Planes of a Pixel Format

The wglDescribeLayerPlane function obtains information about the layer planes of a given pixel format.

wglDescribeLayerPlane

```
BOOL   wglDescribeLayerPlane(hdc, iPixelFormat, iLayerPlane, nBytes, plpd)
HDC    hdc;
int    iPixelFormat;
int    iLayerPlane;
UINT   nBytes;
LPLAYERPLANEDESCRIPTOR plpd;
```

The wglDescribeLayerPlane function obtains information about an overlay or underlay plane of a given pixel format of the device associated with hdc. The function sets the members of the LAYERPLANEDESCRIPTOR structure pointed to by plpd with that layer plane information.

Parameters

hdc

Specifies the device context.

iPixelFormat

Specifies the pixel format whose layer plane is of interest.

iLayerPlane

Identifies the overlay or underlay plane of interest. The value 1 identifies the first overlay plane over the main plane, 2 the second overlay plane over the first overlay plane and the main plane, and so on. The value -1 identifies the first underlay plane under the main plane, -2 the second underlay plane under the first underlay plane and the main plane, and so on. The number of overlay and underlay planes is given in the bReserved field of the pixel format descriptor.

nBytes

Specifies the size, in bytes, of the structure pointed to by plpd. The function stores data to that structure; it stores no more than nBytes bytes. Set this value to sizeof(LAYERPLANEDESCRIPTOR).

plpd

Pointer to a LAYERPLANEDESCRIPTOR structure whose members the function sets with layer plane data. The function stores the number of bytes copied to the structure in the structure's nSize field.

Return Value

If the function succeeds, the return value is TRUE and the function sets the members of the LAYERPLANEDESCRIPTOR structure pointed to by plpd according to the specified layer plane of the given pixel format. If the function fails, the return value is FALSE. Call GetLastError for extended error information.

Comments

The layer planes are ordered by the plane number. Higher numbered planes overlay lower numbered planes.

See Also

DescribePixelFormat, wglCreateLayerContext, PIXELFORMATDESCRIPTOR, LAYERPLANEDESCRIPTOR

Layer Plane Descriptor

```
typedef struct tagLAYERPLANEDESCRIPTOR { // lpd
    WORD     nSize;
    WORD     nVersion;
    DWORD    dwFlags;
    BYTE     iPixelType;
    BYTE     cColorBits;
    BYTE     cRedBits;
    BYTE     cRedShift;
    BYTE     cGreenBits;
    BYTE     cGreenShift;
    BYTE     cBlueBits;
    BYTE     cBlueShift;
    BYTE     cAlphaBits;
    BYTE     cAlphaShift;
    BYTE     cAccumBits;
    BYTE     cAccumRedBits;
    BYTE     cAccumGreenBits;
    BYTE     cAccumBlueBits;
    BYTE     cAccumAlphaBits;
    BYTE     cDepthBits;
```

```
    BYTE        cStencilBits;
    BYTE        cAuxBuffers;
    BYTE        iLayerPlane;
    BYTE        bReserved;
    COLORREF    crTransparent;
} LAYERPLANEDESCRIPTOR;
```

The LAYERPLANEDESCRIPTOR structure describes a layer plane of a pixel format.

Members

nSize
Specifies the size of this data structure. This value should be set to sizeof(LAYERPLANEDESCRIPTOR).

nVersion
Specifies the version of this data structure. This value should be set to 1.

dwFlags
A set of flags that specify the properties of the layer plane.

LPD_SUPPORT_OPENGL
The layer plane supports OpenGL drawing.

LPD_SUPPORT_GDI
The layer plane supports GDI drawing. This flag is not supported in the current implementation.

LPD_DOUBLEBUFFER
The layer plane is double-buffered. A layer plane may be double-buffered even if the main plane is single-buffered and vice versa.

LPD_STEREO
The layer plane is stereoscopic. A layer plane may be stereoscopic even if the main plane is monoscopic and vice versa.

LPD_SWAP_EXCHANGE
In a double-buffered layer plane, swapping the color buffers causes the back buffer and the front buffer contents to be exchanged. Following the swap, the back buffer contains the front buffer content before the swap. This flag is only a hint and may not be provided by the driver.

LPD_SWAP_COPY
In a double-buffered layer plane, swapping the color buffers causes the back buffer content to be copied to the front buffer. The back buffer content is not affected by the swap. This flag is only a hint and may not be provided by the driver.

LPD_TRANSPARENT
There is a transparent color or index value, as given in the crTransparent field of this structure, that allows underlying layer pixels to show through this layer. A transparent color or index exists for all layer planes except the lowest numbered underlay plane.

LPD_SHARE_DEPTH
The layer plane shares the depth buffer with the main plane.

