*Appendix C*

# OpenGL and Window Systems

OpenGL is available on many different platforms and works with many different window systems. OpenGL is designed to complement window systems, not duplicate their functionality. Therefore, OpenGL performs geometric and image rendering in two and three dimensions, but it does not manage windows or handle input events.

However, the basic definitions of most window systems don't support a library as sophisticated as OpenGL, with its complex and diverse pixel formats, including depth, stencil, and accumulation buffers, as well as double-buffering. For most window systems, some routines are added to extend the window system to support OpenGL.

This appendix introduces the extensions defined for several window and operating systems: the X Window System, the Apple Mac OS, OS/2 Warp from IBM, and Microsoft Windows NT and Windows 95. You need to have some knowledge of the window systems to fully understand this appendix.

This appendix has the following major sections:

- "GLX: OpenGL Extension for the X Window System"
- "AGL: OpenGL Extension to the Apple Macintosh"
- "PGL: OpenGL Extension for IBM OS/2 Warp"
- "WGL: OpenGL Extension for Microsoft Windows NT and Windows 95"

## GLX: OpenGL Extension for the X Window System

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense. GLX is an extension to the X protocol (and its associated API) for communicating OpenGL commands to an extended X server. Connection and authentication are accomplished with the normal X mechanisms.

As with other X extensions, there is a defined network protocol for OpenGL's rendering commands encapsulated within the X byte stream, so client-server OpenGL rendering is supported. Since performance is critical in three-dimensional rendering, the OpenGL extension to X allows OpenGL to bypass the X server's involvement in data encoding, copying, and interpretation and instead render directly to the graphics pipeline.

The X Visual is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type XVisualInfo keeps track of pixel information, including pixel type (RGBA or color index), single or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers. The standard X Visuals (for example, PseudoColor, TrueColor) do not describe the pixel format details, so each implementation must extend the number of X Visuals supported.

The GLX routines are discussed in more detail in the *OpenGL Reference Manual*. Integrating OpenGL applications with the X Window System and the Motif widget set is discussed in great detail in *OpenGL Programming for the X Window System* by Mark Kilgard (Reading, MA: Addison-Wesley Developers Press, 1996), which includes full source code examples. If you absolutely want to learn about the internals of GLX, you may want to read the GLX specification, which can be found at

```
ftp://sgigate.sgi.com/pub/opengl/doc/
```

### Initialization

Use glXQueryExtension() and glXQueryVersion() to determine whether the GLX extension is defined for an X server and, if so, which version is present. glXQueryExtensionsString() returns extension information about the client-server connection. glXGetClientString() returns information about the client library, including extensions and version number. glXQueryServerString() returns similar information about the server.

glXChooseVisual() returns a pointer to an XVisualInfo structure describing the visual that meets the client's specified attributes. You can query a visual about its support of a particular OpenGL attribute with glXGetConfig().

## Controlling Rendering

Several GLX routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render off-screen if you want. Routines are also provided for such tasks as synchronizing execution between the X and OpenGL streams, swapping front and back buffers, and using an X font.

### Managing an OpenGL Rendering Context

An OpenGL rendering context is created with glXCreateContext(). One of the arguments to this routine allows you to request a direct rendering context that bypasses the X server as described previously. (Note that to do direct rendering, the X server connection must be local, and the OpenGL implementation needs to support direct rendering.) glXCreateContext() also allows display-list and texture-object indices and definitions to be shared by multiple rendering contexts. You can determine whether a GLX context is direct with glXIsDirect().

To make a rendering context current, use glXMakeCurrent(); glXGetCurrentContext() returns the current context. You can also obtain the current drawable with glXGetCurrentDrawable() and the current X Display with glXGetCurrentDisplay(). Remember that only one context can be current for any thread at any one time. If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with glXCopyContext(). When you're finished with a particular context, destroy it with glXDestroyContext().

### Off-Screen Rendering

To render off-screen, first create an X Pixmap and then pass this as an argument to glXCreateGLXPixmap(). Once rendering is completed,
you can destroy the association between the X and GLX Pixmaps with glXDestroyGLXPixmap(). (Off-screen rendering isn't guaranteed to be supported for direct renderers.)

