# OpenGL Profiler User Guide

Graphics & Imaging > OpenGL

2008-02-08

# Contents

Introduction **Introduction 7**

Organization of This Document 7
See Also 7

Chapter 1 **Getting Started 9**

Overview 9
Before You Use OpenGL Profiler on a System 10
  Setting Up the Environment Variables 10
  Setting Preferences (Optional) 10
Starting a Profiling Session 11
  Attaching to a Running Application 12
  Launching an Application from OpenGL Profiler 13
Customizing Launch Settings 14
  Setting a Custom Pixel Format 15
  Using a Driver Emulation 15
  Setting Environment Variables 17
  Setting a Working Directory 18
Viewing a Trace 18
Viewing Statistics 19
Viewing Pixel Format Context Parameters 20

Chapter 2 **Using Breakpoints 21**

Setting Breakpoints on OpenGL Functions 21
Viewing and Modifying Resources 23
  Modifying Texture Display Parameters 24
  Modifying Programs 26
  Modifying Shaders 26
  Modifying a Renderbuffer 27
Viewing Buffers 28
Creating and Attaching Scripts to a Breakpoint 30
Disabling OpenGL Execution 32

Chapter 3 **Identifying and Solving Performance Issues 33**

Making Sure You Use Functions Correctly 33
Identifying Problem Areas in Your Application 34
Checking for Optimal Data Types and Formats 34
Managing Trace Data 35
Checking for Application Errors, Thread Conflicts, and Software Fallbacks 36

Evaluating The Effect Of The Multithreaded OpenGL Engine 36
Monitoring GPU Use 37
Using Window Resizing to Diagnose Performance 38
Identifying Unmatched Calls 38

Chapter 4 **Controlling Profiling Programmatically 41**

Setting a Breakpoint 41
Writing Comments to the Trace Window 42
Controlling Trace Collection 42
Controlling Statistics Collection 43

**Document Revision History 45**

# Figures and Listings

**Chapter 1** **Getting Started 9**

Figure 1-1 The main window in OpenGL Profiler 12

**Chapter 2** **Using Breakpoints 21**

Figure 2-1 The Breakpoints window displaying the call stack 22
Figure 2-2 The Breakpoints window displaying OpenGL state variables 23
Figure 2-3 The Resources window open to vertex buffer objects 24
Figure 2-4 The Textures pane in the Resources window 25
Figure 2-5 The Shaders pane in the Resources window 27
Figure 2-6 The Renderbuffers pane in the Resources window 28
Figure 2-7 The Depth Buffer window 30
Listing 2-1 A program for a basic shader 26

**Chapter 3** **Identifying and Solving Performance Issues 33**

Listing 3-1 A shell script for finding the slowest calls in a trace 35

**Chapter 4** **Controlling Profiling Programmatically 41**

Listing 4-1 Code that sets a breakpoint 41
Listing 4-2 Code that writes a comment to the Trace window 42
Listing 4-3 Code that enables trace collection 42
Listing 4-4 Code that clears the Trace window 43
Listing 4-5 Code that starts and stops statistics collection 43

# Introduction

OpenGL Profiler is an application that's useful for debugging and assessing performance. It lets you look inside a running application and observe how the application uses OpenGL. It can track the OpenGL functions used by an application, how often each is used, and the execution time of each function. Using this data, you can determine how efficiently an OpenGL application uses the GPU. You can then use the data to guide application development, modifying those parts of the code that slow performance or appear to use resources inefficiently.

OpenGL Profiler has a variety of interactive features. After setting breakpoints, developers can investigate application resources (textures, programs, shaders, and so on), examine the values of OpenGL context parameters, look at buffer contents, and check other aspects of the OpenGL state.

You'll want to read this document if you develop applications that use OpenGL on Mac OS X. By reading it, you'll learn how to set up OpenGL Profiler, collect data, set breakpoints, and use the results to track down problems.

## Organization of This Document

This document is organized into the following chapters:

- "Getting Started" (page 9) shows how to get OpenGL Profiler running and how to start a profiling session.
- "Using Breakpoints" (page 21) provides details on setting breakpoints and describes the tasks you can accomplish when your application pauses.
- "Identifying and Solving Performance Issues" (page 33) gives advice on how to use OpenGL Profiler to track down and analyze performance issues in your application.
- "Controlling Profiling Programmatically" (page 41) describes how to add code to your application that will control various aspects of OpenGL Profiler during a profiling session.

## See Also

These documents contain information that can help you analyze and optimize your OpenGL code:

- *OpenGL Programming Guide for Mac OS X* shows how to program using OpenGL on Mac OS X. You'll want to read the chapter "Improving Performance" to get an overview of the best programming practices to use as well as how to use Apple's tools (Shark and top) for identifying bottlenecks, and gathering and analyzing performance data.
- *OpenGL Driver Monitor User Guide* which is a developer tool that lets you investigate how the graphics processing unit (GPU) works on a system-wide basis.

# Getting Started

Users expect OpenGL applications to have fluid graphics that display without glitches. The more complex the graphics in an OpenGL application are, the more important it is for you to optimize performance and use resources wisely. When an OpenGL application performs less optimally than desired, the cause is often not obvious. That's where the OpenGL Profiler can be of value to you. After installing the Mac OS X Developer Tools, you can find OpenGL Profiler in `/Developer/Applications/Graphics Tools`.

This chapter:

- Gives an overview of how OpenGL Profiler works
- Shows how to set up your computer to use the profiler
- Describes how to start a profiling session for an OpenGL application
- Explains how to view and interpret data collected by the profiler
- Tells how to look at and modify application resources and parameters

Before reading this chapter, you may want to read the "Improving Performance" chapter in *OpenGL Programming Guide for Mac OS X*. That chapter provides a list of best programming practices for OpenGL and shows how to gather baseline data using a few Apple tools in addition to OpenGL Profiler, including Shark. In fact, prior to using OpenGL Profiler, it's best to start your analysis with Shark. The results from Shark will help you determine how to focus your efforts with OpenGL Profiler, or whether you even need to use the profiler application. It may be that your application's problems are not due to your OpenGL code!

## Overview

OpenGL Profiler collects trace and statistics for applications that use OpenGL. A **trace** is an ordered list of the OpenGL calls made by an application. Each entry in a trace shows a function name and the values of the parameters passed to the function. Statistics show cumulative totals, by function, for the number of times an application calls a function and the execution time of the function. You can also see the average execution time for a function, the percentage of time a function is used by OpenGL, and the percentage of time a function is used by the application.

The OpenGL functions that you'll see in the trace and statistics include those defined by the OpenGL specification (see http://www.opengl.org) as well as the functions that are part of the low-level Mac OS X OpenGL programming interfaces—CGL. (See *CGL Reference*.)

OpenGL Profiler is also useful for inspecting and controlling various aspects of your application. For example, you can:

- View the OpenGL resources your application uses, such as textures, vertex programs, and shaders.
- View buffer contents, such as the depth and back buffers
- Set breakpoints on specific OpenGL functions and view the call stack

- Attach a script of OpenGL commands that executes at a breakpoint
- Enable or disable individual OpenGL commands

## Before You Use OpenGL Profiler on a System

If you've never used OpenGL Profiler on your system, if you want to allow the profiler to attach to a running application you must set the environment variable `GL_ENABLE_DEBUG_ATTACH`. You also might want to set preferences.

### Setting Up the Environment Variables

There are two locations that require the environment variable:

- Your shell startup file. The name of this file depends on the shell that you use: `~/.login`, `~/.cshrc`, `~/.bashrc`, `~/.profile`, and so on. Add the following to the appropriate shell file:

  ```
  setenv GL_ENABLE_DEBUG_ATTACH YES
  ```

- The Mac OS X environment property list file (`~/.MacOSX/environment.plist`). The Finder uses this file to set the environment variable when it launches applications. Typically OpenGL Profiler creates this file for you the first time that you launch OpenGL Profiler on a system. The first time you launch the application you should see a dialog that asks whether you want to enable the Attach feature. Click Enable to have the property list file created for you.