LPD_SHARE_STENCIL
The layer plane shares the stencil buffer with the main plane.

LPD_SHARE_ACCUM
The layer plane shares the accumulation buffer with the main plane.

iPixelType
Specifies the type of pixel data. The following types are defined:

Value	Meaning
LPD_TYPE_RGBA	RGBA pixels. Each pixel has four components: red, green, blue, and alpha.
LPD_TYPE_COLORINDEX	Color index pixels. Each pixel uses a color index value.

cColorBits
Specifies the number of color bitplanes in each color buffer in the layer plane. For RGBA pixel types, it is the size of the color buffer excluding the alpha bitplanes. For color index pixels, it is the size of the color index buffer.

cRedBits
Specifies the number of red bitplanes in each RGBA color buffer.

cRedShift
Specifies the shift count for red bitplanes in each RGBA color buffer.

cGreenBits
Specifies the number of green bitplanes in each RGBA color buffer.

cGreenShift
Specifies the shift count for green bitplanes in each RGBA color buffer.

cBlueBits
Specifies the number of blue bitplanes in each RGBA color buffer.

cBlueShift
Specifies the shift count for blue bitplanes in each RGBA color buffer.

cAlphaBits
Specifies the number of alpha bitplanes in each RGBA color buffer.

cAlphaShift
Specifies the shift count for alpha bitplanes in each RGBA color buffer.

cAccumBits
Not used. Must be zero.

cAccumRedBits
Not used. Must be zero.

cAccumGreenBits
Not used. Must be zero.

cAccumBlueBits
Not used. Must be zero.

cAccumAlphaBits
Not used. Must be zero.

cDepthBits
Not used. Must be zero.

cStencilBits
Not used. Must be zero.

cAuxBuffers
Specifies the number of auxiliary buffers.

iLayerPlane
Specifies the layer plane number. The value 1 identifies the first overlay plane over the main plane, 2 the second overlay plane over the first overlay plane and the main plane, and so on. The value -1 identifies the first underlay plane under the main plane, -2 the second underlay plane under the first underlay plane and the main plane, and so on.

bReserved
Not used. Must be zero.

crTransparent
Specifies the transparent color or index value if the LPD_TRANSPARENT flag is set. Typically, this value is 0.

Managing Layer Palette Entries

Each color-index layer plane in a window has an associated palette that can be set with wglSetLayerPaletteEntries. Initially, the layer palette contains white colors. The palette entries can be retrieved by calling wglGetLayerPaletteEntries. They are mapped into the physical palette in wglRealizeLayerPalette. The palette of the main plane is managed by GDI.

Although a RGBA layer plane does not have a settable palette, its palette must be initialized in wglRealizeLayerPalette.

wglSetLayerPaletteEntries

```
int       wglSetLayerPaletteEntries(hdc, iLayerPlane, iStart, cEntries, pcr)
HDC       hdc;
int       iLayerPlane;
int       iStart;
int       cEntries;
CONST COLORREF *pcr;
```

The wglSetLayerPaletteEntries function sets the palette entries in a given color-index layer plane of the window referenced by hdc.

Parameters

hdc
Specifies a device context of the window whose layer's palette is to be modified.

iLayerPlane
Identifies the overlay or underlay plane of interest. The value 1 identifies the first overlay plane over the main plane, 2 the second overlay plane over the first overlay plane and the main plane, and so on. The value -1 identifies the first underlay plane under the main plane, -2 the second underlay plane under the first underlay plane and the main plane, and so on. The number of overlay and underlay planes is given in the bReserved field of the pixel format descriptor.

iStart
Specifies the first palette entry to be set.

cEntries
Specifies the number of palette entries to be set.

pcr
Points to the first member of an array of cEntries COLORREF structures containing the RGB values.

Return Value

If the function succeeds, the return value is the number of entries that were set in the palette in the layer plane of the window. If the function fails or no pixel format has been selected, the return value is zero. To get extended error information, call GetLastError.

Comments

Each color-index layer plane in a window has a palette whose size is determined by the number of color bitplanes. If there are n bitplanes in the layer plane, the size of its palette is 2 ^ n. The palette entry corresponding to the transparent index, if it exists, cannot be modified. The layer palette is realized in wglRealizeLayerPalette.

Initially, the layer palette contains white colors.

The wglSetLayerPaletteEntries function does not set the palette entries of the main plane. Use GDI palette functions to update the main plane palette.