### Synchronizing Execution

To prevent X requests from executing until any outstanding OpenGL rendering is completed, call glXWaitGL(). Then, any previously issued OpenGL commands are guaranteed to be executed before any X rendering calls made after glXWaitGL(). Although the same result can be achieved with glFinish(), glXWaitGL() doesn't require a round trip to the server and thus is more efficient in cases where the client and server are on separate machines.

To prevent an OpenGL command sequence from executing until any outstanding X requests are completed, use glXWaitX(). This routine guarantees that previously issued X rendering calls are executed before any OpenGL calls made after glXWaitX().

### Swapping Buffers

For drawables that are double-buffered, the front and back buffers can be exchanged by calling glXSwapBuffers(). An implicit glFlush() is done as part of this routine.

### Using an X Font

A shortcut for using X fonts in OpenGL is provided with the command glXUseXFont(). This routine builds display lists, each of which calls glBitmap(), for each requested character from the specified font and font size.

## GLX Prototypes

### Initialization

Determine whether the GLX extension is defined on the X server:

Bool glXQueryExtension ( Display *_dpy_, int *_errorBase_, int *_eventBase_ );

Query version and extension information for client and server:

Bool glXQueryVersion ( Display *_dpy_, int *_major_, int *_minor_ );

const char* glXGetClientString ( Display *_dpy_, int _name_ );

const char* glXQueryServerString ( Display *_dpy_, int _screen_, int _name_ );

const char* glXQueryExtensionsString ( Display *_dpy_, int _screen_ );

Obtain the desired visual:

XVisualInfo* glXChooseVisual ( Display *_dpy_, int _screen_,
int *_attribList_ );

int glXGetConfig ( Display *_dpy_, XVisualInfo *_visual_, int _attrib_,
int *_value_ );

### Controlling Rendering

Manage or query an OpenGL rendering context:

GLXContext glXCreateContext ( Display **dpy*, XVisualInfo **visual*, GLXContext *shareList*, Bool *direct* );

void glXDestroyContext ( Display **dpy*, GLXContext *context* );

void glXCopyContext ( Display **dpy*, GLXContext *source*, GLXContext *dest*, unsigned long *mask* );

Bool glXIsDirect ( Display **dpy*, GLXContext *context* );

Bool glXMakeCurrent ( Display **dpy*, GLXDrawable *draw*, GLXContext *context* );

GLXContext glXGetCurrentContext (void);

Display* glXGetCurrentDisplay (void);

GLXDrawable glXGetCurrentDrawable (void);

Perform off-screen rendering:

GLXPixmap glXCreateGLXPixmap ( Display **dpy*, XVisualInfo **visual*, Pixmap *pixmap* );

void glXDestroyGLXPixmap ( Display **dpy*, GLXPixmap *pix* );

Synchronize execution:

void glXWaitGL (void);

void glXWaitX (void);

Exchange front and back buffers:

void glXSwapBuffers ( Display **dpy*, GLXDrawable *drawable* );

Use an X font:

void glXUseXFont ( Font *font*, int *first*, int *count*, int *listBase* );

## AGL: OpenGL Extension to the Apple Macintosh

This section covers the routines defined as the OpenGL extension to the Apple Macintosh (AGL), as defined by Template Graphics Software. An understanding of the way the Macintosh handles graphics rendering (QuickDraw) is required. The *Macintosh Toolbox Essentials* and *Imaging With QuickDraw* manuals from the *Inside Macintosh* series are also useful to have at hand.

For more information (including how to obtain the OpenGL software library for the Power Macintosh), you may want to check out the web site for OpenGL information at Template Graphics Software:

```
http://www.sd.tgs.com/Products/opengl.htm
```

For the Macintosh, OpenGL rendering is made available as a library that is either compiled in or resident as an extension for an application that wishes to make use of it. OpenGL is implemented in software for systems that do not possess hardware acceleration. Where acceleration is available (through the QuickDraw 3D Accelerator), those capabilities that match the OpenGL pipeline are used with the remaining functionality being provided through software rendering.