  You can skip the rest of this section unless, for some reason, the property list file is missing. If that's the case, to use the Attach feature you'll need to create a plain text file, name it `environment.plist`, and enter the following into the file:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
  <plist version="1.0">
  <dict>
          <key>GL_ENABLE_DEBUG_ATTACH</key>
          <string>YES</string>
  </dict>
  </plist>
  ```

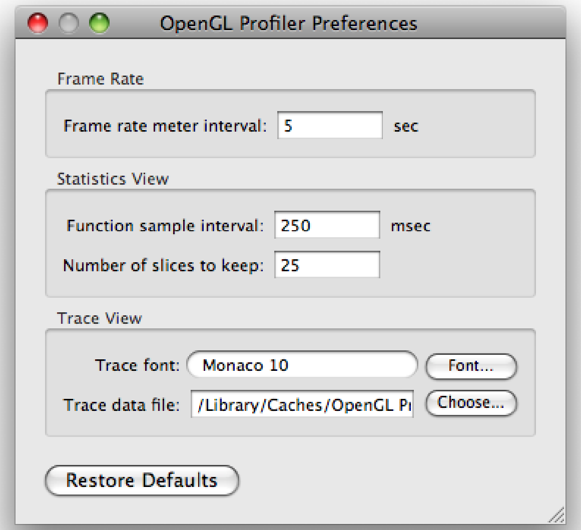
  Copy the file to your `~/.MacOSX` directory. If the directory does not exist, open Terminal and create the directory from within your shell.

### Setting Preferences (Optional)

You can view preferences by choosing OpenGL Profiler > Preferences. You can set the following:

- Frame rate meter interval, which allows you to change the frames-per-second readout from you application. If you need a finer grain of measurement and a more immediate result, you might want to increase this value.

- The function sample interval in the statistics view
- The number of slices to keep in the statistics view. You can limit the number of timing samples that OpenGL Profiler records.
- Trace view font. You can set the font face and the font size.
- Trace data file. You can change the file location and name.


OpenGL Profiler Preferences

Frame Rate

Frame rate meter interval: 5 sec

Statistics View

Function sample interval: 250 msec

Number of slices to keep: 25

Trace View

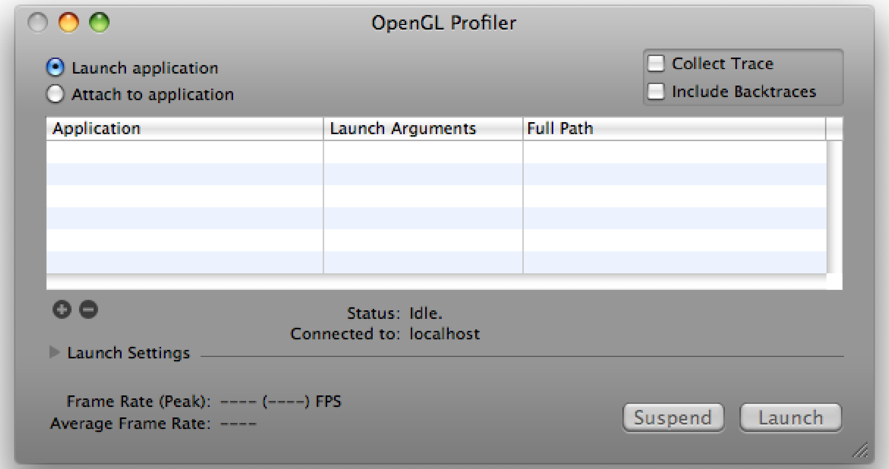
Trace font: Monaco 10

Trace data file: /Library/Caches/OpenGL Pı

## Starting a Profiling Session

When you double-click the OpenGL Profiler icon, the window shown in Figure 1-1 (page 12) opens. You use this window to set up a profiling session. You can start profiling an application either by launching it through OpenGL Profiler or by attaching the profiler to an application that's already running.

**Figure 1-1** The main window in OpenGL Profiler



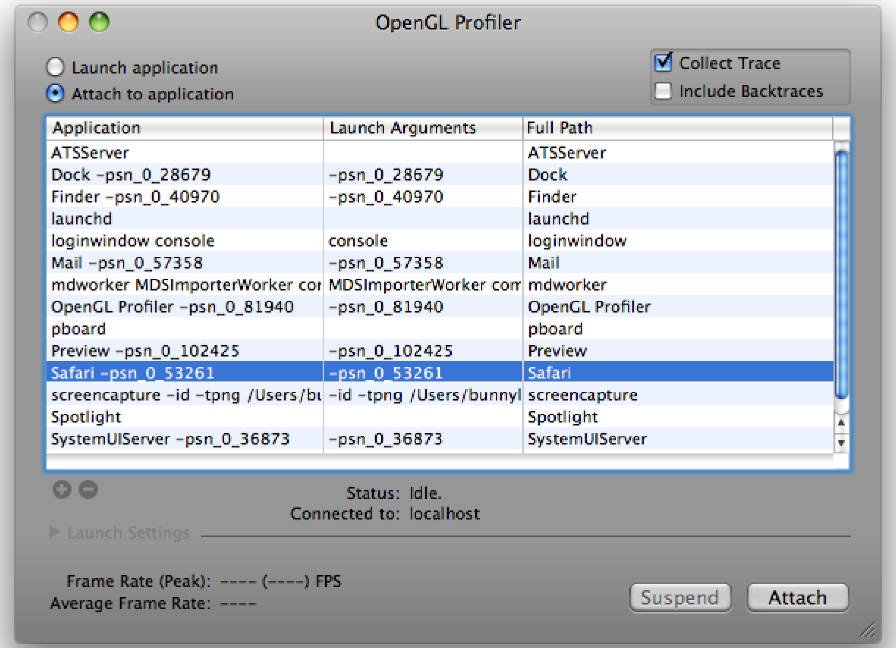
Before you attach or launch your application, select Collect Trace. You also have the option to collect backtraces. After your application launches or attaches, you may see small pauses or stutters in the application. This is normal behavior that is due to the large amount of data that OpenGL Profiler writes out when collecting a trace. It does not significantly affect the performance statistics.

## Attaching to a Running Application

To attach OpenGL Profiler to a running application:

1. Select "Attach to application."

2. In the list that appears, select the application you want to profile.



OpenGL Profiler

( ) Launch application
(●) Attach to application

[✓] Collect Trace
[ ] Include Backtraces

| Application | Launch Arguments | Full Path |
|---|---|---|
| ATSServer | | ATSServer |
| Dock -psn_0_28679 | -psn_0_28679 | Dock |
| Finder -psn_0_40970 | -psn_0_40970 | Finder |
| launchd | | launchd |
| loginwindow console | console | loginwindow |
| Mail -psn_0_57358 | -psn_0_57358 | Mail |
| mdworker MDSImporterWorker cor | MDSImporterWorker com | mdworker |
| OpenGL Profiler -psn_0_81940 | -psn_0_81940 | OpenGL Profiler |
| pboard | | pboard |
| Preview -psn_0_102425 | -psn_0_102425 | Preview |
| Safari -psn_0_53261 | -psn_0_53261 | Safari |
| screencapture -id -tpng /Users/bu | -id -tpng /Users/bunnyl | screencapture |
| Spotlight | | Spotlight |
| SystemUIServer -psn_0_36873 | -psn_0_36873 | SystemUIServer |

Status: Idle.
Connected to: localhost

Frame Rate (Peak): ---- (----) FPS
Average Frame Rate: ----

3. Click Attach.

   The status changes from idle to running, and the application list dims. Now that the profiler is attached to the application, you can perform any of the tasks described in the rest of this document.

   OpenGL Profiler begins collecting data as soon as it attaches to your application. Data collection ends when you quit the application or click the Detach button in the OpenGL Profiler window. You can temporarily stop data collection by clicking Suspend.

   Detaching lets the application continue to run, which is useful for applications (like shell tools) that do not have a user interface.

## Launching an Application from OpenGL Profiler

To set up OpenGL Profiler to launch an application:

1. Select “Launch application.”

2. Click the plus (+) button and navigate to the application you what to profile, then choose it.

   You can add as many applications as you’d like, but you can profile only one at a time.

3. If your application needs launch arguments, click the Launch Arguments text field and add them.

4. Click the disclosure triangle to view the optional launch settings.

You can use these settings when you want to profile an application under specific conditions. For example, you can use a custom pixel format, simulate how the application would work using a specific graphics driver, and set additional environment variables. See "Customizing Launch Settings" (page 14) for details.

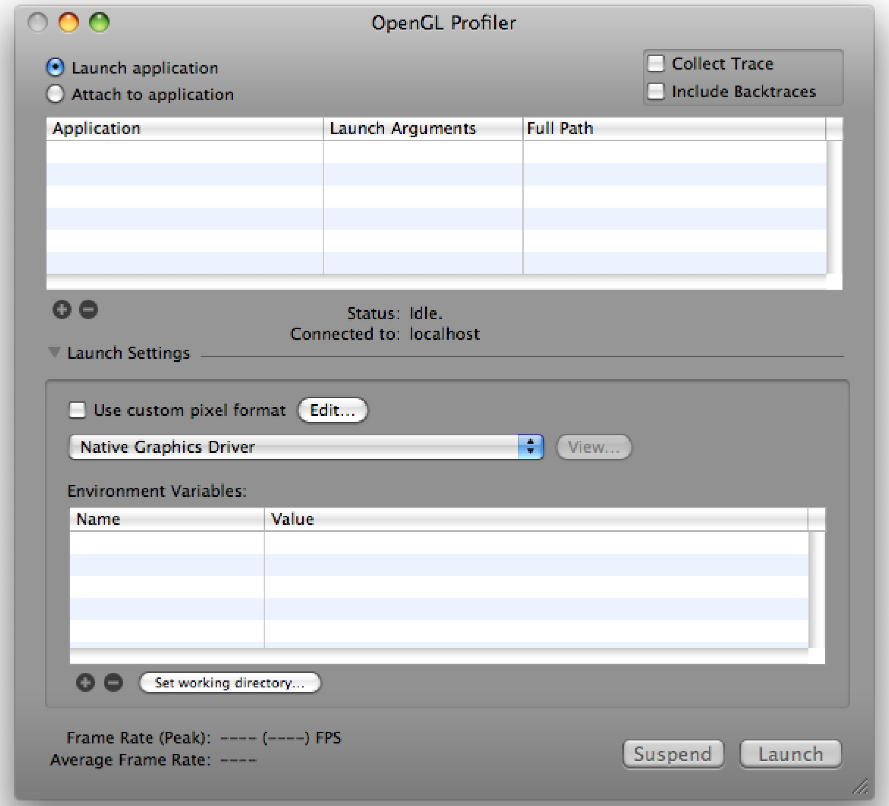
5. Click Launch.

   The status changes from idle to running, and the application and environment variables lists dim.

   OpenGL Profiler begins collecting data as soon as it launches your application. Data collection ends when you quit the application or click the Kill button in the OpenGL Profiler window. You can temporarily stop data collection by clicking Suspend.

## Customizing Launch Settings

Click the disclosure triangle next to Launch Settings in the OpenGL Profiler window to customize how your application launches. These options allow you to observe how your application operates under specific conditions.



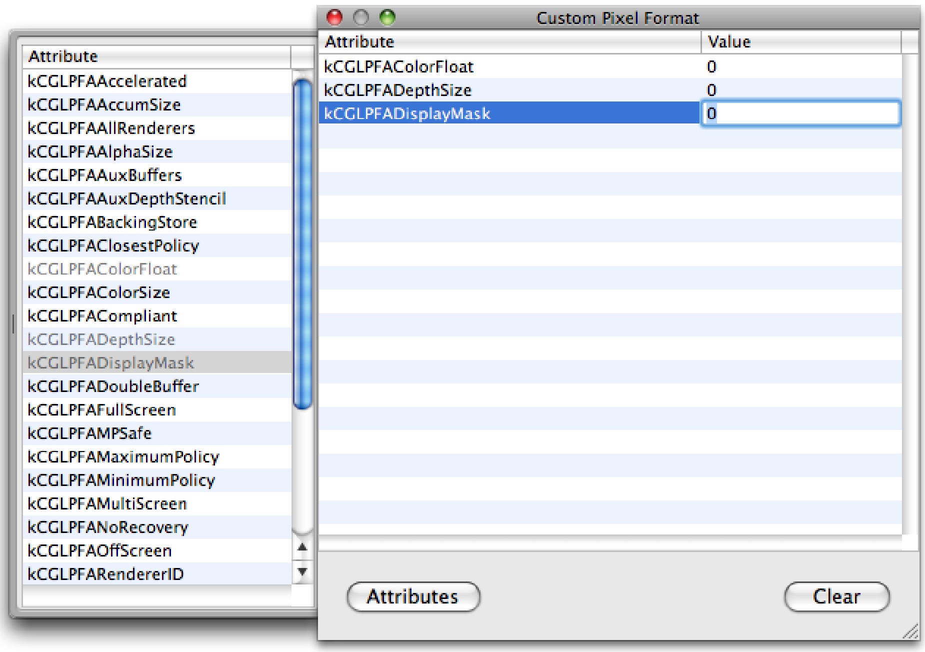
Status: Idle.
Connected to: localhost

Native Graphics Driver

Frame Rate (Peak): ---- (----) FPS
Average Frame Rate: ----

## Setting a Custom Pixel Format

Select “Use custom pixel format” and click Edit to try out different settings for pixel format attributes. You can quickly see the results of modifying attribute values without changing the source code of the application.



| Attribute |
|---|
| kCGLPFAAccelerated |
| kCGLPFAAccumSize |
| kCGLPFAAllRenderers |
| kCGLPFAAlphaSize |
| kCGLPFAAuxBuffers |
| kCGLPFAAuxDepthStencil |
| kCGLPFABackingStore |
| kCGLPFAClosestPolicy |
| kCGLPFAColorFloat |
| kCGLPFAColorSize |
| kCGLPFACompliant |
| kCGLPFADepthSize |
| kCGLPFADisplayMask |
| kCGLPFADoubleBuffer |
| kCGLPFAFullScreen |
| kCGLPFAMPSafe |
| kCGLPFAMaximumPolicy |
| kCGLPFAMinimumPolicy |
| kCGLPFAMultiScreen |
| kCGLPFANoRecovery |
| kCGLPFAOffScreen |
| kCGLPFARendererID |

| Attribute | Value |
|---|---|
| kCGLPFAColorFloat | 0 |
| kCGLPFADepthSize | 0 |
| kCGLPFADisplayMask | 0 |

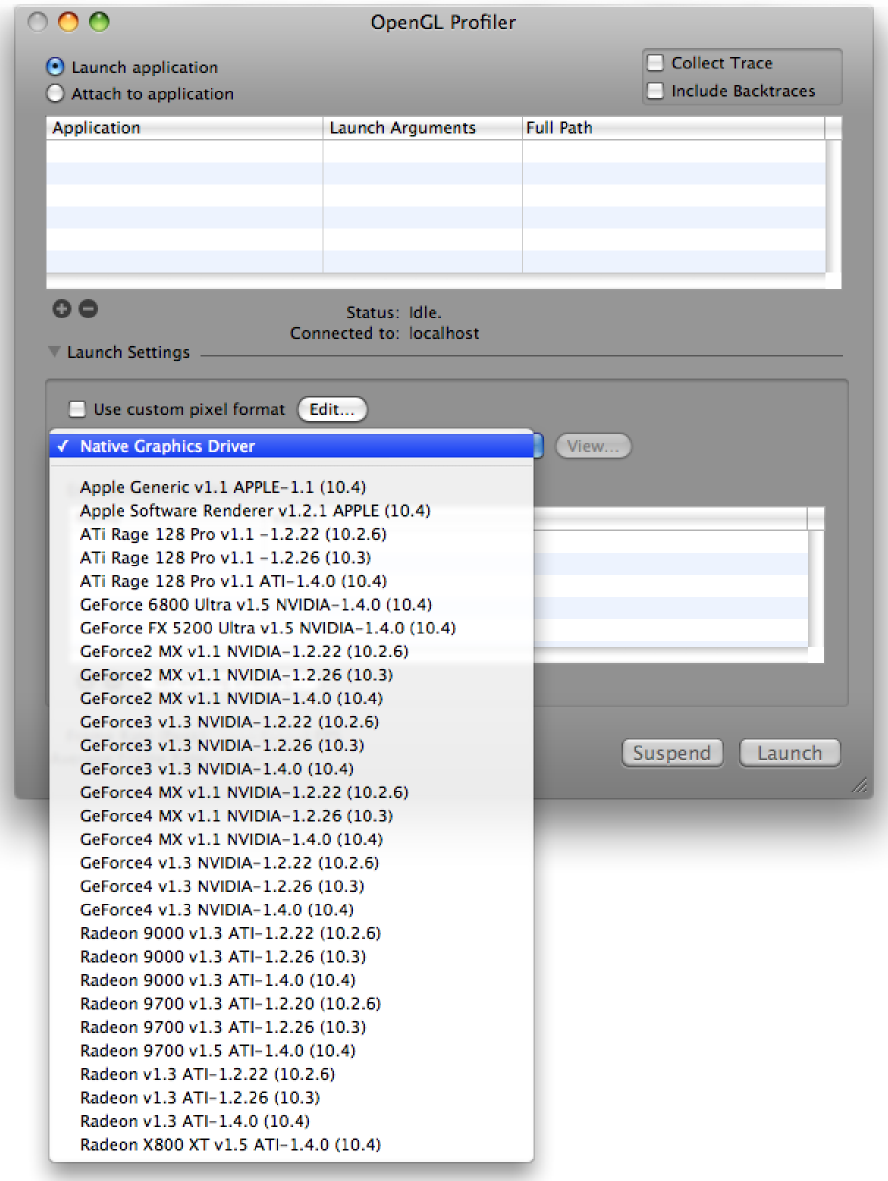
## Using a Driver Emulation

OpenGL Profiler contains parameters and values for a variety of OpenGL drivers, not only those available in the system that you are currently running. This feature comes in handy if you want to examine the values that OpenGL returns for a particular driver. Emulation mode does not affect the performance of your application in any way. The purpose of this feature is to help you determine whether code that depends on certain hardware features works properly. Keep in mind that OpenGL Profiler does not replace the drivers your system.

When using a driver emulation, OpenGL Profiler does changes the return values for some of the `glGet` functions. For example, if you enable the Rage 128 emulator for Mac OS X 10.2 and earlier, calling `glGetString(GL_RENDERER)` returns `"ATi Rage 128 Pro OpenGL Engine"` instead of the renderer that’s actually in your system. In addition, any `glGet` functions that return driver-specific parameters will return the values you’d get if your application ran on the emulated card. For example, calling `glGetInteger(GL_MAX_LIGHTS, &maxLights)` assigns `8` to `maxLights`.

To set a specific driver:

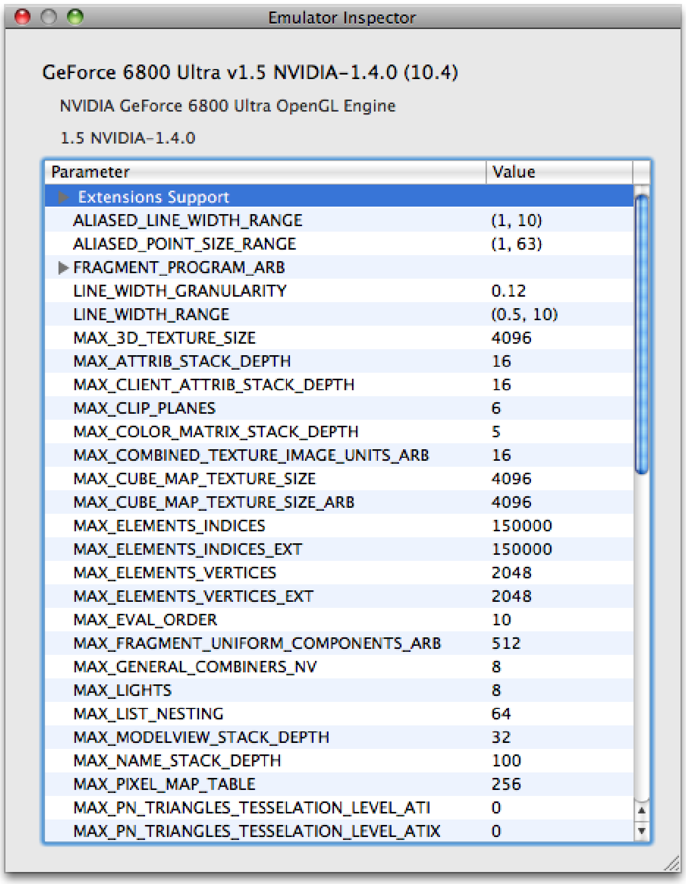
1. Choose a driver from the pop-up menu.



✓ Native Graphics Driver

Apple Generic v1.1 APPLE-1.1 (10.4)
Apple Software Renderer v1.2.1 APPLE (10.4)
ATi Rage 128 Pro v1.1 -1.2.22 (10.2.6)
ATi Rage 128 Pro v1.1 -1.2.26 (10.3)
ATi Rage 128 Pro v1.1 ATI-1.4.0 (10.4)
GeForce 6800 Ultra v1.5 NVIDIA-1.4.0 (10.4)
GeForce FX 5200 Ultra v1.5 NVIDIA-1.4.0 (10.4)
GeForce2 MX v1.1 NVIDIA-1.2.22 (10.2.6)
GeForce2 MX v1.1 NVIDIA-1.2.26 (10.3)
GeForce2 MX v1.1 NVIDIA-1.4.0 (10.4)
GeForce3 v1.3 NVIDIA-1.2.22 (10.2.6)
GeForce3 v1.3 NVIDIA-1.2.26 (10.3)
GeForce3 v1.3 NVIDIA-1.4.0 (10.4)
GeForce4 MX v1.1 NVIDIA-1.2.22 (10.2.6)
GeForce4 MX v1.1 NVIDIA-1.2.26 (10.3)
GeForce4 MX v1.1 NVIDIA-1.4.0 (10.4)
GeForce4 v1.3 NVIDIA-1.2.22 (10.2.6)
GeForce4 v1.3 NVIDIA-1.2.26 (10.3)
GeForce4 v1.3 NVIDIA-1.4.0 (10.4)
Radeon 9000 v1.3 ATI-1.2.22 (10.2.6)
Radeon 9000 v1.3 ATI-1.2.26 (10.3)
Radeon 9000 v1.3 ATI-1.4.0 (10.4)
Radeon 9700 v1.3 ATI-1.2.20 (10.2.6)
Radeon 9700 v1.3 ATI-1.2.26 (10.3)
Radeon 9700 v1.5 ATI-1.4.0 (10.4)
Radeon v1.3 ATI-1.2.22 (10.2.6)
Radeon v1.3 ATI-1.2.26 (10.3)
Radeon v1.3 ATI-1.4.0 (10.4)
Radeon X800 XT v1.5 ATI-1.4.0 (10.4)

2. Click View to inspect the parameters and values for the driver.



GeForce 6800 Ultra v1.5 NVIDIA-1.4.0 (10.4)

NVIDIA GeForce 6800 Ultra OpenGL Engine

1.5 NVIDIA-1.4.0

| Parameter | Value |
|---|---|
| ▶ Extensions Support | |
| ALIASED_LINE_WIDTH_RANGE | (1, 10) |
| ALIASED_POINT_SIZE_RANGE | (1, 63) |
| ▶ FRAGMENT_PROGRAM_ARB | |
| LINE_WIDTH_GRANULARITY | 0.12 |
| LINE_WIDTH_RANGE | (0.5, 10) |
| MAX_3D_TEXTURE_SIZE | 4096 |
| MAX_ATTRIB_STACK_DEPTH | 16 |
| MAX_CLIENT_ATTRIB_STACK_DEPTH | 16 |
| MAX_CLIP_PLANES | 6 |
| MAX_COLOR_MATRIX_STACK_DEPTH | 5 |
| MAX_COMBINED_TEXTURE_IMAGE_UNITS_ARB | 16 |
| MAX_CUBE_MAP_TEXTURE_SIZE | 4096 |
| MAX_CUBE_MAP_TEXTURE_SIZE_ARB | 4096 |
| MAX_ELEMENTS_INDICES | 150000 |
| MAX_ELEMENTS_INDICES_EXT | 150000 |
| MAX_ELEMENTS_VERTICES | 2048 |
| MAX_ELEMENTS_VERTICES_EXT | 2048 |
| MAX_EVAL_ORDER | 10 |
| MAX_FRAGMENT_UNIFORM_COMPONENTS_ARB | 512 |
| MAX_GENERAL_COMBINERS_NV | 8 |
| MAX_LIGHTS | 8 |
| MAX_LIST_NESTING | 64 |
| MAX_MODELVIEW_STACK_DEPTH | 32 |
| MAX_NAME_STACK_DEPTH | 100 |
| MAX_PIXEL_MAP_TABLE | 256 |
| MAX_PN_TRIANGLES_TESSELATION_LEVEL_ATI | 0 |
| MAX_PN_TRIANGLES_TESSELATION_LEVEL_ATIX | 0 |

Apple provides a set of driver emulation files. These are property lists that you can edit yourself. You can create an emulation file for a driver not in the list by using a text editor and then saving the file in this directory:

```
~/Library/OpenGL Profiler/Driver Emulators
```

You'll need to restart OpenGL Profiler to see the new file in the driver list.

## Setting Environment Variables

You can add environment variables by clicking the plus (+) button. Then enter the variable name and its value. You can add any variable your application reads using `getenv(3)`.

## Setting a Working Directory

The working directory refers to your application's current working directory. It's the path returned by the function `getcwd(3)` and the same path that's automatically prepended to any relative path in your application. If your application does not conform to Apple's application wrapper scheme, you need to set the working directory so that your application can find resources such as texture maps.

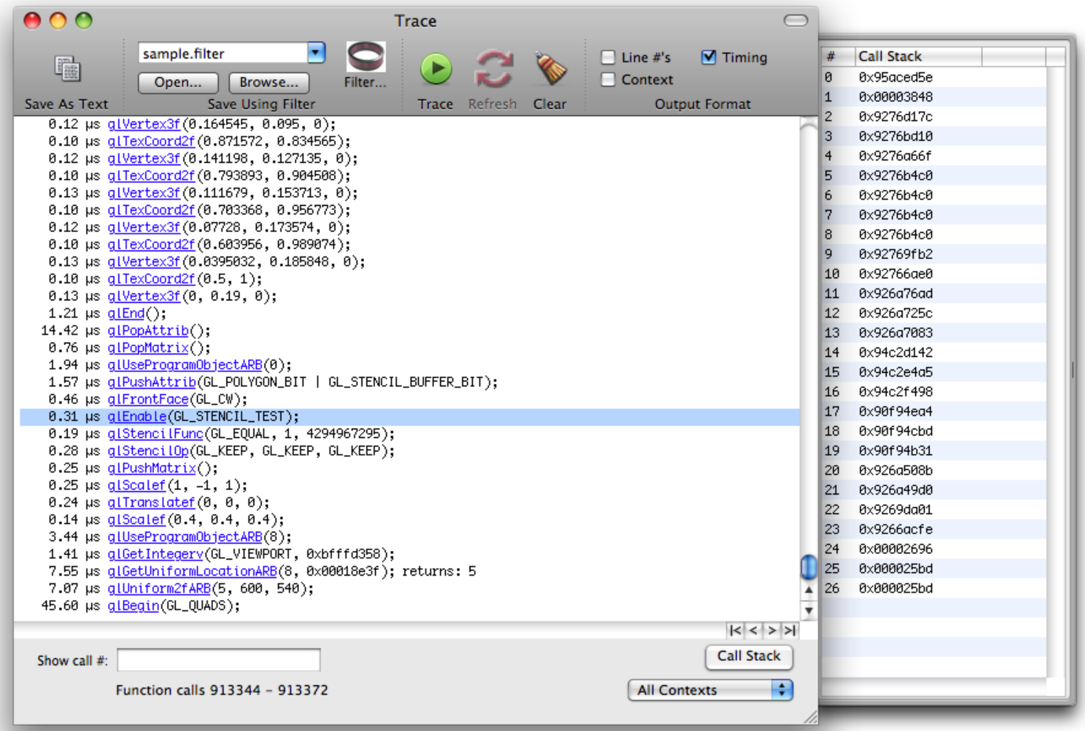
For example, if you launch a nonconforming application from Finder, for this line of code to execute properly:

```
fopen("mytexture", "r+")
```

you need to set the working directory to the folder that contains `"mytexture"`.

## Viewing a Trace

You can view a trace by choosing Views > Trace. The Trace window displays a running list of the OpenGL function calls your application makes. The time value next to the function name gives you an idea of the performance cost of the call. Keep in mind that a lot of OpenGL calls set bits and are acted upon only at draw or flush time. This means that in addition to looking at the time for a particular function, you also need to consider the execution time of the next draw or flush function.



sample.filter
Line #'s
Context
[✓ Timing]