See Also

wglRealizeLayerPalette, wglGetLayerPaletteEntries, wglDescribeLayerPlane, LAYERPLANEDESCRIPTOR

wglGetLayerPaletteEntries

```
int       wglGetLayerPaletteEntries(hdc, iLayerPlane, iStart, cEntries, pcr)
HDC       hdc;
int       iLayerPlane;
int       iStart;
int       cEntries;
COLORREF *pcr;
```

The wglGetLayerPaletteEntries function retrieves the palette entries from a given color-index layer plane of the window referenced by hdc.

Parameters

hdc
Specifies a device context of the window whose layer's palette is to be retrieved.

iLayerPlane
Identifies the overlay or underlay plane of interest. The value 1 identifies the first overlay plane over the main plane, 2 the second overlay plane over the first overlay plane and the main plane, and so on. The value -1 identifies the first underlay plane under the main plane, -2 the second underlay plane under the first underlay plane and the main plane, and so on. The number of overlay and underlay planes is given in the bReserved field of the pixel format descriptor.

iStart
Specifies the first palette entry to be retrieved.

cEntries
Specifies the number of palette entries to be retrieved.

pcr
Points to an array of COLORREF structures to receive the palette RGB values. The array must contain at least as many structures as specified by the cEntries parameter.

Return Value

If the function succeeds, the return value is the number of palette colors retrieved from the layer plane of the window. If the function fails or no pixel format has been selected, the return value is zero. To get extended error information, call GetLastError.

Comments

The RGB value of the palette entry corresponding to the transparent index, if it exists, is undefined.

Initially, the layer palette contains white colors.

The wglGetLayerPaletteEntries function does not retrieve the palette entries of the main plane. Use GDI palette functions to retrieve the main plane palette.

See Also

wglSetLayerPaletteEntries

wglRealizeLayerPalette

```
BOOL      wglRealizeLayerPalette(hdc, iLayerPlane, bRealize)
HDC       hdc;
int       iLayerPlane;
BOOL      bRealize;
```

The wglRealizeLayerPalette function maps palette entries from a given color-index layer plane into the physical palette or initializes the palette of a RGBA layer plane.

Parameters

hdc

Specifies a device context of the window whose layer's palette is to be realized into the physical palette.

iLayerPlane

Identifies the overlay or underlay plane of interest. The value 1 identifies the first overlay plane over the main plane, 2 the second overlay plane over the first overlay plane and the main plane, and so on. The value -1 identifies the first underlay plane under the main plane, -2 the second underlay plane under the first underlay plane and the main plane, and so on. The number of overlay and underlay planes is given in the bReserved field of the pixel format descriptor.

bRealize

Indicates whether the palette is to be realized in the physical palette. If bRealize is TRUE, the palette entries are mapped into the physical palette where available. If bRealize is FALSE, the palette entries for the layer plane of the window are no longer needed and may be released for use by a different foreground window.

Return Value

If the function succeeds, the return value is TRUE. The function returns success even if bRealize is TRUE and the physical palette is unavailable at the time. If the function fails or no pixel format has been selected, the return value is FALSE. To get extended error information, call GetLastError.

Comments

The physical palette for a layer plane is a shared resource among windows with layer planes. If more than one window realizes a palette for a given physical layer plane, only one palette is realized at a time. The layer palette of a foreground window is always realized when wglRealizeLayerPalette is called.

When a window's layer palette is realized, its palette entries are always mapped 1-1 into the physical palette. Unlike GDI logical palettes, the system makes no attempt to map other windows' layer palettes to the current physical palette.

Whenever a window becomes foreground, it should call wglRealizeLayerPalette to re-realize its layer palettes even if the pixel type of the layer plane is RGBA.

The wglRealizeLayerPalette function does not realize the palette entries of the main plane. Use GDI palette functions to realize the main plane palette.

See Also

wglSetLayerPaletteEntries

Swapping Layer Plane Buffers

The layer plane color buffers are swapped in wglSwapLayerBuffers.

wglSwapLayerBuffers

```
BOOL   wglSwapLayerBuffers(hdc, fuPlanes)
HDC    hdc;
UINT   fuPlanes;
```

The wglSwapLayerBuffers function swaps the front and back buffers in the overlay, underlay, and main planes of the window referenced by the specified device context.

Parameters

hdc

Specifies a device context of the window whose layer planes are of interest.

fuPlanes

Specifies the overlay, underlay, and main planes whose front and back buffers are to be swapped. The number of overlay and underlay planes is given in the bReserved field of the pixel format descriptor. This parameter is a bitwise combination of the following values:

Value	Description
WGL_SWAP_MAIN_PLANE	Swaps the front and back buffers of the main plane.
WGL_SWAP_OVERLAYi	Swaps the front and back buffers of the overlay plane i,

	where i is between 1 and 15. WGL_SWAP_OVERLAY1 identifies the first overlay plane over the main plane, WGL_SWAP_OVERLAY2 the second overlay plane over the first overlay plane and the main plane, and so on.
WGL_SWAP_UNDERLAYi	Swaps the front and back buffers of the underlay plane i, where i is between 1 and 15. WGL_SWAP_UNDERLAY1 identifies the first underlay plane under the main plane, WGL_SWAP_UNDERLAY2 the second underlay plane under the first underlay plane and the main plane, and so on.