The data type AGLPixelFmtID (the AGL equivalent to XVisualInfo) maintains pixel information, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

In contrast to other OpenGL implementations on other systems (such as the X Window System), the client/server model is not used. However, you may still need to call glFlush() since some hardware accelerators buffer the OpenGL pipeline and require a flush to empty it.

## Initialization

Use aglQueryVersion() to determine what version of OpenGL for the Macintosh is available.

The capabilities of underlying graphics devices and your requirements for rendering buffers are resolved using aglChoosePixelFmt(). Use aglListPixelFmts() to find the particular formats supported by a graphics device. Given a pixel format, you can determine which attributes are available by using aglGetConfig().

## Rendering and Contexts

Several AGL routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render into either a window or an off-screen graphics world. Routines are also provided that allow you to swap front and back rendering buffers, adjust buffers in response to a move, resize or graphics device change event, and use Macintosh fonts. For software rendering (and in some cases, hardware-accelerated rendering) the rendering buffers are created in your application memory space. For the application to work properly you must provide sufficient memory for these buffers in your application's SIZE resource.

### Managing an OpenGL Rendering Context

An OpenGL rendering context is created (at least one context per window being rendered into) with aglCreateContext(). This takes the pixel format you selected as a parameter and uses it to initialize the context.

Use aglMakeCurrent() to make a rendering context current. Only one context can be current for a thread of control at any time. This indicates which drawable is to be rendered into and which context to use with it. It's possible for more than one context to be used (not simultaneously) with a particular drawable. Two routines allow you to determine which is the current rendering context and drawable being rendered into: aglGetCurrentContext() and aglGetCurrentDrawable().

If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with aglCopyContext(). When a particular context is finished with, it should be destroyed by calling aglDestroyContext().

### On-screen Rendering

With the OpenGL extensions for the Apple Macintosh you can choose whether window clipping is performed when writing to the screen and whether the cursor is hidden during screen writing operations. This is important since these two items may affect how fast rendering can be performed. Call aglSetOptions() to select these options.

### Off-screen Rendering

To render off-screen, first create an off-screen graphics world in the usual way, and pass the handle into aglCreateAGLPixmap(). This routine returns a drawable that can be used with aglMakeCurrent(). Once rendering is completed, you can destroy the association with aglDestroyAGLPixmap().

### Swapping Buffers

For drawables that are double-buffered (as per the pixel format of the current rendering context), call aglSwapBuffers() to exchange the front and back buffers. An implicit glFlush() is performed as part of this routine.

### Updating the Rendering Buffers

The Apple Macintosh toolbox requires you to perform your own event handling and does not provide a way for libraries to automatically hook in to the event stream. So that the drawables maintained by OpenGL can adjust to changes in drawable size, position and pixel depth, aglUpdateCurrent() is provided.

This routine must be called by your event processing code whenever one of these events occurs in the current drawable. Ideally the scene should be rerendered after a update call to take into account the changes made to the rendering buffers.

### Using an Apple Macintosh Font

A shortcut for using Macintosh fonts is provided with aglUseFont(). This routine builds display lists, each of which calls glBitmap(), for each requested character from the specified font and font size.

## Error Handling

An error-handling mechanism is provided for the Apple Macintosh OpenGL extension. When an error occurs you can call aglGetError() to get a more precise description of what caused the error.

## AGL Prototypes

### Initialization

Determine AGL version:

GLboolean aglQueryVersion ( int **major*, int **minor* );

Pixel format selection, availability, and capability:

AGLPixelFmtID aglChoosePixelFmt ( GDHandle **dev*, int *ndev*, int **attribs* );

int aglListPixelFmts ( GDHandle *dev*, AGLPixelFmtID ***fmts* );

GLboolean aglGetConfig ( AGLPixelFmtID *pix*, int *attrib*, int **value* );

### Controlling Rendering

Manage an OpenGL rendering context:

AGLContext aglCreateContext ( AGLPixelFmtID *pix*, AGLContext *shareList* );

GLboolean aglDestroyContext ( AGLContext *context* );

GLboolean aglCopyContext ( AGLContext *source*, AGLContext *dest*, GLuint *mask* );

