Chapter 10

Debugging OpenGL Programs

This chapter explains how to debug graphics applications with the OpenGL debugging tool ogldebug by discussing the following topics:

Ogldebug Overview

The ogldebug tool helps you find OpenGL programming errors and discover OpenGL programming style that may slow down your application. After finding an error, you can correct it and recompile your program.

Using ogldebug, you can perform the following actions at any point during program execution:

- Create a history ("trace") file of all OpenGL calls made. The history file is legal C code and may be compiled and run, letting you experiment with calls to find bugs or try test cases.
- Set a breakpoint at all occurrences of a given OpenGL call.
- Step through (or skip) OpenGL calls.
- Find out about OpenGL errors.
- Display information about OpenGL state, current display lists, and the window that belongs to the application you're debugging.

Note: If you're debugging a multi-window or multi-context application, ogldebug starts a new session (a new window appears) when the application starts a new process. The process ID is displayed in the title bar for each new window.

Note that the OpenGL debugger isn't a general-purpose debugger. Use *dbx* and related tools such as cvd (CASEVision/Workshop Debugger) to find problems in the non-OpenGL portions of a program.

How ogldebug Operates

The OpenGL debugger works like this:

- You invoke ogldebug for an application.
- A special library (*libogldebug.so*) intercepts OpenGL or GLX calls from the application to the OpenGL library (*libGL.so*).
- You can run, halt, step, and trace the application using the ogldebug graphical interface.
- After ogldebug-related processing, the actual OpenGL calls are made as they would have been had ogldebug not been present.

Setting Up ogldebug

Before you can use ogldebug, you must install the *gl_dev.sw.ogldebug* subsystem. You can use the Software Manager from your Toolchest or execute *swmgr* from the command line. Consider also installing *gl_dev.man.ogldebug* to have access to the reference pages.

Using a Configuration File

As you work with ogldebug, you will find that certain settings are best suited for certain situations. You can save and reload groups of ogldebug settings as follows:

- To save settings, choose Save Configuration from the File menu, then enter a filename using the dialog.
- To load settings, choose Load Configuration from the File menu, then select a file using the dialog.

Basic Debugging With ogldebug

This section introduces basic debugging. It first explains how to start ogldebug, then provides step-by-step instructions on creating a trace file to discover OpenGL problems.

Starting ogldebug

To debug your OpenGL program, type on the command line:

`% ogldebug` *program_name*

where *program_name* is the name of your (executable) application.

Note: It's not necessary to compile the application with any special options. The debugger works with any program compiled with *-lGL*.

The debugger brings up its main window (see Figure 10-1) and launches your application; it halts execution just before the first OpenGL call in the application. The window lets you interact with your application and displays information about it.