Return Value

If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. Call GetLastError for extended error information.

Comments

If the layer plane does not include a back buffer, then this call has no effect on that layer plane. The state of the back buffer content following the swap is given in the corresponding layer plane descriptor of the layer plane or the pixel format descriptor of the main plane.

The wglSwapLayerBuffers function swaps the front and back buffers in the specified layer planes simultaneously.

Not all devices support swapping layer planes individually other than all layer planes as a group. This is indicated by the PFD_SWAP_LAYER_BUFFERS flag in the pixel format descriptor.

A multithreaded application should flush the drawing commands in any other threads drawing to the same window before calling the wglSwapLayerBuffers function.

See Also

SwapBuffers, LAYERPLANEDESCRIPTOR, PIXELFORMATDESCRIPTOR

SwapBuffers

Add to Comments

If the window includes double-buffered overlay and underlay planes, SwapBuffers swaps the front and back buffers of all planes simultaneously. Use the wglSwapLayerBuffers function to swap individual layer plane buffers.

Rendering in Layer Planes

A layer plane specific rendering context is used to render into the layer buffers. GDI drawing is not available in layer planes.

wglCreateLayerContext

```
HGLRC wglCreateLayerContext(hdc, iLayerPlane)
HDC     hdc;
int     iLayerPlane;
```

The wglCreateLayerContext function creates a new OpenGL rendering context. The rendering context is suitable for drawing to the specified layer plane on the device referenced by hdc. The rendering context has the same pixel format as the device context.

Parameters

hdc

Handle to a device context. The function creates an OpenGL rendering context suitable for the device the device context references.

iLayerPlane

Identifies the layer plane to which the rendering context is bound. The value 0 identifies the main plane. The value 1 identifies the first overlay plane over the main plane, 2 the second overlay plane over the first overlay plane and the main plane, and so on. The value -1 identifies the first underlay plane under the main plane, -2 the second underlay plane under

the first underlay plane and the main plane, and so on. The number of overlay and underlay planes is given in the bReserved field of the pixel format descriptor.

Return Value

If the function succeeds, the return value is a valid handle to an OpenGL rendering context. If the function fails, the return value is NULL. Call GetLastError for extended error information.

Comments

A rendering context, or GLRC, is a port through which all OpenGL commands pass. Every thread that makes OpenGL calls needs to have a current OpenGL rendering context.

A rendering context is not the same as a device context. A device context contains information pertinent to GDI, and a rendering context contains information pertinent to OpenGL.

An application should set the device context's pixel format before creating a rendering context. See the SetPixelFormat function.

A rendering context can only be used in the specified plane of a window with the same pixel format.

To use OpenGL, an application creates a rendering context, selects it as a thread's current rendering context, then calls OpenGL functions. When the application is finished with the rendering context, it disposes of it by calling wglDeleteContext. Here's a code example:

```
// The following code fragment shows how to render to overlay plane 1.
// This assumes that the pixel format of hdc includes overlay plane 1.

HDC  hdc;
HGLRC  hglrc;

// create a rendering context for overlay plane 1
hglrc = wglCreateLayerContext (hdc, 1);


// make it the calling thread's current rendering context
wglMakeCurrent (hdc, hglrc);

// call OpenGL APIs as desired ...
.
.
.
// when the rendering context is no longer needed ...


// make the rendering context not current
wglMakeCurrent (NULL, NULL) ;


// delete the rendering context
wglDeleteContext (hglrc);
```

See Also

SetPixelFormat, wglCreateContext, wglDeleteContext, wglGetCurrentContext, wglGetCurrentDC, wglMakeCurrent

Synchronizing Rendering State in Layer Planes

The rendering state of a rendering context can be copied to another rendering context in wglCopyContext. Rendering contexts can also share display lists in the existing wglShareLists function.

wglCopyContext

BOOL wglCopyContext (hglrcSrc, hglrcDst, mask)
HGLRC hglrcSrc;
HGLRC hglrcDst;
UINT mask;

The wglCopyContext function copies selected groups of rendering state from one OpenGL rendering context to another.

Parameters

hglrcSrc
Specifies the source OpenGL rendering context.

hglrcDst
Specifies the destination OpenGL rendering context.

mask
Specifies which groups of hglrcSrc rendering state are to be copied to hglrcDst. It contains the bitwise OR of the same symbolic names that are passed to the OpenGL command glPushAttrib. The value GL_ALL_ATTRIB_BITS can be used to copy the maximum possible groups of rendering state.