```
    0.12 µs glVertex3f(0.164545, 0.095, 0);
    0.10 µs glTexCoord2f(0.871572, 0.834565);
    0.12 µs glVertex3f(0.141198, 0.127135, 0);
    0.10 µs glTexCoord2f(0.793893, 0.904508);
    0.13 µs glVertex3f(0.111679, 0.153713, 0);
    0.10 µs glTexCoord2f(0.703368, 0.956773);
    0.12 µs glVertex3f(0.07728, 0.173574, 0);
    0.10 µs glTexCoord2f(0.603956, 0.989074);
    0.13 µs glVertex3f(0.0395032, 0.185848, 0);
    0.10 µs glTexCoord2f(0.5, 1);
    0.13 µs glVertex3f(0, 0.19, 0);
    1.21 µs glEnd();
   14.42 µs glPopAttrib();
    0.76 µs glPopMatrix();
    1.94 µs glUseProgramObjectARB(0);
    1.57 µs glPushAttrib(GL_POLYGON_BIT | GL_STENCIL_BUFFER_BIT);
    0.46 µs glFrontFace(GL_CW);
    0.31 µs glEnable(GL_STENCIL_TEST);
    0.19 µs glStencilFunc(GL_EQUAL, 1, 4294967295);
    0.28 µs glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    0.25 µs glPushMatrix();
    0.25 µs glScalef(1, -1, 1);
    0.24 µs glTranslatef(0, 0, 0);
    0.14 µs glScalef(0.4, 0.4, 0.4);
    3.44 µs glUseProgramObjectARB(8);
    1.41 µs glGetIntegerv(GL_VIEWPORT, 0xbfffd358);
    7.55 µs glGetUniformLocationARB(8, 0x00018e3f); returns: 5
    7.07 µs glUniform2fARB(5, 600, 540);
   45.60 µs glBegin(GL_QUADS);
```

Show call #:

Function calls 913344 – 913372

All Contexts

| # | Call Stack |
|---|---|
| 0 | 0x95aced5e |
| 1 | 0x00003848 |
| 2 | 0x9276d17c |
| 3 | 0x9276bd10 |
| 4 | 0x9276a66f |
| 5 | 0x9276b4c0 |
| 6 | 0x9276b4c0 |
| 7 | 0x9276b4c0 |
| 8 | 0x9276b4c0 |
| 9 | 0x92769fb2 |
| 10 | 0x92766ae0 |
| 11 | 0x926a76ad |
| 12 | 0x926a725c |
| 13 | 0x926a7083 |
| 14 | 0x94c2d142 |
| 15 | 0x94c2e4a5 |
| 16 | 0x94c2f498 |
| 17 | 0x90f94ea4 |
| 18 | 0x90f94cbd |
| 19 | 0x90f94b31 |
| 20 | 0x926a508b |
| 21 | 0x926a49d0 |
| 22 | 0x9269da01 |
| 23 | 0x9266acfe |
| 24 | 0x00002696 |
| 25 | 0x000025bd |
| 26 | 0x000025bd |

For strategies on interpreting trace data, see:

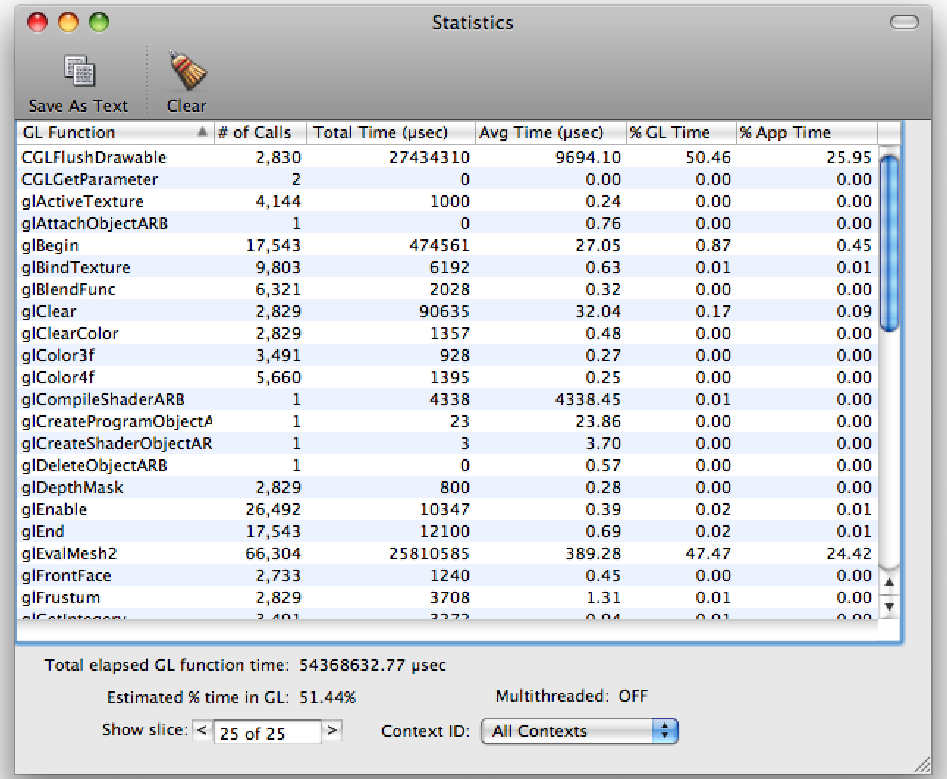
- "Making Sure You Use Functions Correctly" (page 33)

- "Identifying Problem Areas in Your Application" (page 34)
- "Managing Trace Data" (page 35)

## Viewing Statistics

Choose Views > Statistics to open the Statistics window. This view aggregates the trace data so that you can see function call frequency, execution time sums and averages, and these percentages:

- % GL Time indicates the amount of time a function takes to execute compared only to the OpenGL code in an application.
- % Application Time indicates the amount of time a function takes to execute compared to all code in an application.
- Estimated % Time in OpenGL indicates the amount of time an application spends executing OpenGL code compared to all application code.



| GL Function | # of Calls | Total Time (µsec) | Avg Time (µsec) | % GL Time | % App Time |
|---|---|---|---|---|---|
| CGLFlushDrawable | 2,830 | 27434310 | 9694.10 | 50.46 | 25.95 |
| CGLGetParameter | 2 | 0 | 0.00 | 0.00 | 0.00 |
| glActiveTexture | 4,144 | 1000 | 0.24 | 0.00 | 0.00 |
| glAttachObjectARB | 1 | 0 | 0.76 | 0.00 | 0.00 |
| glBegin | 17,543 | 474561 | 27.05 | 0.87 | 0.45 |
| glBindTexture | 9,803 | 6192 | 0.63 | 0.01 | 0.01 |
| glBlendFunc | 6,321 | 2028 | 0.32 | 0.00 | 0.00 |
| glClear | 2,829 | 90635 | 32.04 | 0.17 | 0.09 |
| glClearColor | 2,829 | 1357 | 0.48 | 0.00 | 0.00 |
| glColor3f | 3,491 | 928 | 0.27 | 0.00 | 0.00 |
| glColor4f | 5,660 | 1395 | 0.25 | 0.00 | 0.00 |
| glCompileShaderARB | 1 | 4338 | 4338.45 | 0.01 | 0.00 |
| glCreateProgramObjectA | 1 | 23 | 23.86 | 0.00 | 0.00 |
| glCreateShaderObjectAR | 1 | 3 | 3.70 | 0.00 | 0.00 |
| glDeleteObjectARB | 1 | 0 | 0.57 | 0.00 | 0.00 |
| glDepthMask | 2,829 | 800 | 0.28 | 0.00 | 0.00 |
| glEnable | 26,492 | 10347 | 0.39 | 0.02 | 0.01 |
| glEnd | 17,543 | 12100 | 0.69 | 0.02 | 0.01 |
| glEvalMesh2 | 66,304 | 25810585 | 389.28 | 47.47 | 24.42 |
| glFrontFace | 2,733 | 1240 | 0.45 | 0.00 | 0.00 |
| glFrustum | 2,829 | 3708 | 1.31 | 0.01 | 0.00 |

Total elapsed GL function time: 54368632.77 µsec

Estimated % time in GL: 51.44%

Multithreaded: OFF

Show slice: < 25 of 25 >

Context ID: All Contexts

These statistics can help you identify the portions of your OpenGL code that consume the most time. Finding that your application stalls during certain calls or that some functions seem to be called at an unusually high frequency can help you pinpoint the portions of your code that might need fine-tuning.

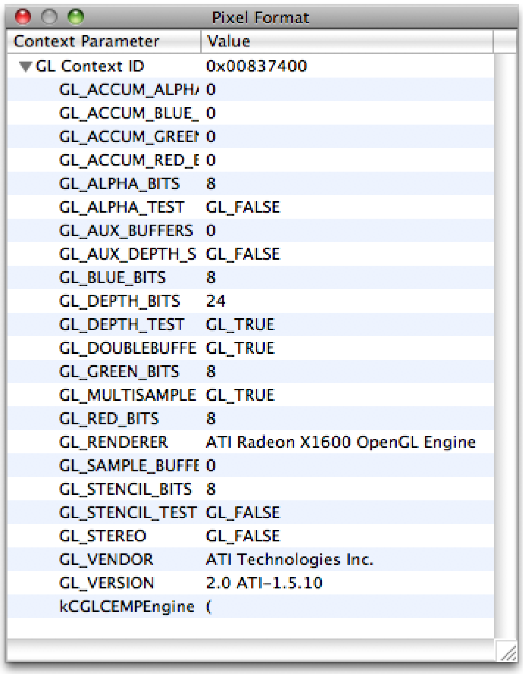
One way to find out where to focus your optimization efforts is to compare the time spent executing OpenGL calls to the time spent executing non-OpenGL calls. For example, if your application runs slowly but spends most of its time executing non-OpenGL calls, you'll get the most performance gains by analyzing and optimizing the non-OpenGL portion.

For other strategies on interpreting statistics data, see:

- "Making Sure You Use Functions Correctly" (page 33)
- "Identifying Problem Areas in Your Application" (page 34)

## Viewing Pixel Format Context Parameters

To view the context parameters for the application's pixel format, choose Views > Pixel Format. For each OpenGL context, you'll see a list of its parameters and the value for each parameter.



| Context Parameter | Value |
|---|---|
| ▼GL Context ID | 0x00837400 |
| GL_ACCUM_ALPH/ | 0 |
| GL_ACCUM_BLUE_ | 0 |
| GL_ACCUM_GREEI | 0 |
| GL_ACCUM_RED_E | 0 |
| GL_ALPHA_BITS | 8 |
| GL_ALPHA_TEST | GL_FALSE |
| GL_AUX_BUFFERS | 0 |
| GL_AUX_DEPTH_S | GL_FALSE |
| GL_BLUE_BITS | 8 |
| GL_DEPTH_BITS | 24 |
| GL_DEPTH_TEST | GL_TRUE |
| GL_DOUBLEBUFFE | GL_TRUE |
| GL_GREEN_BITS | 8 |
| GL_MULTISAMPLE | GL_TRUE |
| GL_RED_BITS | 8 |
| GL_RENDERER | ATI Radeon X1600 OpenGL Engine |
| GL_SAMPLE_BUFFE | 0 |
| GL_STENCIL_BITS | 8 |
| GL_STENCIL_TEST | GL_FALSE |
| GL_STEREO | GL_FALSE |
| GL_VENDOR | ATI Technologies Inc. |
| GL_VERSION | 2.0 ATI-1.5.10 |
| kCGLCEMPEngine | ( |

# Using Breakpoints

You can use OpenGL Profiler to set breakpoints in your application. A breakpoint can occur before or after a specific OpenGL function, when your application uses a software renderer, or when an error or thread conflict occurs. When your application is paused, you can perform a number of tasks, including:

- Examining the call state and OpenGL state
- Viewing and modifying resources
- Viewing buffer contents
- Executing a script that you attach to the breakpoint

You can also choose to disable the execution of one or more OpenGL functions.

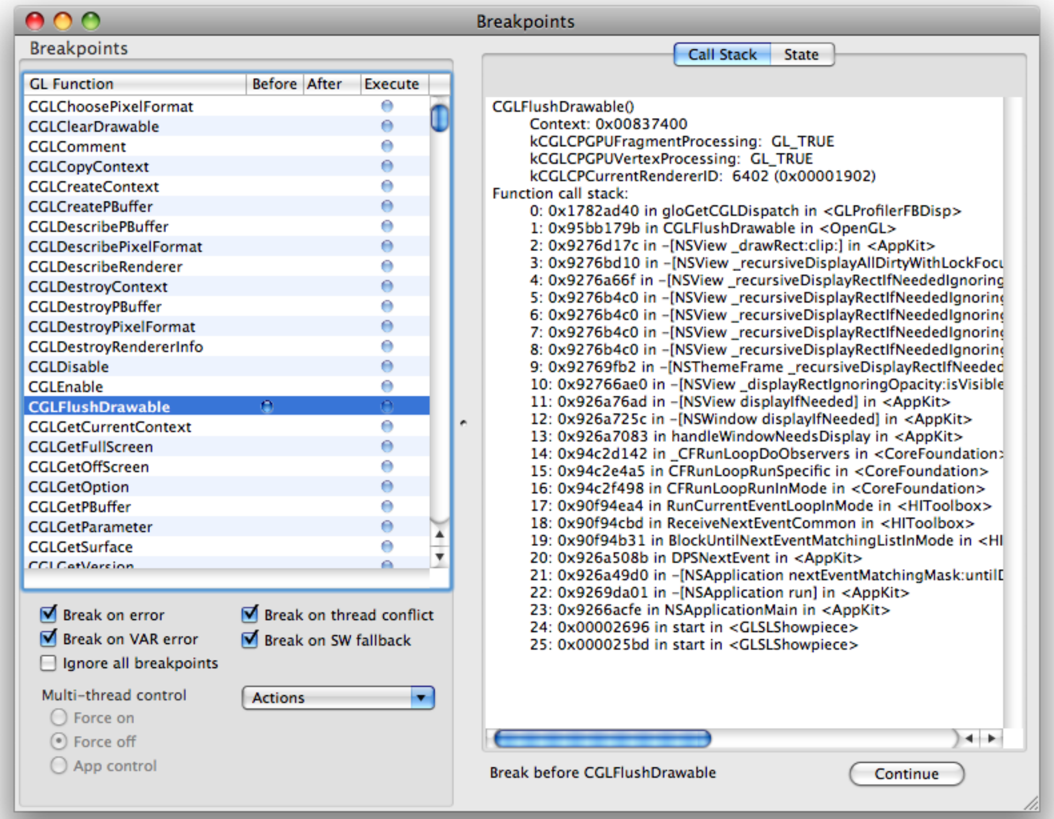
This chapter describes how to set breakpoints and describes all the tasks that you can perform only after your application pauses at a breakpoint.

## Setting Breakpoints on OpenGL Functions

You set breakpoints in the Breakpoints window. To open this window, choose Views > Breakpoints. As shown in Figure 2-1 (page 22), the left side of the window lists the OpenGL functions that you can set breakpoints on. This list includes functions defined by the OpenGL specification and the CGL calls available in Mac OS X. Keep in mind that AGL functions and Cocoa OpenGL class methods call into the CGL framework, so AGL and Cocoa methods aren't explicitly on the function list.

The right side of the window lets you view the call stack, as shown in Figure 2-1 (page 22), or OpenGL state variables, as shown in Figure 2-2 (page 23).

**Figure 2-1** The Breakpoints window displaying the call stack



Breakpoints

| GL Function | Before | After | Execute |
|---|---|---|---|
| CGLChoosePixelFormat | | | ● |
| CGLClearDrawable | | | ● |
| CGLComment | | | ● |
| CGLCopyContext | | | ● |
| CGLCreateContext | | | ● |
| CGLCreatePBuffer | | | ● |
| CGLDescribePBuffer | | | ● |
| CGLDescribePixelFormat | | | ● |
| CGLDescribeRenderer | | | ● |
| CGLDestroyContext | | | ● |
| CGLDestroyPBuffer | | | ● |
| CGLDestroyPixelFormat | | | ● |
| CGLDestroyRendererInfo | | | ● |
| CGLDisable | | | ● |
| CGLEnable | | | ● |
| CGLFlushDrawable | ● | | ● |
| CGLGetCurrentContext | | | ● |
| CGLGetFullScreen | | | ● |
| CGLGetOffScreen | | | ● |
| CGLGetOption | | | ● |
| CGLGetPBuffer | | | ● |
| CGLGetParameter | | | ● |
| CGLGetSurface | | | ● |
| CGLGetVersion | | | ● |

[✓] Break on error
[✓] Break on VAR error
[ ] Ignore all breakpoints
[✓] Break on thread conflict
[✓] Break on SW fallback

Multi-thread control
Force on [✓ Force off] App control

