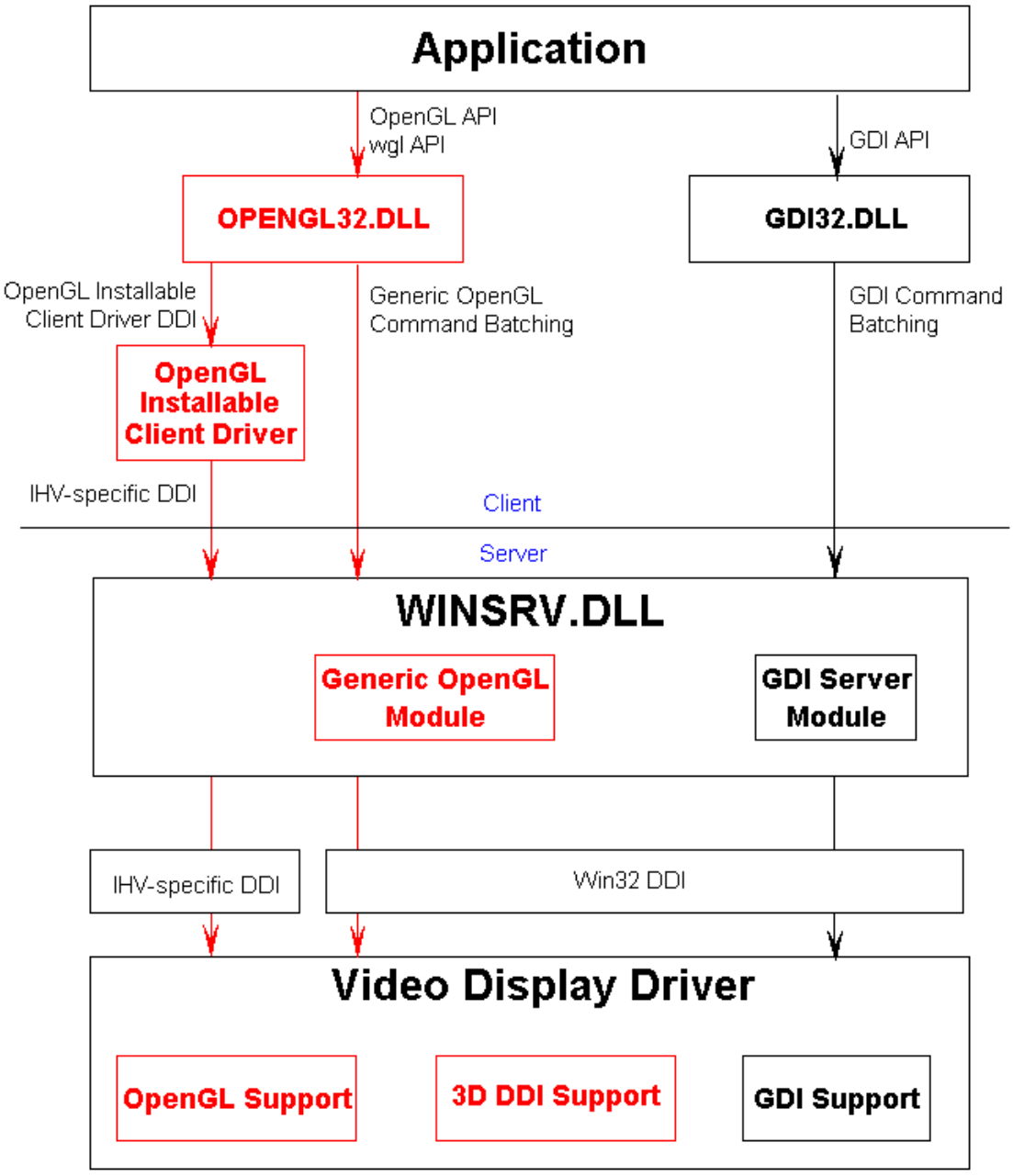
Generic vs. Device Format, AKA OpenGL/NT Architecture

It is always helpful to understand the architecture of a new feature. From an application developer's perspective, a good understanding of the architecture eases the application development process. Design and implementation decisions can be made with intelligence instead of confusion. If you buy that, take a look at Figure 1. It is the infamous architecture diagram with an OpenGL/NT flavor this time.