Return Value

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.

Comments

The wglCopyContext function allows an application to synchronize rendering state of two OpenGL rendering contexts.

You can only copy rendering state between two rendering contexts within the same process. In addition, the rendering contexts must be provided by the same OpenGL implementation. For example, you can always copy rendering state between two rendering contexts of a given pixel format in the same process.

Not all values for OpenGL state can be copied. For example, pixel pack and unpack state, render mode state, and select and feedback state are not copied. The state that can be copied is exactly the state that is manipulated by the OpenGL command glPushAttrib.

The destination rendering context hglrcDst should not be current to any thread when calling wglCopyContext.

See Also

glPushAttrib, wglCreateLayerContext, wglCreateContext, wglShareLists

APPENDIX B
OpenGL Paletted Texture Extension Support

This appendix describes the programming steps required for OpenGL applications to take advantage of the support in the GLINT 500TX for palette textures. The implementation is based upon the extension recently proposed by Microsoft (refer to Appendix C for further details).

In the example code for downloading a 4-bit indexed texture below, note the following:

- Extensions to OpenGL (in this case the functions glColorTableEXT, glGetColorTableEXT etc..) are not directly exported by the OpenGL dynamic link library. Instead the name of the required function (for example "glGetColorTableEXT") is passed as a string to the Win32 function wglGetProcAddress which returns a suitable function pointer . The function is invoked by dereferencing this pointer with the appropriate arguments.

- The GLINT 500TX only supports 1, 2 and 4-bit indexed textures with an on-chip texture LUT of 16 entries of RGBA (with each component in the table stored to 8-bits precision).

- The PERMEDIA only supports 4-bit indexed textures with an on-chip texture LUT of 16 entries of RGB (the alpha component is unused. Each component in the table is stored to 5-bits precision, the bottom 3-bits are ignored internally).

- When downloading the palette texture by calling glTexImage1D or glTexImage2D, the format parameter must be set to GL_COLOR_INDEX and the components parameter to GL_COLOR_INDEX4_EXT (in the case of a 4-bit texture). There is no way at present of presenting the texel indices pre-packed to the OpenGL call (i.e. each index occupies a full data type, e.g. a byte if GL_UNSIGNED_BYTE is specified for the type parameter even though two 4-bit indices could be packed per byte). *However* the texel indices are stored packed after downloading into the local buffer memory.

```
// Add these defines to gl.h
#define GL_COLOR_INDEX1_EXT                     0x80E2
#define GL_COLOR_INDEX2_EXT                     0x80E3
#define GL_COLOR_INDEX4_EXT                     0x80E4

#define GL_COLOR_TABLE_FORMAT_EXT               0x80D8
#define GL_COLOR_TABLE_WIDTH_EXT                0x80D9
#define GL_COLOR_RED_SIZE_EXT                   0x80DA
#define GL_COLOR_GREEN_SIZE_EXT                 0x80DB
#define GL_COLOR_BLUE_SIZE_EXT                  0x80DC
#define GL_COLOR_ALPHA_SIZE_EXT                 0x80DD
```

```
#define GL_COLOR_LUMINANCE_SIZE_EXT                    0x80DE
#define GL_COLOR_INTENSITY_SIZE_EXT                    0x80DF

typedef  void   (APIENTRY * PFNGLCOLORTABLEEXTPROC) ( GLenum  target,
                                                      GLenum  internalformat,
                                                      GLsizei width,
                                                      GLenum  format,
                                                      GLenum  type,
                                                      const void *data );

typedef  void   (APIENTRY * PFNGLCOLORSUBTABLEEXTPROC) ( GLenum  target,
                                                         GLsizei start,
                                                         GLsizei count,
                                                         GLenum  format,
                                                         GLenum  type,
                                                         const void *data );

typedef  void   (APIENTRY * PFNGLGETCOLORTABLEEXTPROC) ( GLenum  target,
                                                         GLenum  format,
                                                         GLenum  type,
                                                         const void *data );

typedef  void   (APIENTRY * PFNGLGETCOLORTABLEPARAMETERIVEXTPROC) ( GLenum  target,
                                                                    Glenum pname,
                                                                    int *params );

typedef  void   (APIENTRY * PFNGLGETCOLORTABLEPARAMETERFVEXTPROC) ( GLenum  target,
                                                                    Glenum pname,
                                                                    float *params );

// define a suitable struct for each entry in the texture lut
typedef struct { GLubyte r, g, b, a; } ubLutEntry;

// allocate a variable to hold the table of lut entries
// note the maximum number of entries on the TX500 is 16
ubLutEntry texLUT[16];

// initialise the lut
// e.g. from black
texLUT[0].r = 0; texLUT[0].g = 0; texLUT[0].b = 0; texLUT[0].a = 255;
texLUT[1]  ...
texLUT[14] ...
// to white etc..
```

```
texLUT[15].r = 255; texLUT[15].g = 255; texLUT[15].b = 255; texLUT[15].a = 255;


// declare a suitable function pointer for updating the texture lut
PFNGLCOLORTABLEEXTPROC  fp_glColorTableEXT;

// bind the function pointer by passing the name of the function as a string
fp_glColorTableEXT = (PFNGLCOLORTABLEEXTPROC) wglGetProcAddress( "glColorTableEXT" );

// download the lut by dereferencing the function pointer
(*fp_glColorTableEXT)(GL_TEXTURE_2D, GL_RGBA, 16, GL_RGBA, GL_UNSIGNED_BYTE, texLUT);

...

// initialise the texture image data
imageWidth  = 256;
imageHeight = 128;

// allocate a byte per texel index
imageBuffer = (GLubyte*) malloc(imageWidth * imageHeight);

...

// in this example download a 4-bit indexed texture
// (but note that each 4-bit (or 2 or 1 bit) index occupies a full byte
// when passed to OpenGL
// but ends up packed 2 (or 4 or 8) to a byte in the TX local buffer)

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexImage2D( GL_TEXTURE_2D, 0,
                  GL_COLOR_INDEX4_EXT,
                  imageWidth,       // must be power of 2 (+ 2*border)
                  imageHeight,      // must be power of 2 (+ 2*border)
                  0,                // no border in this example
                  GL_COLOR_INDEX,   // must be color index format
                  GL_UNSIGNED_BYTE, // byte per index
                  imageBuffer );
...

// now repeated calls to glColorTableEXT followed by a flush will
// instantly update the texture colors without a download
// e.g change all blacks in the texture to yellow:
texLUT[0].r = 255; texLUT[0].g = 255; texLUT[0].b = 0;
(*fp_glColorTableEXT)(GL_TEXTURE_2D, GL_RGBA, 16, GL_RGBA, GL_UNSIGNED_BYTE, texLUT);
```

```
glFlush();
```