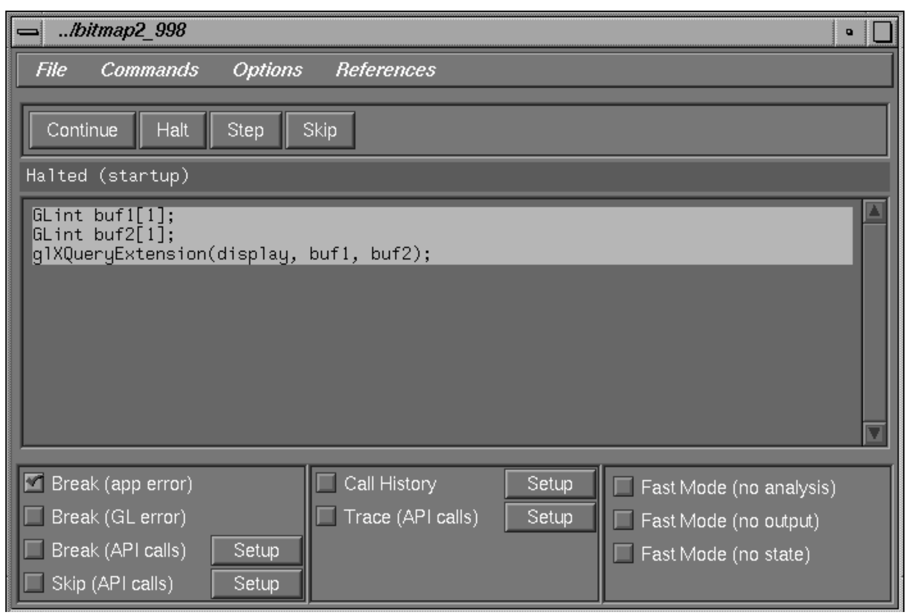


Figure 10-1 The *ogldebug* Window After Startup

Creating a Trace File to Discover OpenGL Problems

A trace file helps you find bugs in the OpenGL portion of your code without having to worry about the mechanics of window operations. Here's an example of how to collect one frame of OpenGL calls:

1. Launch ogldebug:

   `% ogldebug` *your_program_name*

2. Click the Break (API calls) checkbox to select it, then click the Setup button next to it.
3. In the Break Selection panel that appears, select a call that will result in a break after each frame: Either **glXSwapBuffers()** or **glFlush()**, depending on your program.
4. Click the Trace (API calls) checkbox to record the sequence of OpenGL calls.
5. Click the Continue button.

   Execution continues until just before the call(s) selected in the Break Selection panel—see Step 3. The ogldebug tool captures one frame worth of OpenGL calls, embeds them into the X shell window sample program, and saves them under the name *program.*trace.

6. From a shell window, in your source directory, compile the shell:

   `% mv` *program_name*`.trace trace.c`

   `% cc trace.c -lGL -lX11`

7. Launch the new *a.out* program to play back one frame of your application.

You can use *dbx* on the *trace.c* program to track down problems with your OpenGL code without having to deal with the rest of your application.

Note: In the trace file, all GLX calls are commented out with **#ifdef** to prevent them from being compiled unless the ALLOW_GLX_CALLS preprocessor constant is defined. This is because the GLX calls rely on values determined during execution of your application. On the next execution, these values are completely different; they're only provided as reference points, so you can determine where GLX calls were made.

Interacting With ogldebug

Table 10-1 provides more detailed information on working with ogldebug. It explains interacting with ogldebug using buttons or menu commands, interpreting information in display areas, and using checkboxes to determine control flow, tracing, and information collection options (fast mode).

Commands for Basic Interaction

You can perform all basic interaction using the row of buttons just below the menu bar. You can access the same commands using the Commands menu. This section describes each command, including the keyboard shortcut (also listed in the Commands menu).

Table 10-1 Command Buttons and Shortcuts

Command	Result
Continue **Ctrl+C**	Resumes program execution after execution has been stopped (such as after encountering a breakpoint or after you used the Halt or Step command). The program continues running until it reaches another breakpoint or until you explicitly halt it.
Halt **Ctrl+H**	Temporarily stops the application. All state and program information is retained so you can continue execution if you wish.
Step **Ctrl+T**	Continues executing up to the next OpenGL call, then stops before executing that call.
Skip **Ctrl+K**	Skips over the OpenGL command that's about to execute. Useful if you think that command contains an error, or is likely to cause one. The program executes until it reaches the next OpenGL call, then stops.

Understanding Display Areas

Below the row of buttons are two display areas:

- **Status display.** Immediately below the row of buttons is a one-line status display field, where ogldebug posts confirmation of commands and other status indicators.
- **OpenGL display.** Below the status display area is the OpenGL call-information display area. This area shows the next OpenGL command to be executed. Use the scroll bar to look at previously executed commands.