GLboolean aglMakeCurrent ( AGLDrawable *drawable*,
AGLContext *context* );

GLboolean aglSetOptions ( int *opts* );

AGLContext aglGetCurrentContext (void);

AGLDrawable aglGetCurrentDrawable (void);

Perform off-screen rendering:

AGLPixmap aglCreateAGLPixmap ( AGLPixelFmtID *pix*,
GWorldPtr *pixmap* );

GLboolean aglDestroyAGLPixmap ( AGLPixmap *pix* );

Exchange front and back buffers:

GLboolean aglSwapBuffers ( AGLDrawable *drawable* );

Update the current rendering buffers:

GLboolean aglUpdateCurrent (void);

Use a Macintosh font:

GLboolean aglUseFont ( int *familyID*, int *size*, int *first*, int *count*,
int *listBase* );

Find the cause of an error:

GLenum aglGetError (void);

## PGL: OpenGL Extension for IBM OS/2 Warp

OpenGL rendering for IBM OS/2 Warp is accomplished by using PGL routines added to integrate OpenGL into the standard IBM Presentation Manager. OpenGL with PGL supports both a direct OpenGL context (which is often faster) and an indirect context (which allows some integration of Gpi and OpenGL rendering).

The data type VISUALCONFIG (the PGL equivalent to XVisualInfo) maintains the visual configuration, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information (including how to obtain the OpenGL software library for IBM OS/2 Warp, Version 3.0), you may want to start at

```
http://www.austin.ibm.com/software/OpenGL/
```

Packaged along with the software is the document, *OpenGL On OS/2 Warp*, which provides more detailed information. OpenGL support is included with the base operating system with OS/2 Warp Version 4.

## Initialization

Use pglQueryCapability() and pglQueryVersion() to determine whether the OpenGL is supported on this machine and, if so, how it is supported and which version is present. pglChooseConfig() returns a pointer to an VISUALCONFIG structure describing the visual configuration that best meets the client's specified attributes. A list of the particular visual configurations supported by a graphics device can be found using pglQueryConfigs().

## Controlling Rendering

Several PGL routines are provided for creating and managing an OpenGL rendering context, capturing the contents of a bitmap, synchronizing execution between the Presentation Manager and OpenGL streams, swapping front and back buffers, using a color palette, and using an OS/2 logical font.

### Managing an OpenGL Rendering Context

An OpenGL rendering context is created with pglCreateContext(). One of the arguments to this routine allows you to request a direct rendering context that bypasses the Gpi and render to a PM window, which is generally faster. You can determine whether a OpenGL context is direct with pglIsIndirect().

To make a rendering context current, use pglMakeCurrent(); pglGetCurrentContext() returns the current context. You can also obtain the current window with pglGetCurrentWindow(). You can copy some OpenGL state variables from one context to another with pglCopyContext(). When you're finished with a particular context, destroy it with pglDestroyContext().

### Access the Bitmap of the Front Buffer

To lock access to the bitmap representation of the contents of the front buffer, use pglGrabFrontBitmap(). An implicit glFlush() is performed, and you can read the bitmap, but its contents are effectively read-only. Immediately after access is completed, you should call pglReleaseFrontBitmap() to restore write access to the front buffer.

### Synchronizing Execution

To prevent Gpi rendering requests from executing until any outstanding OpenGL rendering is completed, call pglWaitGL(). Then, any previously issued OpenGL commands are guaranteed to be executed before any Gpi rendering calls made after pglWaitGL().

To prevent an OpenGL command sequence from executing until any outstanding Gpi requests are completed, use pglWaitPM(). This routine guarantees that previously issued Gpi rendering calls are executed before any OpenGL calls made after pglWaitPM().

**Note:** OpenGL and Gpi rendering can be integrated in the same window only if the OpenGL context is an indirect context.

### Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling pglSwapBuffers(). An implicit glFlush() is done as part of this routine.

### Using a Color Index Palette

When you are running in 8-bit (256 color) mode, you have to worry about color palette management. For windows with a color index Visual Configuration, call pglSelectColorIndexPalette() to tell OpenGL what color-index palette you want to use with your context. A color palette must be selected before the context is initially bound to a window. In RGBA mode, OpenGL sets up a palette automatically.