This is an example routine for querying the extensions available using glGetString:

```
int checkPaletteTextureEXTAvailable( void )
{
   char seps[]   = " ,";
   char *token;
   static char extStr[128];
   const GLubyte *pExtStr;

   pExtStr = glGetString( GL_EXTENSIONS );
   strcpy(extStr, pExtStr );

   token = strtok( extStr, seps );
   while( token != NULL ) {
      // While there are tokens in "string"
      if (strcmp( token, "GL_EXT_paletted_texture" ) == 0)
          return TRUE;

      // Get next token:
      token = strtok( NULL, seps );
   }

   return FALSE;
}
```

Restrictions

None

APPENDIX C

Windows NT OpenGL Group

OpenGL Paletted Texture Extension

Author: *Drew Bliss (Microsoft Corp)*

Version 0.8 March 1, 1996

Name

EXT_paletted_texture

Name Strings

GL_EXT_paletted_texture

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.

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, and TexImage2D:

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

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*.

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, 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**.

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** or **PROXY_TEXTURE_2D**. *internalformat* specifies the desired format and resolution of the palette when in its internal form. *internalformat* can be any of the 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.

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 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.'

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 processed through the color index pixel transfer modes rather than RGBA. 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.

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.

APPENDIX D

Windows NT OpenGL Group

OpenGL EXT_BGRA Extension Specification

Name

EXT_bgra

Name Strings

GL_EXT_bgra

Dependencies

EXT_abgr affects the definition of this extension

EXT_cmyka affects the definition of this extension

EXT_color_table affects the definition of this extension

EXT_color_subtable affects the definition of this extension

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, TexImage2D, ColorTableEXT and ColorSubTableEXT:

BGR_EXT	0x80E0
BGRA_EXT	0x80E1

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

None.

Additions to Chapter 3 of the GL Specification (Rasterization)

Two entries are 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
CMYK_EXT	Component	Cyan value Magenta value, Yellow value, Black value	Color
CMYKA_EXT	Component	Cyan value, Magenta value, Yellow value, Black 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 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, BGR_EXT, BGRA_EXT, LUMINANCE, LUMINANCE_ALPHA, CMYK_EXT, or CMYKA_EXT, 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, BGR_EXT, BGRA_EXT, LUMINANCE, LUMINANCE_ALPHA, CMYK_EXT, or CMYKA_EXT, 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.