Using Checkboxes

The checkboxes at the bottom of the ogldebug window allow finer control over how information is collected. This section discusses the three groups of available checkboxes:

- Control-Flow Checkboxes
- Trace Checkboxes
- Fast Mode Checkboxes

Control-Flow Checkboxes

Control-flow checkboxes let you determine when a break occurs and which API calls you want to skip. Here's what happens when each of these boxes is checked:

Break (app error)	Your application halts before an OpenGL function is called with an illegal parameter. You must then skip that call to continue.
Break (GL error)	Your application halts because of an OpenGL error.
Break (API calls)	Your application halts before calling any of the OpenGL functions specified using the Setup button. By default, no functions are selected. Figure 10-2 shows a Break Selection Panel with no calls selected so far.
Skip (API calls)	Lets you use the Setup button to specify calls you want to skip. Just as the Break Selection Panel, the skip selection panel lets you create sets of calls for later use.

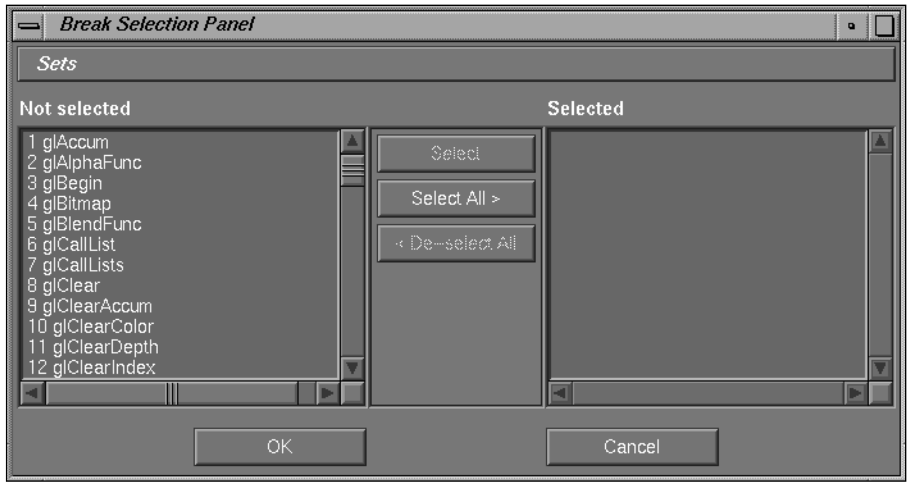


Figure 10-2 Break Selection Panel Menu for Selecting API Calls

Trace Checkboxes

The two Trace checkboxes let you record OpenGL calls in either of two formats:

Trace (API Calls) Collects a trace file in the C-like format displayed in the call information display. For example, choose to select just vertex calls or specifically deselect calls with large output if they are not relevant to the current problem.

Call History Collects the information displayed in the OpenGL display area in a file. The call history also contains some analysis information; during collection, ogldebug looks for frequent program errors or bad programming style and includes that information.

Fast Mode Checkboxes

The Fast Mode checkboxes let you determine which information ogldebug actually collects. If you don't check any of the boxes, execution of your application is quite slow. For example, when state information is being collected, ogldebug makes a **glGet*()** call for every piece of state information after every OpenGL call. Performance degradation is especially noticeable when you're debugging remotely (over a network).

Because of the potential speed penalty, collect only the information you need and turn on fast mode for information you don't need.

Fast Mode (no analysis)	Determines whether analysis information is collected.
Fast Mode (no output)	Determines whether OpenGL calls are shown in the main display area as they're invoked. Checking this box speeds up program execution but you can no longer follow the program as it's executing.
Fast Mode (no state)	Determines whether ogldebug collects OpenGL state information. If it does, you can examine the current state using the State Information command in the Options menu.

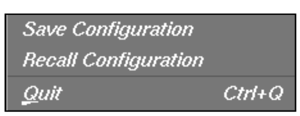
Using Menus to Interact With ogldebug

This section describes how you can interact with ogldebug using menus. You learn about:

- Using the File Menu to Interact With ogldebug
- Using the Commands Menu to Interact With Your Program
- Using the Options Menu to Access Information
- Using the References Menu for Background Information

Using the File Menu to Interact With ogldebug

The File menu lets you save and reload a configuration file, and exit ogldebug, as shown in Figure 10-3.



Saves checkbox settings in a file.
Loads checkbox settings from a file.
Quits ogldebug.