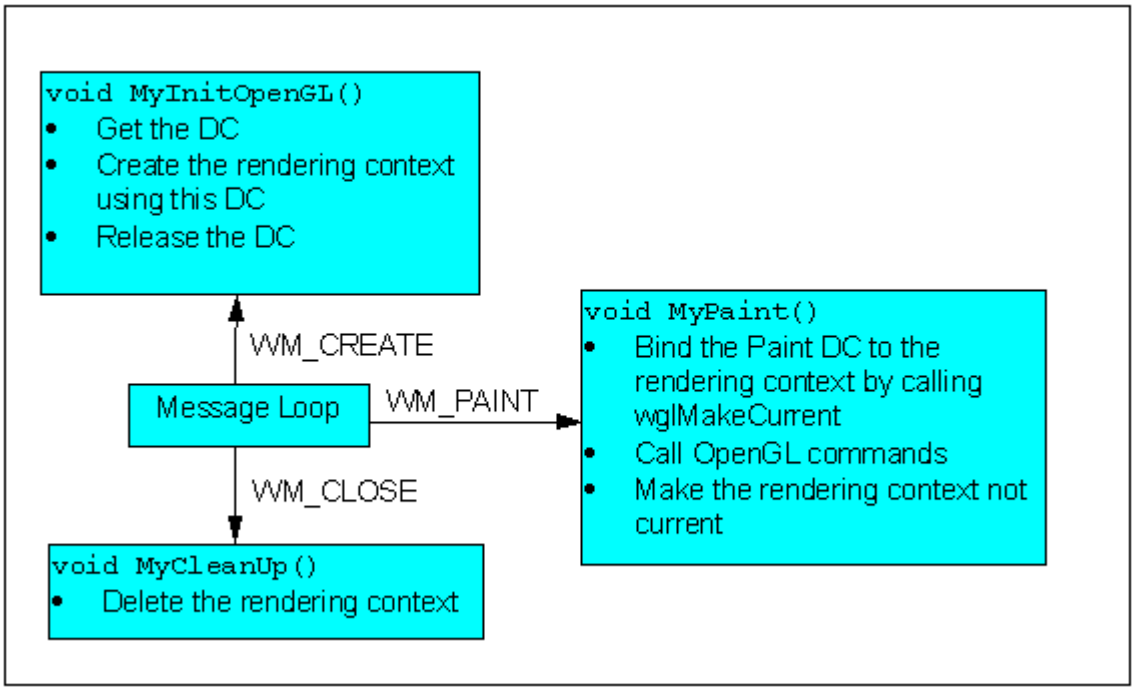
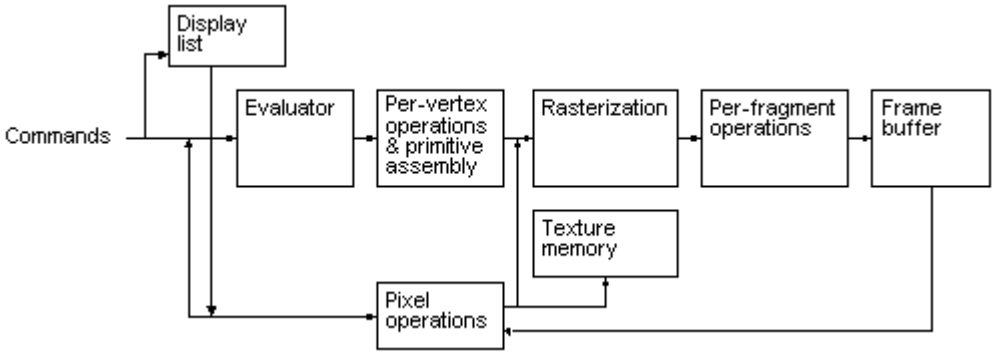


Figure 1. OpenGL/NT architecture

Basic OpenGL Operation

The following diagram illustrates how OpenGL processes data. As shown, commands enter from the left and proceed through a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during various processing stages.