```
CGLFlushDrawable()
      Context: 0x00837400
      kCGLCPGPUFragmentProcessing:  GL_TRUE
      kCGLCPGPUVertexProcessing:  GL_TRUE
      kCGLCPCurrentRendererID:  6402 (0x00001902)
Function call stack:
      0: 0x1782ad40 in gloGetCGLDispatch in <GLProfilerFBDisp>
      1: 0x95bb179b in CGLFlushDrawable in <OpenGL>
      2: 0x9276d17c in -[NSView _drawRect:clip:] in <AppKit>
      3: 0x9276bd10 in -[NSView _recursiveDisplayAllDirtyWithLockFocu
      4: 0x9276a66f in -[NSView _recursiveDisplayRectIfNeededIgnoring
      5: 0x9276b4c0 in -[NSView _recursiveDisplayRectIfNeededIgnorin
      6: 0x9276b4c0 in -[NSView _recursiveDisplayRectIfNeededIgnorin
      7: 0x9276b4c0 in -[NSView _recursiveDisplayRectIfNeededIgnorin
      8: 0x9276b4c0 in -[NSView _recursiveDisplayRectIfNeededIgnorin
      9: 0x92769fb2 in -[NSThemeFrame _recursiveDisplayRectIfNeeded
      10: 0x92766ae0 in -[NSView _displayRectIgnoringOpacity:isVisible
      11: 0x926a76ad in -[NSView displayIfNeeded] in <AppKit>
      12: 0x926a725c in -[NSWindow displayIfNeeded] in <AppKit>
      13: 0x926a7083 in handleWindowNeedsDisplay in <AppKit>
      14: 0x94c2d142 in _CFRunLoopDoObservers in <CoreFoundation
      15: 0x94c2e4a5 in CFRunLoopRunSpecific in <CoreFoundation>
      16: 0x94c2f498 in CFRunLoopRunInMode in <CoreFoundation>
      17: 0x90f94ea4 in RunCurrentEventLoopInMode in <HIToolbox>
      18: 0x90f94cbd in ReceiveNextEventCommon in <HIToolbox>
      19: 0x90f94b31 in BlockUntilNextEventMatchingListInMode in <HI
      20: 0x926a508b in DPSNextEvent in <AppKit>
      21: 0x926a49d0 in -[NSApplication nextEventMatchingMask:untilD
      22: 0x9269da01 in -[NSApplication run] in <AppKit>
      23: 0x9266acfe in NSApplicationMain in <AppKit>
      24: 0x00002696 in start in <GLSLShowpiece>
      25: 0x000025bd in start in <GLSLShowpiece>
```

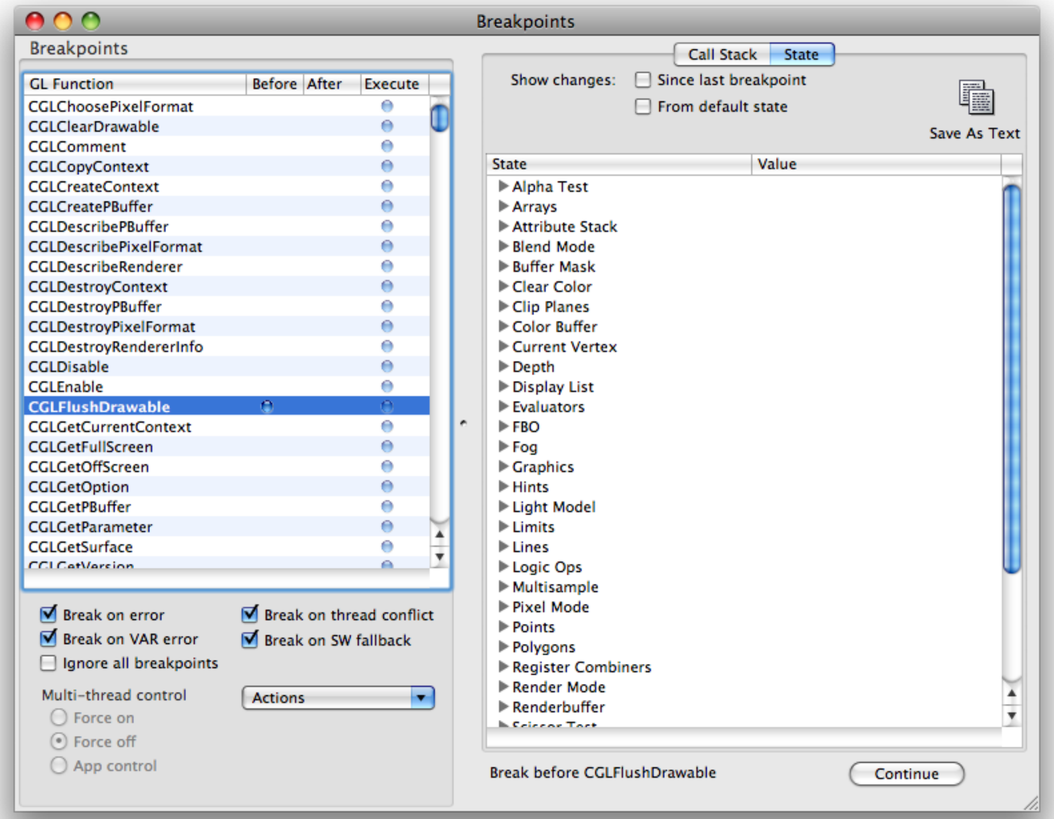
Break before CGLFlushDrawable

To set a breakpoint, choose the OpenGL function in the function list that you want to set a breakpoint on. Then click one of the following columns:

- Before pauses your application just before it executes the OpenGL function.
- After pauses your application just after it executes the OpenGL function.

A blue indicator appears wherever you set a breakpoint.

**Figure 2-2** The Breakpoints window displaying OpenGL state variables



| GL Function | Before | After | Execute |
|---|---|---|---|
| CGLChoosePixelFormat | | | ● |
| CGLClearDrawable | | | ● |
| CGLComment | | | ● |
| CGLCopyContext | | | ● |
| CGLCreateContext | | | ● |
| CGLCreatePBuffer | | | ● |
| CGLDescribePBuffer | | | ● |
| CGLDescribePixelFormat | | | ● |
| CGLDescribeRenderer | | | ● |
| CGLDestroyContext | | | ● |
| CGLDestroyPBuffer | | | ● |
| CGLDestroyPixelFormat | | | ● |
| CGLDestroyRendererInfo | | | ● |
| CGLDisable | | | ● |
| CGLEnable | | | ● |
| CGLFlushDrawable | ● | | ● |
| CGLGetCurrentContext | | | ● |
| CGLGetFullScreen | | | ● |
| CGLGetOffScreen | | | ● |
| CGLGetOption | | | ● |
| CGLGetPBuffer | | | ● |
| CGLGetParameter | | | ● |
| CGLGetSurface | | | ● |
| CGLGetVersion | | | ● |

☑ Break on error ☑ Break on thread conflict
☑ Break on VAR error ☑ Break on SW fallback
☐ Ignore all breakpoints

Multi-thread control
Force on [✓ Force off] App control

Show changes: ☐ Since last breakpoint ☐ From default state

| State | Value |
|---|---|
| ▶ Alpha Test | |
| ▶ Arrays | |
| ▶ Attribute Stack | |
| ▶ Blend Mode | |
| ▶ Buffer Mask | |
| ▶ Clear Color | |
| ▶ Clip Planes | |
| ▶ Color Buffer | |
| ▶ Current Vertex | |
| ▶ Depth | |
| ▶ Display List | |
| ▶ Evaluators | |
| ▶ FBO | |
| ▶ Fog | |
| ▶ Graphics | |
| ▶ Hints | |
| ▶ Light Model | |
| ▶ Limits | |
| ▶ Lines | |
| ▶ Logic Ops | |
| ▶ Multisample | |
| ▶ Pixel Mode | |
| ▶ Points | |
| ▶ Polygons | |
| ▶ Register Combiners | |
| ▶ Render Mode | |
| ▶ Renderbuffer | |
| ▶ Scissor Test | |

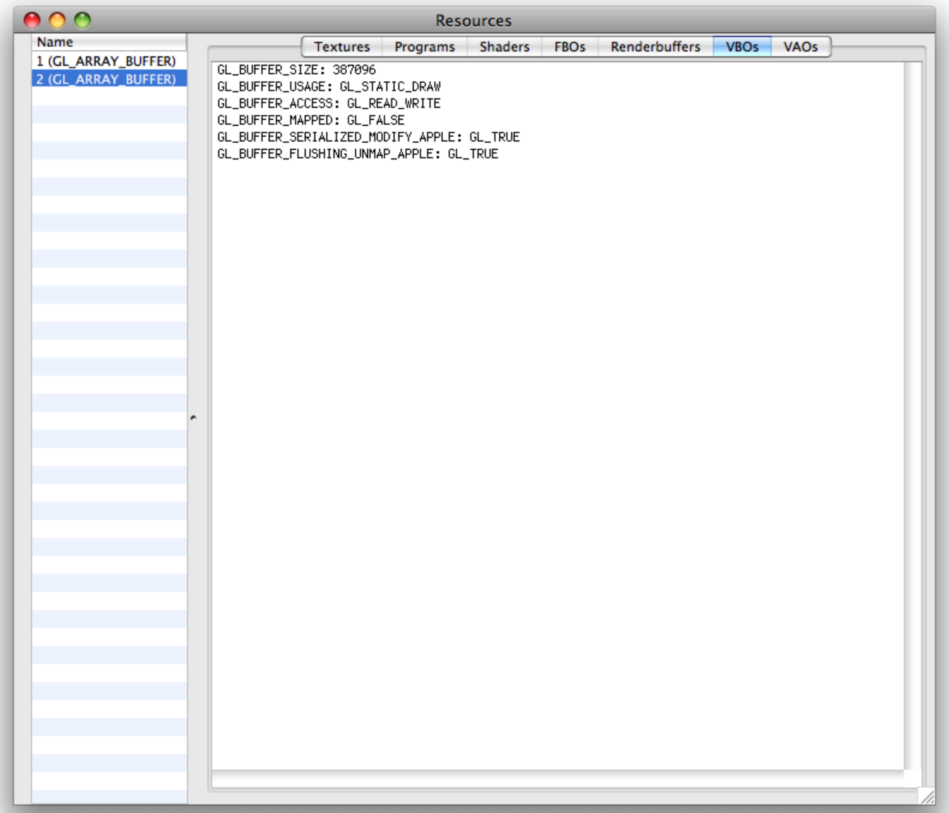
Break before CGLFlushDrawable

The checkboxes below the function list let you set breakpoint conditions. You can also turn the function list breakpoints on and off quickly by selecting "Ignore all breakpoints."

## Viewing and Modifying Resources

OpenGL resources include textures, programs, shaders, framebuffer objects, renderbuffer objects, vertex buffer objects, and vertex array objects. You can view these resources by choosing Views > Resources. The right side of the Resources window lets you choose a resource type. You can see the individual resources, for the resource types that your application uses, listed in the left side. Click a resource name to view its settings and other relevant information on the right side. For example, see Figure 2-3 (page 24) which shows two vertex buffer objects. The parameters and values for the one that's selected are shown on the right side.

**Figure 2-3** The Resources window open to vertex buffer objects