Figure 10-3 The ogldebug File Menu

Using the Commands Menu to Interact With Your Program

The Commands menu (Figure 10-4) provides the same commands as the buttons discussed in “Commands for Basic Interaction” on page 199. You can also find the keyboard accelerators for each command in the menu.

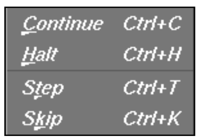


Figure 10-4 The ogldebug Commands Menu

Using the Options Menu to Access Information

The Options menu gives access to some of the information collected by ogldebug (see Figure 10-5).

Here’s a brief description of each command in the Options menu:

Call Count	Brings up a window with information about the total number of OpenGL calls as well as individual counts for each OpenGL call. You can choose between displaying all available OpenGL calls and all calls that have been made so far.
Display List Information	Brings up a window with information about the application’s display lists, if any.
Primitive Information	Provides the current count for all primitives (for example, quads, polygons, and so on).
State Information	Brings up a window that displays information on OpenGL state variables. To view state information, choose one variable by name. A window displaying the current and default value appears.
Window Information	Brings up window information for the application you are running from ogldebug.

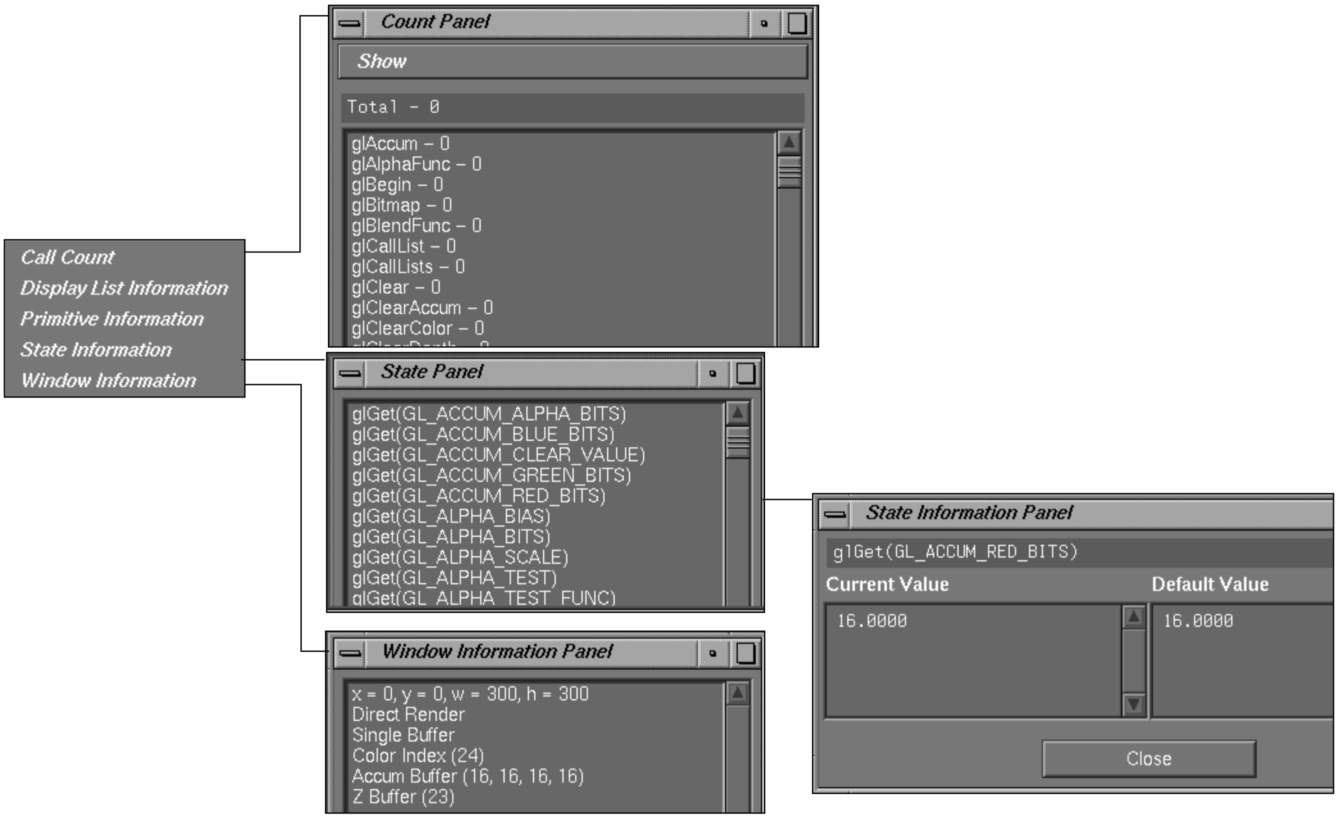


Figure 10-5 The Options Menu and Associated Panels

Using the References Menu for Background Information

The References menu provides access to useful information. You don't use it to interact with ogldebug, but it can facilitate debugging by making needed information available at the click of a mouse. To find a specific item in any list, scroll through the entries (which are arranged in alphabetical order) or use the Search field at the top of the list window.

If you choose Enumerants, a window displays a list of the symbolic names of OpenGL enumerated constants, together with the actual number (in hexadecimal and decimal) that each name represents.

Tips for Debugging Graphics Programs

The following sections are not directly related to the ogldebug debugging tool. They give advice on aspects of debugging that are known to be important for OpenGL debugging. Most points apply primarily to graphics programs and may not be obvious to people used to debugging text-based programs.