The processing stages in basic OpenGL operation are as follows:

- **Display list** Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.
- **Evaluator** The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.
- **Per-vertex operations and primitive assembly** OpenGL processes geometric primitives—points, line segments, and polygons—all of which are described by vertices.

Vertices are transformed and lit, and primitives are clipped to the viewport in preparation for rasterization.

- **Rasterization** The rasterization stage produces a series of frame-buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per-fragment operations.
- **Per-fragment operations** These are the final operations performed on the data before it's stored as pixels in the framebuffer.

  Per-fragment operations include conditional updates to the framebuffer based on incoming and previously stored z values (for z buffering) and blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values.

Data can be input in the form of pixels rather than vertices. Data in the form of pixels, such as might describe an image for use in texture mapping, skips the first stage of processing described above and instead is processed as pixels, in the pixel operations stage. Following pixel operations, the pixel data is either:

- Stored as texture memory, for use in the rasterization stage.
- Rasterized, with the resulting fragments merged into the framebuffer just as if they were generated from geometric data.

Primitives and Commands

OpenGL draws *primitives*—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of one another. That is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the framebuffer). To specify primitives, set modes, and perform other OpenGL operations, you issue commands in the form of function calls.

Primitives are defined by a group of one or more *vertices.* A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normals, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exceptions to this rule are cases in which the group of vertices must be clipped so that a particular primitive fits within a specified region. In this case, vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that is consistent with complete execution of all previously issued OpenGL commands.

OpenGL Graphic Control

OpenGL provides you with fairly direct control over the fundamental operations of two- and three-dimensional graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel-update operators. However, it doesn't provide you with a means for describing or modeling complex geometric objects. Thus, the OpenGL commands you issue specify how a certain result should be produced (what procedure should be followed) rather than what exactly that result should look like. That is, OpenGL is fundamentally procedural rather than descriptive. To fully understand how to use OpenGL, it helps to know the order in which it carries out its operations.

Execution Model

The model for interpretation of OpenGL commands is client/server. Application code (the client) issues commands, which are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client. In this sense, OpenGL is network-transparent. A server can maintain several OpenGL contexts, each of which is an encapsulated OpenGL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol (such as that of the X Window System) or by using an independent protocol. No OpenGL commands are provided for obtaining user input.

The window system that allocates framebuffer resources ultimately controls the effects of OpenGL commands on the framebuffer. The window system:

- Determines which portions of the framebuffer OpenGL may access at any given time.
- Communicates to OpenGL how those portions are structured.