```
GL_BUFFER_SIZE: 387096
GL_BUFFER_USAGE: GL_STATIC_DRAW
GL_BUFFER_ACCESS: GL_READ_WRITE
GL_BUFFER_MAPPED: GL_FALSE
GL_BUFFER_SERIALIZED_MODIFY_APPLE: GL_TRUE
GL_BUFFER_FLUSHING_UNMAP_APPLE: GL_TRUE
```

When your application pauses at a breakpoint, you can modify programs and shaders. The modifications take effect when your application resumes, allowing you to get immediate feedback on how changes effect the behavior of your application. For details, see:

- "Modifying Programs" (page 26)
- "Modifying Shaders" (page 26)

You can make modifications to textures and renderbuffers, but these affect only the view in the resource window, allowing for better or correct viewing within OpenGL Profiler. For details, see:

- "Modifying Texture Display Parameters" (page 24)
- "Modifying a Renderbuffer" (page 27)

## Modifying Texture Display Parameters

You can modify how a texture displays by changing its mipmap level, zoom level, source and destination blend modes, and the background color and opacity. You cannot change a texture's target, internal format, source format, source type, or dimensions. Keep in mind that modifications affect only what you see in the Resources window.

To modify the display parameters for a texture:

1. Click Textures in the Resources window.

   The texture names are listed on the left, as shown in Figure 2-4 (page 25).

2. Choose a texture name.

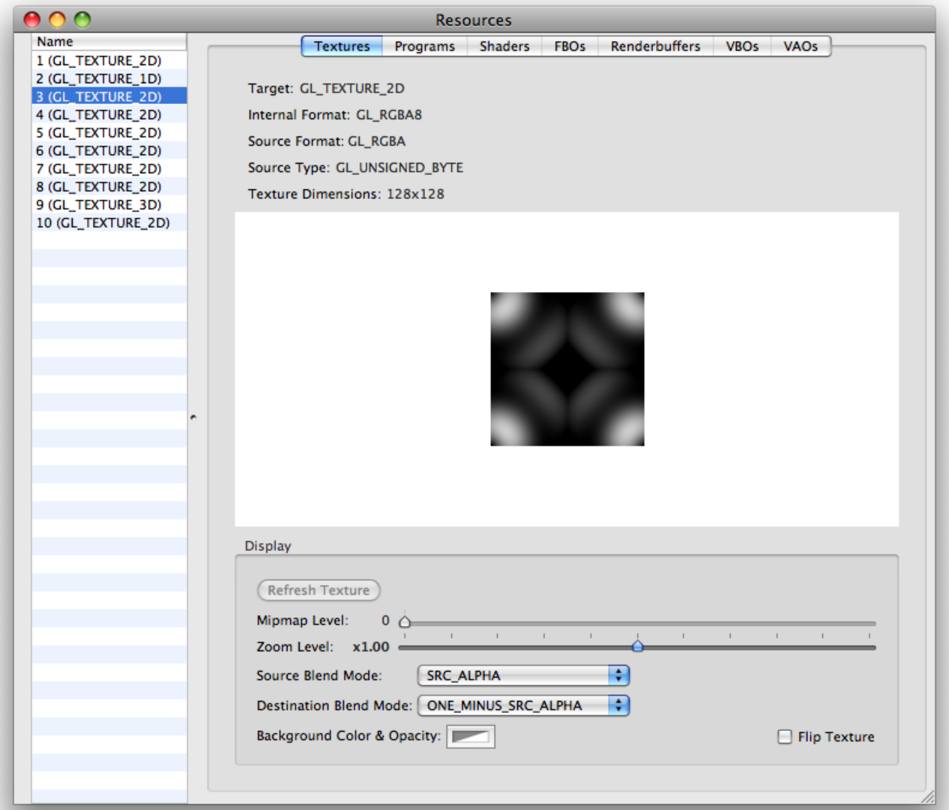
3. Modify the display parameters that you'd like to change.

   You should see the changes take effect immediately.

4. Click Continue in the Breakpoints window.

You need to refresh the texture only if the application modifies it after you open the Resources window.

**Figure 2-4** The Textures pane in the Resources window



Target: GL_TEXTURE_2D
Internal Format: GL_RGBA8
Source Format: GL_RGBA
Source Type: GL_UNSIGNED_BYTE
Texture Dimensions: 128x128

Mipmap Level: 0
Zoom Level: x1.00
Source Blend Mode: SRC_ALPHA
Destination Blend Mode: ONE_MINUS_SRC_ALPHA
Background Color & Opacity:

## Modifying Programs

A program is similar to a shader in that it describes the source code that modifies data in the GPU vertex or fragment pipeline. Programs were available for coding vertex and fragment operations prior to shaders. They use an arcane coding language compared to the C-like syntax used for OpenGL Shading Language. Listing 2-1 (page 26) shows a program for a basic shader.

**Listing 2-1** A program for a basic shader

```
!!ARBvp1.0

ATTRIB vertexPosition  = vertex.position;
OUTPUT outputPosition  = result.position;

# Transform the vertex by the modelview/projection matrix
DP4    outputPosition.x, state.matrix.mvp.row[0], vertexPosition;
DP4    outputPosition.y, state.matrix.mvp.row[1], vertexPosition;
DP4    outputPosition.z, state.matrix.mvp.row[2], vertexPosition;
DP4    outputPosition.w, state.matrix.mvp.row[3], vertexPosition;

# Pass the color and texture coordinate through
MOV    result.color, vertex.color;
MOV    result.texcoord, vertex.texcoord;

END
```

To modify program code:

1. Click Programs in the Resources window.
2. Choose a program from the list on the left.
3. Modify the code.
4. Click Compile.
5. Click Continue in the Breakpoints window.

   When the application resumes, you should see the result of your modified program.

## Modifying Shaders

Shaders are small programs, written using OpenGL Shading Language, that compute surface properties. OpenGL lets you modify shaders that are part of your application, giving you the ability to get immediate feedback on the changes.

To modify a shader:

1. Click Shaders in the Resources window.

   The shader objects are listed on the left, as shown in Figure 2-5 (page 27).
2. Choose a shader object from the list on the left.

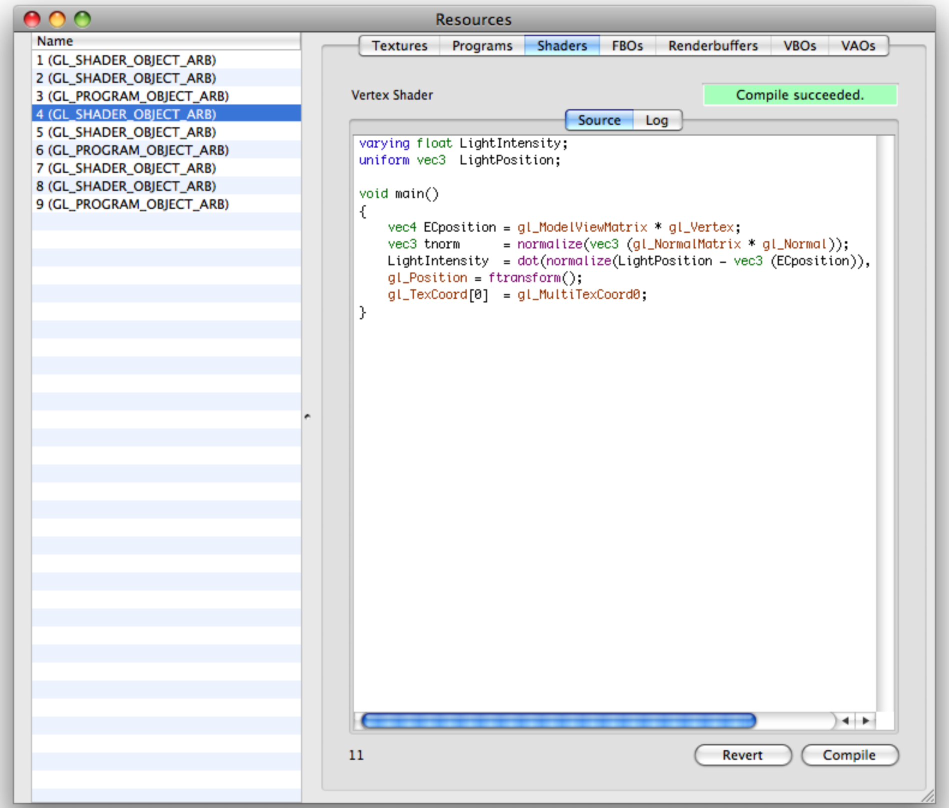
3. Click Source and modify the shader code.

4. Click Compile.

   If compilation fails, click Log to examine the errors. You can return to the original code by clicking Revert.

   If the compilation succeeds, release the breakpoint and click Continue in the breakpoints view.

**Figure 2-5** The Shaders pane in the Resources window



1 (GL_SHADER_OBJECT_ARB)
2 (GL_SHADER_OBJECT_ARB)
3 (GL_PROGRAM_OBJECT_ARB)
4 (GL_SHADER_OBJECT_ARB)
5 (GL_SHADER_OBJECT_ARB)
6 (GL_PROGRAM_OBJECT_ARB)
7 (GL_SHADER_OBJECT_ARB)
8 (GL_SHADER_OBJECT_ARB)
9 (GL_PROGRAM_OBJECT_ARB)

```
varying float LightIntensity;
uniform vec3  LightPosition;

void main()
{
    vec4 ECposition = gl_ModelViewMatrix * gl_Vertex;
    vec3 tnorm      = normalize(vec3 (gl_NormalMatrix * gl_Normal));
    LightIntensity  = dot(normalize(LightPosition - vec3 (ECposition)),
    gl_Position = ftransform();
    gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

## Modifying a Renderbuffer

A renderbuffer is an offscreen rendering destination for a 2D pixel image. Renderbuffers are defined in the OpenGL specification for the GL_EXT_framebuffer_object extension. OpenGL Profiler lets you modify how the contents of the renderbuffer display by changing its zoom level, source and destination blend modes, and the background color and opacity. You cannot change a renderbuffer's target, source format, source type, or image dimensions. Keep in mind that modifications affect only what you see in the Resources window.

To modify the display parameters of a renderbuffer object:

1. Click Renderbuffers in the Resources window.

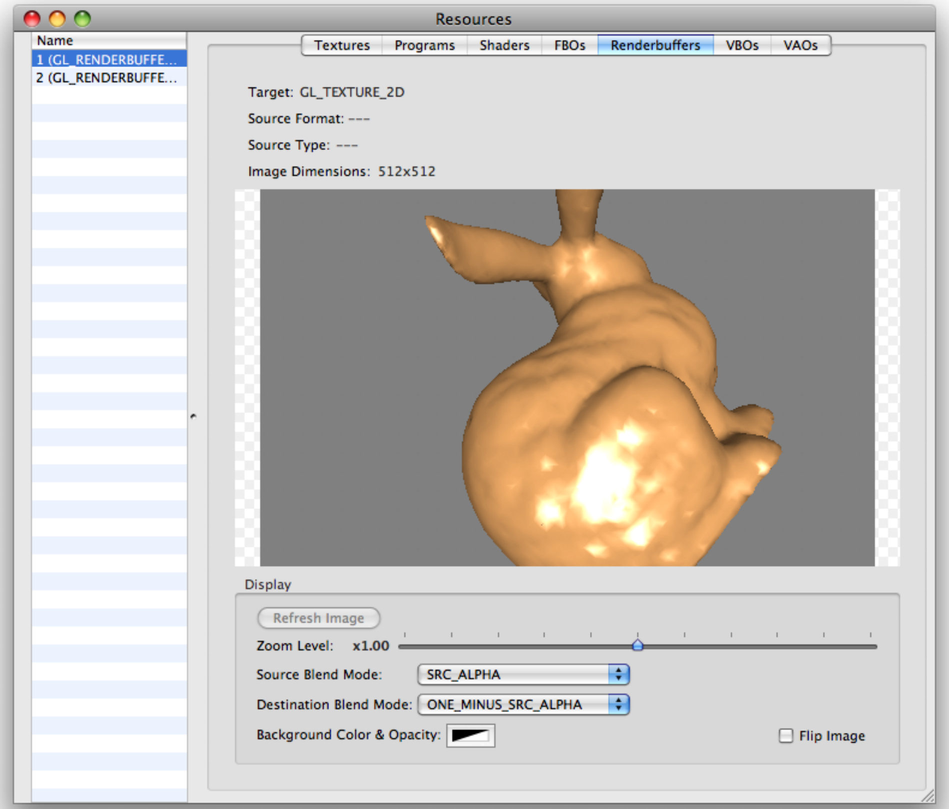
   The renderbuffer objects are listed on the left, as shown in Figure 2-6 (page 28).