### Using an OS/2 Logical Font

A shortcut for using OS/2 logical fonts in OpenGL is provided with the command pglUseFont(). This routine builds display lists, each of which calls glBitmap(), for each requested character from the specified font and font size.

## PGL Prototypes

### Initialization

Determine whether OpenGL is supported and, if so, its version number:

long pglQueryCapability (HAB *hab*);

void pglQueryVersion (HAB *hab*, int **major*, int **minor*);

Visual configuration selection, availability and capability:

PVISUALCONFIG pglChooseConfig (HAB *hab*, int **attribList*);

PVISUALCONFIG * pglQueryConfigs (HAB *hab*);

### Controlling Rendering

Manage or query an OpenGL rendering context:

HGC pglCreateContext (HAB *hab*, PVISUALCONFIG *pVisualConfig*, HGC *shareList*, Bool isD*irect*);

Bool pglDestroyContext (HAB *hab*, HGC *hgc*);

Bool pglCopyContext (HAB *hab*, HGC *source*, HGC *dest*, GLuint *mask*);

Bool pglMakeCurrent (HAB *hab*, HGC *hgc*, HWND *hwnd*);

long pglIsIndirect (HAB *hab*, HGC *hgc*);

HGC pglGetCurrentContext (HAB *hab*);

HWND pglGetCurrentWindow (HAB *hab*);

Access and release the bitmap of the front buffer:

Bool pglGrabFrontBitmap (HAB *hab*, HPS **hps*, HBITMAP **phbitmap*);

Bool pglReleaseFrontBitmap (HAB *hab*);

Synchronize execution:

HPS pglWaitGL (HAB *hab*);

void pglWaitPM (HAB *hab*);

Exchange front and back buffers:

void pglSwapBuffers (HAB *hab*, HWND *hwnd*);

Finding a color-index palette:

void pglSelectColorIndexPalette (HAB *hab*, HPAL, *hpal*, HGC *hgc*);

Use an OS/2 logical font:

Bool pglUseFont (HAB *hab*, HPS *hps*, FATTRS **fontAttribs*, long *logicalId*, int *first*, int *count*, int *listBase*);

# WGL: OpenGL Extension for Microsoft Windows NT and Windows 95

OpenGL rendering is supported on systems that run Microsoft Windows NT and Windows 95. The functions and routines of the Win32 library are necessary to initialize the pixel format and control rendering for OpenGL. Some routines, which are prefixed by wgl, extend Win32 so that OpenGL can be fully supported.

For Win32/WGL, the PIXELFORMATDESCRIPTOR is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type PIXELFORMATDESCRIPTOR keeps track of pixel information, including pixel type (RGBA or color index), single- or double- buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information about WGL, you may want to start with technical articles available through the Microsoft Developer Network at

`http://www.microsoft.com/msdn/`

## Initialization

Use GetVersion() or the newer GetVersionEx() to determine version information. ChoosePixelFormat() tries to find a PIXELFORMATDESCRIPTOR with specified attributes. If a good match for the requested pixel format is found, then SetPixelFormat() should be called to actually use the pixel format. You should select a pixel format in the device context before calling wglCreateContext().

If you want to find out details about a given pixel format, use DescribePixelFormat() or, for overlays or underlays, wglDescribeLayerPlane().

## Controlling Rendering

Several WGL routines are provided for creating and managing an OpenGL rendering context, rendering to a bitmap, swapping front and back buffers, finding a color palette, and using either bitmap or outline fonts.

### Managing an OpenGL Rendering Context

wglCreateContext() creates an OpenGL rendering context for drawing on the device in the selected pixel format of the device context. (To create an OpenGL rendering context for overlay or underlay windows, use wglCreateLayerContext() instead.) To make a rendering context current, use wglMakeCurrent(); wglGetCurrentContext() returns the

current context. You can also obtain the current device context with wglGetCurrentDC(). You can copy some OpenGL state variables from one context to another with wglCopyContext() or make two contexts share the same display lists and texture objects with wglShareLists(). When you're finished with a particular context, destroy it with wglDestroyContext().