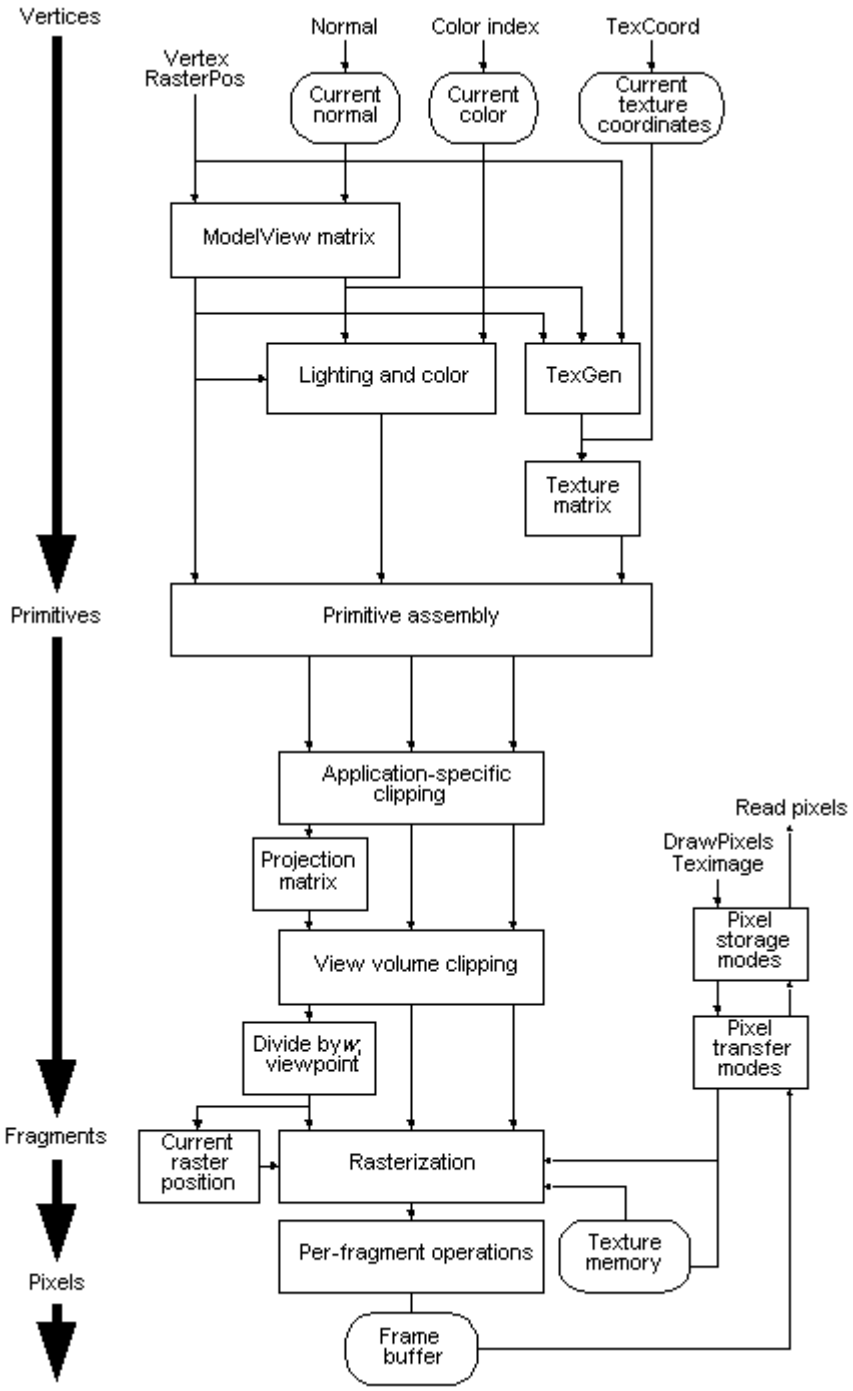
Therefore, there are no OpenGL commands to configure the framebuffer or initialize OpenGL. Frame buffer configuration is done outside of OpenGL in conjunction with the window system; OpenGL initialization takes place when the window system allocates a window for OpenGL rendering.

OpenGL Processing Pipeline

Many OpenGL functions are used specifically for drawing objects such as points, lines, polygons, and bitmaps. Some functions control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Other functions are specifically concerned with framebuffer manipulation. The topics in this section describe how all of the OpenGL functions work together to create the OpenGL processing pipeline. This section also takes a closer look at the stages in which data is actually processed, and ties these stages to OpenGL functions.

The following diagram details the OpenGL processing pipeline. For most of the pipeline, you can see three vertical arrows between the major stages. These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates. Also note that vertices are assembled into primitives, then into fragments, and finally into pixels in

the framebuffer. This progression is discussed in more detail in Vertices, Primitives, Fragments, and Pixels.



OpenGL Function Names

Many OpenGL functions are variations of each other, differing mostly in the data types of their arguments. Some functions differ in the number of related arguments and whether those arguments can be specified as a vector or must be specified separately in a list. For example, if you

use the **glVertex2f** function, you need to supply x- and y-coordinates as 32-bit floating-point numbers; with **glVertex3sv**, you must supply an array of three short (16-bit) integer values for x, y, and z. Only the base name of the function is used in the topics that follow. An asterisk indicates that there may be more to the actual function name than is shown. For example, **glVertex*** stands for all the variations of the function you use to specify vertices: **glVertex2d**, **glVertex2f**, **glVertex2i**, and so on.