Dependencies on EXT_abgr

If EXT_abgr is not implemented, then references to ABGR_EXT in this specification are void.

Dependencies on EXT_cmyka

If EXT_cmyka is not implemented, then references to CMYK_EXT and CMYKA_EXT in this specification are void.

Dependencies on EXT_color_table

If EXT_color_table is not implemented, then references to ColorTableEXT in this specification are void.

Dependencies on EXT_color_subtable

If EXT_color_subtable is not implemented, then references to ColorSubTableEXT in this specification are void.

Revision History

APPENDIX E
OpenGL Texture Object Extension Support

This appendix describes the programming steps required of OpenGL applications that need to switch between textures without resorting to display lists (in order to avoid downloading the texture data each time glTexImage1D/2D is invoked). The implementation is based upon the texture object extension of version 1.0 and adopted as standard in version 1.1 (refer to Appendix F for further details).

Unlike display list textures which depend on the texture parameter state of the default immediate mode 1D and 2D texture targets, texture objects maintain their own separate copy of all texture state (such as wrap modes, min/mag filters etc. including for palette textures, the colour lut). By selecting a texture object to be the current texture (referred to as 'binding'), all subsequent OpenGL commands such as glTexImage1/2D and glTexParameter will affect the state of that texture object only, until a different texture object is made the new target. The following code fragment should make this process clear.

```
// declare suitable function pointers for calling the texture object routines
typedef void (APIENTRY * PFNGLBINDTEXTUREEXTPROC) (GLenum target, GLuint texture);
typedef void (APIENTRY * PFNGLGENTEXTURESEXTPROC) (GLsizei n, GLuint *textures);
typedef void (APIENTRY * PFNGLDELTEXTURESEXTPROC) (GLsizei n, GLuint *textures);

PFNGLBINDTEXTUREEXTPROC   fpglBindTextureEXT;
PFNGLGENTEXTURESEXTPROC   fpglGenTexturesEXT;
PFNGLDELTEXTURESEXTPROC   fpglDeleteTexturesEXT;

// bind the function pointer by passing the name of the function as a string
fpglBindTextureEXT = (PFNGLBINDTEXTUREEXTPROC) wglGetProcAddress("glBindTextureEXT");
fpglGenTexturesEXT = (PFNGLGENTEXTURESEXTPROC) wglGetProcAddress("glGenTexturesEXT");
fpglDeleteTexturesEXT =
                  (PFNGLDELTEXTURESEXTPROC) glGetProcAddress("glDeleteTexturesEXT");

// texture objects are given unique handles (their name or identifier)
// here we have a 4-bit palette texture, a large non-palette texture (512x256)
// and a small texture (64x64)
GLuint palTexObj;
GLuint largeTexObj;
GLuint smallTexObj;


// obtain a free texture object handle
fpglGenTexturesEXT( 1, &palTexObj );

// make this the current texture target
```

```
// Note the first time we bind to a object handle, OpenGL creates a new texture
// parameter state record in an internal table. Assuming valid texel data
// has been downloaded for this object, subsequent binds will use this texture
// for rendering
fpglBindTextureEXT(GL_TEXTURE_2D, palTexObj);

// download the texels for this object
glTexImage2D( GL_TEXTURE_2D, 0, GL_COLOR_INDEX4_EXT, palTexObjWidth,
              palTexObjHeight, 0, GL_COLOR_INDEX, GL_UNSIGNED_BYTE,
              palTexObjImageBuffer );

// and the palette lut
fpglColorTableEXT( GL_TEXTURE_2D, GL_RGBA, 16, GL_RGBA, GL_UNSIGNED_BYTE,
                   palTexObjLUT );
// set filter modes
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// setup the large texture (3 component)
fpglGenTexturesEXT( 1, &largeTexObj );
fpglBindTextureEXT(GL_TEXTURE_2D, largeTexObj);

glTexImage2D( GL_TEXTURE_2D, 0, 3, largeTexObjWidth, largeTexObjHeight,
              0, GL_RGB, GL_UNSIGNED_BYTE,
              largeTexObjImageBuffer );

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// setup the small texture (4 component with alpha)
fpglGenTexturesEXT( 1, &smallTexObj );
fpglBindTextureEXT(GL_TEXTURE_2D, smallTexObj);

glTexImage2D( GL_TEXTURE_2D, 0, 4, smallTexObjWidth, smallTexObjHeight,
              0, GL_RGBA, GL_UNSIGNED_BYTE,
              smallTexObjImageBuffer );

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// and so on, you get the idea

// now lets switch between textures for rendering
glEnable(GL_TEXTURE_2D);
```

```
fpglBindTextureEXT(GL_TEXTURE_2D, palTexObj);
// all textured primitives will now use the palette texture object
// with nearest-neighbour filtering
myDrawDisplay();

fpglBindTextureEXT(GL_TEXTURE_2D, largeTexObj);
// all textured primitives will now use the large texture object
// with linear filtering
myDrawDisplay();

fpglBindTextureEXT(GL_TEXTURE_2D, smallTexObj);
// all textured primitives will now use the small texture object
// and now back to nearest-neighbour filtering
myDrawDisplay();

// etc.

// free up texture memory when finished
fpglDeleteTexturesEXT( 1, &palTexObj );
fpglDeleteTexturesEXT( 1, &largeTexObj);
fpglDeleteTexturesEXT( 1, &smallTexObj);
```