General Debugging Tips

Here are some general debugging tips that are useful when debugging an OpenGL program.

- OpenGL never signals errors but simply records them; determining whether an error occurred is up to the user. During the debugging phase, your program should call **glGetError()** to look for errors frequently (for example, once per redraw) until **glGetError()** returns GL_NO_ERROR. While this slows down performance somewhat, it helps you debug the program efficiently. You can use ogldebug to automatically call **glGetError()** after every OpenGL call.
- Use an iterative coding process: add some graphics-related code, build and test to ensure expected results, and repeat as necessary.
- Be careful with OpenGL enumerated constants that are similar in name. For example, **glBegin(GL_LINES)** works; **glBegin(GL_LINE)** does not. Using **glGetError()** can help to detect problems like this (it reports GL_INVALID_ENUM for this specific case).
- Use only per-vertex operations in a **glBegin()/glEnd()** sequence. Within a **glBegin()/glEnd()** sequence, the only graphics commands that may be used are commands for setting materials, colors, normals, edge flags, texture coordinates, surface parametric coordinates, and vertex coordinates. The use of any other graphics command is illegal. The exact list of allowable commands is given in the reference page for **glBegin()**. Even if other calls appear to work, they are not guaranteed to work in the future and may have severe performance penalties.
- If your program is too slow, see the chapters on tuning in this guide for help on increasing performance.
- Check for matching **glPushMatrix()** and **glPopMatrix()** calls.
- Check matrix mode state information. Generally, an application should stay in GL_MODELVIEW mode, but odd visual effects can occur if the matrix mode isn’t right.

Specific Problems and Troubleshooting

This section looks at some specific problems frequently encountered by OpenGL users. Note that one generally useful approach is to experiment with an ogldebug trace of the first few frames. See “Creating a Trace File to Discover OpenGL Problems” on page 198 .

Blank Window

A common problem encountered in graphics programming is a blank window. If you find your display doesn’t show what you expected, do the following:

- To make sure you’re bound to the right window, try clearing the image buffers with **glClear()**. If you can’t clear, you may be bound to the wrong window (or no window at all).
- To make sure you’re not rendering in the background color, use an unusual color to clear the window with **glClear()**.
- To make sure you’re not clipping everything inadvertently, temporarily move the near and far clipping planes to extreme distances (such as 0.001 and 1000000.0). (Note that a range like this is totally inappropriate for actual use in a program).
- Try backing the viewpoint up to see more of the space.
- Check the section “Troubleshooting Transformations” in Chapter 3 of the *OpenGL Programming Guide*.
- Make sure you’re using the correct projection matrix.
- Remember that **glOrtho()** and **glPerspective()** calls multiply onto the current projection matrix; they don’t replace it.
- In a double-buffered program, make sure you’re calling **glXSwapBuffers()**.
- Make sure you requested a depth buffer if the program is using depth buffering.
- Check the aspect ratio of the viewing frustrum. Don’t set up your program using:

```
GLfloat aspect = event.xconfigure.width/event.xconfigure.height
                 /* 0 by integer division */
```

Rotation and Translation Problems

- **Z axis direction.** Remember that by default you start out looking down the negative z axis. Unless you move the viewpoint, objects should have negative z coordinates to be visible.

- **Rotation.** Make sure you've translated back to the origin before rotating (unless you intend to rotate about some other point).
- **Transformation order.** First translating, then rotating an object yields a different result than first rotating, then translating. The order of rotation is also important, for example, R(x), R(y), R(z) is not the same as R(z), R(y), R(x).

Depth Buffering Problems

When your program uses depth testing, be sure to:

- Enable depth testing, using **glEnable()** with a GL_DEPTH_TEST argument—depth testing is off by default. Set the depth function to the desired function, using **glDepthFunc()** —the default function is GL_LESS.
- Request a visual that supports a depth buffer. Note that on some platforms a depth buffer is automatically returned for certain color configuration (for example, RGBA on Indy systems), while on other platforms a depth buffer is only returned when one is specifically requested (RealityEngine systems for example). To guarantee that your program is portable, always ask for a depth buffer explicitly.

Animation Problems

- **Double-buffering.** After drawing to the back buffer, make sure you swap buffers with **glXSwapBuffers()**.
- **Observing the image during drawing.** If you have a performance problem and want to see which part of the image takes the longest to draw, use a single-buffered visual. You can then observe the image as it's drawn. If you don't use resources to control visual selection, call **glDrawBuffer()** with a GL_FRONT argument before rendering.

Lighting Problems

- Turn off specular shading in the early debugging stages. It's harder to visualize where specular highlights should be than where diffuse highlights should be.
- For local light sources, try drawing lines from the light source to the object you are trying to light. This helps make sure the spacial and directional nature of the light is right.
- Make sure you have both GL_LIGHTING enabled and the appropriate GL_LIGHT#'s enabled.

- Try enabling GL_NORMALIZE to see whether normals are being scaled and causing lighting problems. This is particularly important if you call **glScale()**.
- Make sure normals are pointing in the right direction.
- Make sure the light is actually at the intended position. Positions are affected by the current model-view matrix. Enabling light without any **glLight(GL_POSITION)** provides a headlight if called before **gluLookAt()** and so on.

X Window System Problems

- X and OpenGL have different notions of the Y direction. X has the origin (0, 0) in the upper left corner of the window; OpenGL has it in the lower left corner. If you try to track the mouse but find that the object is moving in the "wrong" direction vertically, this is probably the cause.
- Display lists defined in one context aren't visible to other contexts unless they explicitly share contexts.
- **glXUseXFont()** creates display lists for characters. The display lists are only visible in contexts that share objects with the one they were created in.

Pixel and Texture Write Problems

- Make sure the pixel storage mode GL_UNPACK_ALIGNMENT is set to the correct value depending on the type of data. For example:

  ```
  GLubyte buf[] = {0x9D, ... 0xA7};
           /* a lot of bitmap images are passed as bytes! */
  glBitmap(w, h, x, y, 0, 0, buf);
  ```

  The default GL_UNPACK_ALIGNMENT is 4. It should be 1 in the case above.

  The same thing applies to textures.

System-Specific Problems

- Make sure you don't exceed implementation-specific resource limits such as maximum projection stack depth.
- When moving an application from a RealityEngine system to a low-end system, make sure you're not using destination alpha planes. The current low-end machines don't support them.