The effect of an OpenGL function can vary depending on whether certain modes are enabled. For example, you need to enable lighting if the lighting-related functions are to produce a properly lit object. To enable a particular mode, use the **glEnable** function and supply the appropriate constant to identify the mode (for example, GL_LIGHTING). To disable a mode, use **glDisable**. See **glEnable** for a complete list of the modes that can be enabled.

Vertices

The topics in this section discuss the OpenGL functions that perform per-vertex operations to the processing stages shown in OpenGL Processing Pipeline on the preceding page.

Primitives

During the next stage of processing, primitives are converted to pixel fragments in the following steps:

- Primitives are clipped appropriately.
- Necessary corresponding adjustments are made to the color and texture data, and the relevant coordinates are transformed to window coordinates.
- Rasterization converts the clipped primitives to pixel fragments.

Fragments

A fragment produced by rasterization modifies the corresponding pixel in the framebuffer only if it passes the following tests:

- Pixel ownership test
- Scissor test
- Alpha test
- Stencil test

- Depth-buffer test

If it passes, the fragment's data can replace the existing framebuffer values, or you can combine it with existing data in the framebuffer, depending on the state of certain modes. You can combine the fragment with data in the framebuffer by:

- Blending
- Dithering
- Logical operations

Pixels

Fragments are converted to pixels in the framebuffer. The framebuffer is organized into a set of logical buffers—the color, depth, stencil, and accumulation buffers. The color buffer itself consists of a front left, front right, back left, back right, and some number of auxiliary buffers. You can issue functions to control these buffers, and directly read or copy pixels from them. (Note that the particular OpenGL context you're using may not provide all these buffers.)

OpenGL Performance Tips

These programming practices optimize your application's performance:

- Use **glColorMaterial** when only a single material property is being varied rapidly (at each vertex, for example). Use **glMaterial** for infrequent changes, or when more than a single material property is being varied rapidly.
- Use **glLoadIdentity** to initialize a matrix, rather than loading your own copy of the identity matrix.
- Use specific matrix calls such as **glRotate**, **glTranslate**, and **glScale**, rather than composing your own rotation, translation, and scale matrices and calling **glMultMatrix**.
- Use **glPushAttrib** and **glPopAttrib** to save and restore state values. Use query functions only when your application requires the state values for its own computations.
- Use display lists to encapsulate potentially expensive state changes. For example, place all the **glTexImage** calls required to completely specify a texture (and perhaps the associated **glTexParameter**, **glPixelStore**, and **glPixelTransfer** calls as well) into a single display list. Call this display list to select the texture.
- Use display lists to encapsulate the rendering calls of rigid objects that will be drawn repeatedly.
- To minimize network bandwidth in client/server environments, use evaluators even for simple surface tessellations.

- If possible, to avoid the overhead of GL_NORMALIZE, provide unit-length normals. Because **glScale** almost always requires enabling GL_NORMALIZE, avoid using **glScale** when doing lighting.
- If smooth shading isn't required, set **glShadeModel** to GL_FLAT.
- Use a single **glClear** call per frame, if possible. Do not use **glClear** to clear small subregions of the buffers; use it only to clear the buffer completely or nearly completely.
- To draw multiple independent triangles, use a single call rather than multiple calls to **glBegin**(GL_TRIANGLES) or a call to **glBegin**(GL_POLYGON). Similarly:

  To draw a single triangle, use GL_TRIANGLES rather than GL_POLYGON.

  Use a single call to **glBegin**(GL_QUADS) rather than calling **glBegin**(GL_POLYGON) repeatedly.

  Use a single call to **glBegin**(GL_LINES) to draw multiple independent line segments, rather than calling **glBegin**(GL_LINES) multiple times.

- In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
- Avoid making redundant mode changes, such as setting the color to the same value between each vertex of a flat-shaded polygon.
- When drawing or copying images, disable rasterization and per-fragment operations, to optimize resources. OpenGL can apply textures to pixel images.