APPENDIX F
OpenGL Texture Object Extension Specification

Name

EXT_texture_object

Name Strings

GL_EXT_texture_object

Version

$Date: 1995/06/17 03:38:44 $ $Revision: 1.26 $

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.

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 NEAREST_MIPMAP_LINEAR	GetTexParameteriv texture	n x Z7		
TEXTURE_MAG_FILTER texture	GetTexParameteriv	n x Z3	LINEAR	
TEXTURE_WRAP_S texture	GetTexParameteriv	n x Z2	REPEAT	
TEXTURE_WRAP_T texture	GetTexParameteriv	n x Z2	REPEAT	
TEXTURE_WRAP_R_EXT texture	GetTexParameteriv	n x Z2	REPEAT	
TEXTURE_WRAP_Q_SGIS texture	GetTexParameteriv	n x Z2	REPEAT	

New Implementation Dependent State

None

APPENDIX G
3Dlabs OpenGL Driver State Extension Specification

Name
3DLabs_Driver_State

Overview

These driver extensions are accessed through a generic enable/disable function using the routines below (allowing for future expansion). There are currently two 3Dlabs specific driver extensions.

1) FORCE_POSITIVE_NORMALS_3DLABS

When lighting is being used and negative scaling factors are applied to the modeling matrix it can produce undesirable effects with respect to the lighting operation when the objective of the negative scale factors is that of mirroring an object about an axis. This extension is to allow reasonable visual results to be obtained when viewing a model exported by Autocad, along with its own matrix.

The operation of the extension is simple - and when the extension is enabled will cause the normalisation of normals to flip any negative normal component. This state is held on a per rendering context basis.

2) LOCK_GLOBAL_TEXTURE_PALETTE

When paletted textures are in use, it is possible, using this extension, to download and lock a single palette into the chip. This speeds up texture switches considerably especially when 8 bit paletted textures are supported (as with GLINT MX and Permedia2). The desired global palette should be downloaded and then the driver extension used to lock the palette with target LOCK_GLOBAL_TEXTURE_PALETTE and value GL_TRUE (see below). The palette is then unlocked with the value GL_FALSE. This state is held on a per rendering context basis.

Name Strings
GL_3DLabs_Driver_State (this is the string that should be exported)

Dependencies
None

Procedures and Functions
int DriverStateSet3Dlabs (int target, int value);

Where target is one of:

FORCE_POSITIVE_NORMALS_3DLABS or LOCK_GLOBAL_TEXTURE_PALETTE

and value should either be GL_FALSE (for off) or GL_TRUE (for on).

The DriverGetState3Dlabs function will return the current value of the parameter.

GL_TRUE will be returned if the call succeeded. GL_FALSE will be returned if the call failed because the 'target' value was not recognised.

```
int     DriverStateGet3Dlabs(int target);
```

The returned value will be the current value of the 'target' piece of state.

Mechanism for using the extensions.

The extension's presence can be detected by searching the supported extensions string for named string. If the string is present, then the user can locate the two extension functions by calling the wglGetProcAddress() routine to get a pointer to the function.

```
// Declarations of function pointers..
int   (APIENTRY *MyDriverSetState)   (int, int);
int   (APIENTRY *MyDriverGetState)  (int);
```

// Get the addresses of the functions

MyDriverSetState = (void *) wglGetProcAddress("glDriverSetState3Dlabs");

MyDriverGetState = (void *) wglGetProcAddress("glDriverGetState3Dlabs");

```
// Switch into the mode to force the normals to be positive - assuming that the pointers
// are not NULL.
MyDriverSetState (FORCE_POSITIVE_NORMALS_3DLABS,  GL_TRUE);

// Switch out of the state.
MyDriverSetState (FORCE_POSITIVE_NORMALS_3DLABS,  GL_FALSE);
```

Values

```
#define  FORCE_POSITIVE_NORMALS_3DLABS        0x01
#define  RESERVED                             0x02
#define  LOCK_GLOBAL_TEXTURE_PALETTE          0x03
```