2. Choose a renderbuffer object.
3. Modify the display parameters that you'd like to change.

   You should see the changes take effect immediately.
4. Click Continue in the Breakpoints window.

You need to refresh the image only if the application modifies it after you open the Resources window.

**Figure 2-6** The Renderbuffers pane in the Resources window

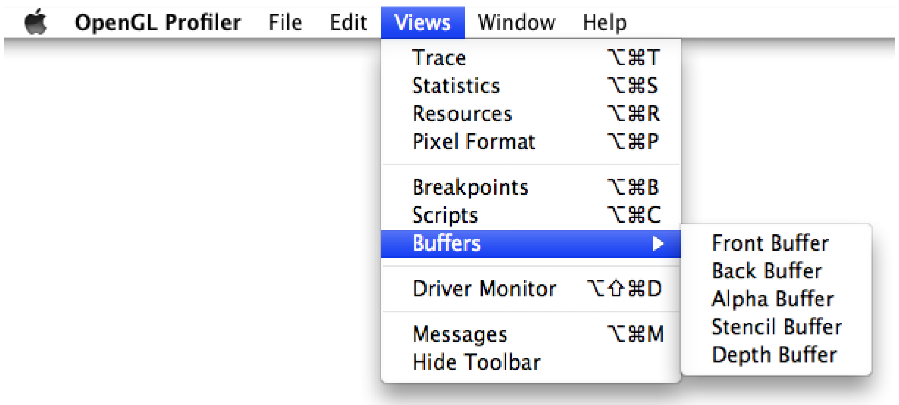


Target: GL_TEXTURE_2D
Source Format: ---
Source Type: ---
Image Dimensions: 512x512

Zoom Level: x1.00
Source Blend Mode: SRC_ALPHA
Destination Blend Mode: ONE_MINUS_SRC_ALPHA
Background Color & Opacity:

## Viewing Buffers

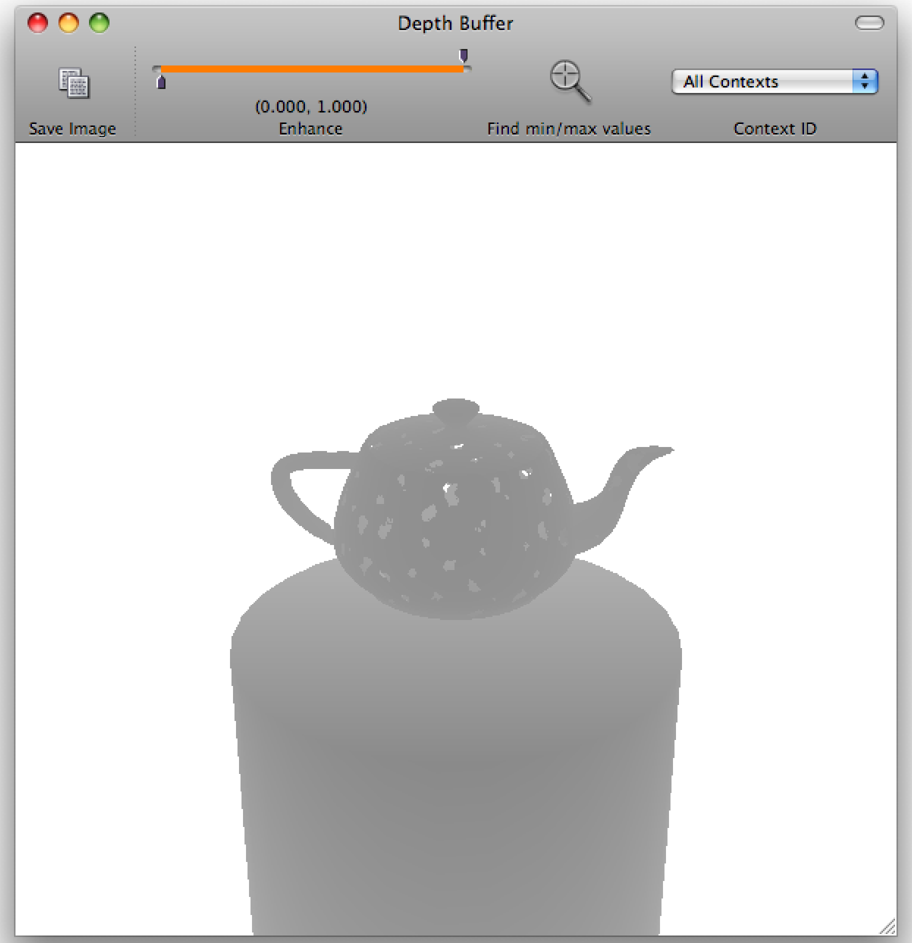
You can view the contents of an OpenGL buffer by choosing Views > Buffers and then choosing a buffer—Front Buffer, Back Buffer, Alpha Buffer, Stencil Buffer, or Depth Buffer. Buffers not used by your application appear dimmed. Buffer views are available only when your application pauses at a breakpoint.



After selecting a buffer, its contents appear in a window. Each time your application pauses at a breakpoint, each buffer view updates automatically.

Figure 2-7 (page 30) shows the depth buffer contents for the GLSLShowpiece application available in `/Developer/Examples/OpenGL/Cocoa`.

**Figure 2-7** The Depth Buffer window



## Creating and Attaching Scripts to a Breakpoint

OpenGL Profiler allows you to execute (at a performance cost) small amounts of OpenGL code or other scripting calls when your application pauses at a breakpoint. A script is a text file that contains OpenGL function calls. For example, a script can be as simple as the following line of code:

```
glClear(1.0, 0.0, 0.0, 1.0);
```

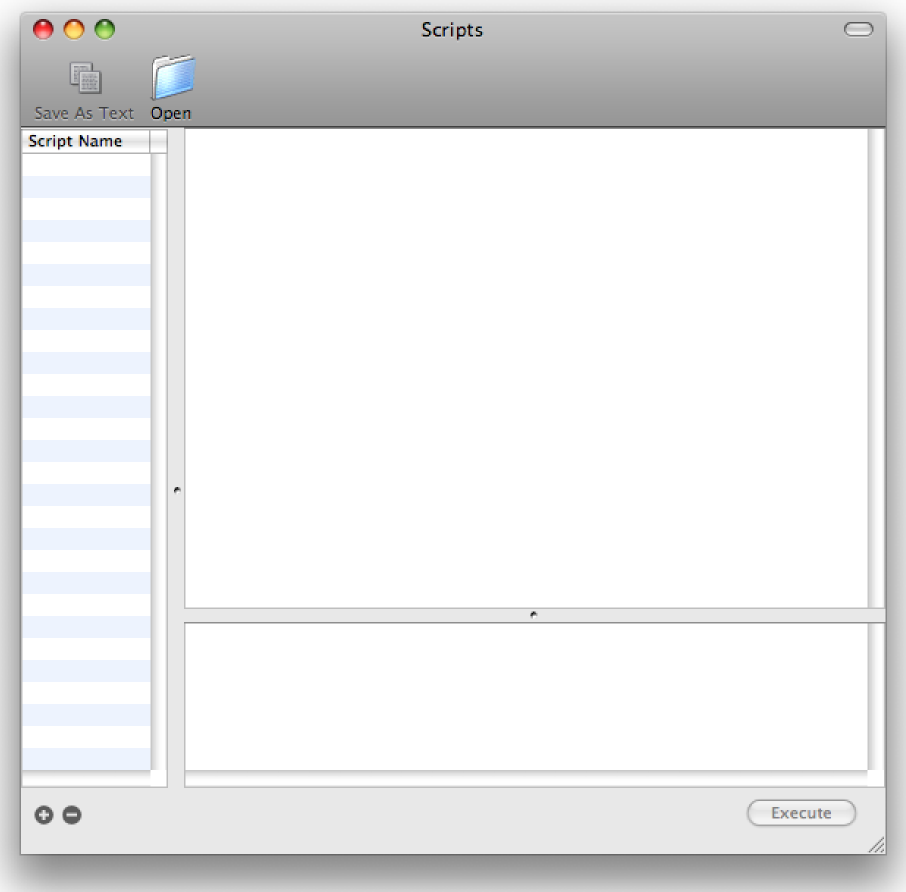
Or it can be more complex, containing many lines of code, such as:

```
glBegin(GL_POINTS);
glVertex3f(0,0,1);
glVertex3f(1,0,1);
glVertex3f(1,1,1);
glVertex3f(0,1,1);
glEnd();
```

You can create a script using any text editor, or you can enter a script directly into the Scripts window.

To add a script:

1. Choose Views > Scripts to open the Scripts window.



2. If you created the script in a text editor, click Open, navigate to the script, and choose it. If you want to create a script in the Scripts window, click the plus (+) button, enter a script name, then type commands in the text window on the right.

   Any script that you add in the Scripts window will be available from the Breakpoints window.

After you add a script to OpenGL Profiler, you can either execute it manually or attach it to a breakpoint.

To execute a script manually when your application pauses at a breakpoint:

1. Choose a script in the Scripts window.

2. Click Execute.

   OpenGL Profiler sends the commands to your application for execution. Errors in your script appear in the pane located in the lower left of the Scripts window.

To attach a script to a breakpoint:

1. Open the Breakpoints window.
2. Choose Actions > Attach Script.
3. In the sheet that appears, choose a script.
4. Select one or more execution options.

   You can choose to have OpenGL Profiler execute the script before, after, or both before and after the OpenGL function. You can also choose to have your application continue after executing the script.

5. Click Attach.

   A small document icon appears in the functions list next to the blue breakpoint indicator for the function.

**Note:** Attaching a script to a function that your application uses repeatedly is an expensive operation. Data that you look at when the script executes is not an accurate measure of performance.

To remove a script from a breakpoint, choose Actions > Remove Script.

## Disabling OpenGL Execution

Under some circumstances you might want to disable the execution of one or more OpenGL functions in your application.

To disable the execution of all OpenGL functions in your application, in the Breakpoints window choose Actions > Execute none. To enable execution, choose Actions > Execute all.

To disable execution of individual OpenGL functions, in the Breakpoints window, select the function you want to disable, and click the Execute column. OpenGL Profiler ignores breakpoints set on disabled functions. To enable a function, click the Execute column so that the blue indicator appears.

**Note:** You cannot disable functions that have a dimmed indicator in the Execute column. These functions manage the context.

# Identifying and Solving Performance Issues

This chapter describes a number of strategies that can help you track down performance issues and understand how OpenGL works with your application. Before you read this chapter, you should already be familiar with how to start and run a profiling session, and how to set breakpoints.

## Making Sure You Use Functions Correctly

OpenGL is an evolving specification. As time goes on, programming practices that were acceptable in the past are replaced by techniques that work much better. There are several functions in the OpenGL specification that you should watch for when you profile your application. If you are using any of these OpenGL functions in your application, make sure that you really need to use them, and that you are using them correctly.