OpenGL Correctness Tips

Follow these guidelines to create OpenGL applications that perform as you intend:

- Assume error behavior on the part of an OpenGL implementation may change in a future release of OpenGL. OpenGL error semantics may change in future revisions.
- Use the projection matrix to collapse all geometry to a single plane. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.
- Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling **glRotate** with an incremental angle. Rather, use **glLoadIdentity** to initialize the given matrix for each frame, and then call **glRotate** with the desired complete angle for that frame.
- Expect multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. Otherwise, multiple passes might generate varying sets of fragments.

- Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.
- To optimize the operation of the depth buffer, place the near frustum plane as far from the viewpoint as possible.
- Call **glFlush** to force execution of all previous OpenGL commands. Do not count on **glGet** or **glIsEnabled** to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don't necessarily complete all pending rendering commands.
- Turn off dithering when rendering predithered images (for example, when **glCopyPixels** is called).
- Make use of the full range of the accumulation buffer. For example, if accumulating four images, scale each by one-quarter as it's accumulated.
- To obtain exact two-dimensional rasterization, carefully specify both the orthographic projection and the vertices of the primitives that are to be rasterized. Specify the orthographic projection with integer coordinates, as shown in the following example:

```
gluOrtho2D(0, width, 0, height);
```

The parameters *width* and *height* are the dimensions of the viewport. Given this projection matrix, place polygon vertices and pixel image positions at integer coordinates to rasterize predictably. For example, **glRecti**(0, 0, 1, 1) reliably fills the lower-left pixel of the viewport, and **glRasterPos2i**(0, 0) reliably positions an unzoomed image at the lower-left pixel of the viewport. However, point vertices, line vertices, and bitmap positions should be placed at half-integer locations. For example, a line drawn from (*x* (1) , 0.5) to (*x* (2) , 0.5) will be reliably rendered along the bottom row of pixels in the viewport, and a point drawn at (0.5, 0.5) will reliably fill the same pixel as **glRecti**(0, 0, 1, 1).

An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate *x* and *y* by 0.375, as shown in the following code sample. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

```
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
gluOrtho2D(0, width, 0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
glTranslatef(0.375, 0.375, 0.0);
/* render all primitives at integer positions */
```

- Avoid using negative *w* vertex coordinates and negative *q* texture coordinates. OpenGL might not clip such coordinates correctly and can make interpolation errors when shading primitives defined by such coordinates.

Limitations

The generic implementation has the following limitations:

- Printing.

  You can print an OpenGL image directly to a printer using metafiles only. For more information, see Printing an OpenGL Image.

- OpenGL and GDI graphics cannot be mixed in a double-buffered window.

  An application can directly draw both OpenGL graphics and GDI graphics into a single-buffered window, but not into a double-buffered window.

- There are no per-window hardware color palettes.

  Windows NT/2000 and Windows 95/98 have a single system hardware color palette, which applies to the whole screen. An OpenGL window cannot have its own hardware palette, but can have its own logical palette. To do so, it must become a palette-aware application. For more information, see OpenGL Color Modes and Windows Palette Management.

- There is no direct support for the Clipboard, dynamic data exchange (DDE), or OLE.

  A window with OpenGL graphics does not directly support these Windows NT/2000 and Windows 95/98 capabilities. There are workarounds, however, for using the Clipboard. For more information, see Copying an OpenGL Image to the Clipboard.

- The Inventor 2.0 C++ class library is not included.

  The Inventor class library, built on top of OpenGL, provides higher-level constructs for programming 3-D graphics. It is not included in the current version of Microsoft's implementation of OpenGL for Windows NT/2000 and Windows 95/98.

- There is no support for the following pixel format features: stereoscopic images, alpha bitplanes, and auxiliary buffers.

  There is, however, support for several ancillary buffers including: stencil buffer, accumulation buffer, back buffer (double buffering), overlay and underlay plane buffer, and depth (z-axis) buffer.