### OpenGL Rendering to a Bitmap

Win32 has a few routines to allocate (and deallocate) bitmaps, to which you can render OpenGL directly. CreateDIBitmap() creates a device-dependent bitmap (DDB) from a device-independent bitmap (DIB). CreateDIBSection() creates a device-independent bitmap (DIB) that applications can write to directly. When finished with your bitmap, you can use DeleteObject() to free it up.

### Synchronizing Execution

If you want to combine GDI and OpenGL rendering, be aware there are no equivalents to functions like glXWaitGL(), glXWaitX(), or pglWaitGL() in Win32. Although glXWaitGL() has no equivalent in Win32, you can achieve the same effect by calling glFinish(), which waits until all pending OpenGL commands are executed, or by calling GdiFlush(), which waits until all GDI drawing has completed.

### Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling SwapBuffers() or wglSwapLayerBuffers(); the latter for overlays and underlays.

### Finding a Color Palette

To access the color palette for the standard (non-layer) bitplanes, use the standard GDI functions to set the palette entries. For overlay or underlay layers, use wglRealizeLayerPalette(), which maps palette entries from a given color-index layer plane into the physical palette or initializes the palette of an RGBA layer plane. wglGetLayerPaletteEntries() is used to query the entries in palettes of layer planes.

### Using a Bitmap or Outline Font

WGL has two routines, wglUseFontBitmaps() and wglUseFontOutlines(), for converting system fonts to use with OpenGL. Both routines build a display list for each requested character from the specified font and font size.

## WGL Prototypes

### Initialization

Determine version information:

BOOL GetVersion ( LPOSVERSIONINFO *lpVersionInformation* );

BOOL GetVersionEx ( LPOSVERSIONINFO *lpVersionInformation* );

Pixel format availability, selection, and capability:

int ChoosePixelFormat ( HDC *hdc*,
CONST PIXELFORMATDESCRIPTOR * *ppfd* );

BOOL SetPixelFormat ( HDC *hdc*, int *iPixelFormat*,
CONST PIXELFORMATDESCRIPTOR * *ppfd* );

int DescribePixelFormat ( HDC *hdc*, int *iPixelFormat*, UINT *nBytes*,
LPPIXELFORMATDESCRIPTOR *ppfd* );

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

### Controlling Rendering

Manage or query an OpenGL rendering context:

HGLRC wglCreateContext ( HDC *hdc* );

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

BOOL wglShareLists ( HGLRC *hglrc1*, HGLRC *hglrc2* );

BOOL wglDeleteContext ( HGLRC *hglrc* );

BOOL wglCopyContext ( HGLRC *hglrcSource*, HGLRC *hlglrcDest*,
UINT *mask* );

BOOL wglMakeCurrent ( HDC *hdc*, HGLRC *hglrc* );

HGLRC wglGetCurrentContext (VOID) ;

HDC wglGetCurrentDC (VOID);

Access and release the bitmap of the front buffer:

HBITMAP CreateDIBitmap ( HDC *hdc*,
CONST BITMAPINFOHEADER **lpbmih*, DWORD *fdwInit*,
CONST VOID **lpbInit*, CONST BITMAPINFO **lpbmi*, UINT *fuUsage* );

HBITMAP CreateDIBSection ( HDC *hdc*, CONST BITMAPINFO **pbmi*,
UINT *iUsage*, VOID **ppvBits*, HANDLE *hSection*, DWORD *dwOffset* );

BOOL DeleteObject ( HGDIOBJ *hObject* );

Exchange front and back buffers:

BOOL SwapBuffers ( HDC *hdc* );

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

Finding a color palette for overlay or underlay layers:

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

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

Use a bitmap or an outline font:

BOOL wglUseFontBitmaps ( HDC *hdc*, DWORD *first*, DWORD *count*,
DWORD *listBase* );

BOOL wglUseFontOutlines ( HDC *hdc*, DWORD *first*, DWORD *count*,
DWORD *listBase*, FLOAT *deviation*, FLOAT *extrusion*, int *format*,
LPGLYPHMETRICSFLOAT *lpgmf* );