- Avoid `glBegin` because it signals that you are using immediate mode. Unless you are drawing a simple shape or creating a small prototype, immediate mode is typically not an optimal technique for using OpenGL.
- `glFinish` forces your application to wait until all OpenGL commands in the pipeline have executed on the graphics card.
- `glFlush` is typically not needed because flushing is usually handled by other calls, such as `CGLFlushDrawable`. The function `glFlush` flushes at the macro level, an expensive operation. Flushing calls provided by the Mac OS X interfaces (AGL, CGL, Cocoa OpenGL classes) act at a microlevel to give finer control over flushing and, as a result, much better performance.
- `glTexImage` and related calls. Check to see if you are using this function to redefine the texture each frame. If so, change your code to define the texture outside frame rendering. If your data changes, you can instead use a data replacement technique, such as pixel buffer objects, or call `glTexSubImage` to specify the image again. Keep in mind that you can call `glTexSubImage` to specify the entire image again (maintaining the current texture parameters), and it will be a less expensive call than `glTexImage`. You should call `glTexImage` only if you must specify a larger image.
- Any form of `glGet` or `glIs`. Getting state values slows your application. Unless your application is a "middle ware" application, you shouldn't need to retrieve state values. During development, however, it's quite common to call `glGetError`. When your application is ready to go into production, make sure that your remove `glGetError` calls and any other state getting and checking functions. As an alternative during development, you can look for errors by setting OpenGL Profiler to break on errors.
- `glPushAttrib` or `glPopAttrib`. You should keep track of OpenGL state instead of using the server attribute stack.
- The function `glDrawPixels` is a convenience function that, under the hood, uses screen-aligned textured quads. If you use screen-aligned textured quads directly, you'll save the overhead of calling `glDrawPixels`. Make sure that when you use textured quads that you also use appropriate texture filtering.
- If you are calling `glReadPixels`, you should also be using pixel buffer objects.

- `glProgramLocalParameter` and `glProgramEnvParameter` calls. These calls, defined by the GL_ARB_vertex_program extension, load only one parameter at a time. It's more efficient to use `glProgramLocalParameters` and `glProgramEnvParameters`, defined by the GL_EXT_gpu_program_parameters extension, which loads multiple parameters.

## Identifying Problem Areas in Your Application

You can get a quick view of what your application is doing by collecting a trace for a single frame. Although viewing the trace and statistics for one frame provides a narrow view of application behavior, you can use this strategy to narrow down problem areas in your application.

To collect a trace for a single frame:

1. Launch or attach to the application of interest.
2. Open the Statistics window.
3. Navigate in the application to the area where you suspect a problem.
4. Set a breakpoint on the function `CGLFlushDrawable`. If you are using a single-buffered rendering context, you might also need to set a breakpoint on the function `glFlush`.
5. When your application pauses, click Clear in the Statistics window.
6. Click Continue in the Breakpoints window. When your application pauses, make sure it rendered the frame completely. If not, your code likely calls more than one flush operation per frame.

   Then, check for the following in the Statistics window:

   - State management. Check to see if you are calling `glPopAttrib` calls. If possible you should instead track your own state. It's not a good idea to set state on a per frame basis. It's best to consolidate state changes and set them outside the frame.
   - Calls that take significantly more time than the others.
   - Any OpenGL calls that are listed in "Making Sure You Use Functions Correctly" (page 33).
7. Click the Continue button in the Breakpoints window to resume execution of your application.

## Checking for Optimal Data Types and Formats

To get the best performance, make sure your data is using an optimal data type and format combination. You won't get the best performance otherwise.

Ideally, you'll want to use:

`GL_BGRA`, `GL_UNSIGNED_INT_8_8_8_8_REV`

This is the fastest data type and format combination. If it isn't fast, you may have a problem with your driver.

If that's not possible, you can often get acceptable performance from the following. Just make sure to test these combinations on a device-by-device basis:

`GL_BGRA`, `GL_UNSIGNED_SHORT_1_5_5_5_REV`

`GL_YCBR_422_APPLE`, `GL_UNSIGNED_SHORT_8_8_REV_APPLE`

OpenGL Profiler does not know what the inbound data format is. So you need to check the point at which your data gets uploaded by OpenGL by performing this steps:

1. Select Collect Trace and Include Backtraces in the OpenGL Profiler window.
2. Set a breakpoint where your data is uploaded.
3. When your application breaks, click Call Stack in the Trace window.

## Managing Trace Data

The amount of data generated when collecting a trace can be overwhelming. Most of the time you'll collect a trace for only one frame, which is typically enough to track down the most common issues. (See "Identifying Problem Areas in Your Application" (page 34).) If you do need to collect more than a frame of data, you can create a custom shell script that operates on trace data so that you can get it into a more manageable state. When you want to apply a script, click the Filter button in the Trace window. OpenGL Profiler provides the trace data as input (`stdin`) to your script and writes the results from your scripting calls to `stdout`.

If a drawing call is slow and you suspect a shader is the cause, you may need to collect more than frame of data. You can then use a script to pare down the data to suspicious calls. Listing 3-1 (page 35) shows a very simple script that sorts calls by function call time, with the slowest calls last. Your script would need to be customized so that it performs operations appropriate for the problem you are trying to isolate. After you use a script to identify suspicious calls, you can use the line number of the output to trace back to where the call actually took place.

**Listing 3-1** A shell script for finding the slowest calls in a trace

```
#!/bin/tcsh -f
awk '{ print $3" "$0 }' | sort -n
```

To create a script:

1. Open an application, such as Xcode, that lets you create a plain text file.
2. Enter the appropriate script commands using any scripting language that accepts input from `stdin`.
3. Save the script with a `.filter` extension.

   You can save it to any location that you'd like.
4. For the first script that you create, select it in the Finder and choose File > Get Info.

   Make sure that the "Open with" pop-up menu is set to the application that you used for creating the script. This ensures that OpenGL Profiler will use this application to open any files with the `.filter` extension.

To filter trace data:

1. Open the Trace window.
2. Click Browse, navigate to the script you want to use, and select it.

   You can modify an existing script by clicking Open.
3. Click Filter, then provide a name for the output file.

## Checking for Application Errors, Thread Conflicts, and Software Fallbacks

You can quickly check for errors in your application by setting one or more error breakpoints. To use OpenGL Profiler to check for application errors:

1. Choose View > Breakpoints.
2. Select one or more of these options:
   - Break on error. Your application pauses when it encounters any type of error.
   - Break on VAR error. Your application pauses when there is a problem using the vertex array range extension.
   - Break on thread conflict. You can select this if your application is multithreaded.
   - Break on SW fallback. You application pauses when it uses the software renderer as a fallback. Although this condition is not strictly an error, it alerts you to situations for which the system does not have the appropriate hardware renderer to carry out a particular OpenGL call.
3. If you have not already done so, launch or attach your application.
4. Monitor the Breakpoints window for errors.

## Evaluating The Effect Of The Multithreaded OpenGL Engine

In Mac OS X v10.5 and later, the OpenGL framework can offload processing onto a separate thread that runs on a different CPU core. You use the `CGLEnable` function to enable multithreaded execution programmatically using this code:

```
#include <OpenGL/OpenGL.h>

CGLError err = 0;
CGLContextObj ctx = CGLGetCurrentContext();

// Enable the multi-threading
err =  CGLEnable( ctx, kCGLCEMPEngine);
```

```
if (err != kCGLNoError )
{
     // Multi-threaded execution is possibly not available
     // Insert your code to take appropriate action
}
```

For more details see Technical Note TN2085: *Enabling multi-threaded execution of the OpenGL framework*.

After enabling multithreading, some applications see a dramatic increase in OpenGL performance; others might not. In general, the multithreaded OpenGL engine is a good option for applications that are CPU bound.

If your application enables the multithreaded OpenGL engine, it's a good idea to check whether it actually improves performance. After you programmatically enable multithreading, evaluate its effect by following these steps:

1. Launch or attach to the application of interest.
2. In the main OpenGL Profiler window, check the frame rate at a point in your application that is repeatable. You'll check this same point later.
3. Choose Views > Breakpoints.
4. Make sure the multithreaded control option is set to "App control."
5. In the main OpenGL Profiler window, check the frame rate.
6. Set a breakpoint on a function.

   Although you can choose any function, typically you'd set a breakpoint on `CGLFlushDrawable` for a double-buffered rendering context or `glFlush` for a single-buffered rendering context.
7. When your application pauses, select the "Force off" option for multithreaded control.
8. Press Continue to resume execution of your application.
9. In the main OpenGL Profiler window, check the frame rate.

   Compare this frame rate to the rate you observed when using the multithreaded OpenGL engine.

**Note:** If you are using the multithreaded OpenGL engine, debugging is often easier after you use the option in OpenGL Profiler to turn off multithreading.

## Monitoring GPU Use

Applications that can't use the GPU for some reason (such as the graphics card does not support some of the OpenGL extensions that your code uses) use the software renderer as a fallback. If you notice a drop in the performance of your application, you may want to check whether the application is using the GPU as you expect.

Starting in Mac OS X v10.5, you can set your application to break whenever it uses the software renderer as a fallback. (See "Checking for Application Errors, Thread Conflicts, and Software Fallbacks" (page 36).) Prior to Mac OS X v10.5, you can monitor the GPU use of your application whenever your application pauses at a breakpoint. The best breakpoints to check are:

- `CGLFlushDrawable`
- After `glUseProgramObjectARB`
- After `glBindProgramARB`
- Any `glDraw` or related command, including the following:

```
glAccum
glBegin
glBlitFramebufferEXT
glBitmap
glClear
glCopyPixels
glDrawPixels
glReadPixels
glCopyTexSubImage*D
glCopyTexImage*D
glDrawArrays
glDrawElements
glDrawRangeElements
```

When your application pauses at these breakpoints, check the values of `kCGLCPGPUFragmentProcessing` and `kCGLCPGPUVertexProcessing` shown in the Call Stack pane of the Breakpoints window. A value of `GL_TRUE` indicates that your application is using the GPU for the process associated with the constant.

## Using Window Resizing to Diagnose Performance

If your application renders to a window, you can often identify the cause of performance issues by resizing the window. While your application renders to a window, resize it. After you shrink the window, if the execution time is significantly faster, then the issue might be related to low VRAM. If the execution time is faster proportional to the window size, then your application is fragment bound. If the execution time is the same, then your application either is vertex bound on the GPU or is CPU bound.

## Identifying Unmatched Calls

Many calls in OpenGL are used as sets, such as:

- `glBegin` and `glEnd`
- `glEnable` and `glDisable`

You can check the # of Calls column in the Statistics window to make sure that these (and similar sets) are always matched in your application. For example, if you find 5 `glBegin` calls but only 3 `glEnd` call, you should modify your code so that you have the same number of each. Unmatched calls typically are a symptom of unneeded code and always indicate imprecise code. Call sets should always match within a frame.

# Controlling Profiling Programmatically

You can add code to your application that allows it to interact with OpenGL Profiler during a profiling session. This chapter shows you how to perform these tasks programmatically:

- "Setting a Breakpoint" (page 41)
- "Writing Comments to the Trace Window" (page 42)
- "Controlling Trace Collection" (page 42)
- "Controlling Statistics Collection" (page 43)

## Setting a Breakpoint

Your application can programmatically set breakpoints when it is attached to OpenGL Profiler.

To set a breakpoint:

1. Include the `CGLProfiler.h` and `CGLProfilerFunctionEnum.h` header files in your application.

2. Declare an array of three `GLint` values, set to the following:

   - The function ID, as defined in the header file `CGLProfilerFunctionEnum.h`.
   - The logical `OR` of `kCGLProfBreakBefore` or `kCGLProfBreakAfter`, indicating how you want the breakpoint to stop—before entering the OpenGL function, on return from it, or both.
   - A Boolean that turns the breakpoint on or off.

3. Call the function `CGLSetOption`, passing the array as a parameter.

Listing 4-1 (page 41) shows code that sets a breakpoint before the `CGLFlushDrawable` function.

**Listing 4-1** Code that sets a breakpoint

```
#include "OpenGL/CGLProfiler.h"
#include "OpenGL/CGLProfilerFunctionEnum.h"
...
   GLint myBreakpoint[] = { kCGLFECGLFlushDrawable, kCGLProfBreakBefore, 1;}
   CGLSetOption( kCGLGOEnableBreakpoint, myBreakpoint );
...
```

## Writing Comments to the Trace Window

Your application can programmatically write comments to the Trace window during a profiling session. To write comments:

1. Include the `CGLProfiler.h` header file in your application.
2. Call the function `CGLSetOption` with the constant `kCGLGOComment` and your comment cast as a long.

Listing 4-2 (page 42) shows code that writes a comment that looks like this in the Trace window:

```
21561: 0.00 µs /* ***** My Comment is here ***** */
```

**Listing 4-2** Code that writes a comment to the Trace window

```
#include <OpenGL/CGLProfiler.h>
...
  CGLSetOption(kCGLGOComment, (long) "***** My Comment is here *****");
...
```

## Controlling Trace Collection

Your application can programmatically control when to start and stop collecting a trace, which lets you control which traces to collect in a specific part of your application or during a particular period of time. You can also clear the Trace window.

To control trace collection:

1. Include the `CGLProfiler.h` header file in your application.
2. Call the function `CGLSetOption` with the constant `kCGLGOEnableFunctionTrace` and either `GL_TRUE` (to turn on trace collection) or `GL_FALSE` (to turn off trace collection).

Listing 4-3 (page 42) shows code that enables trace collection.

**Listing 4-3** Code that enables trace collection

```
#include <OpenGL/CGLProfiler.h>
...
  CGLSetOption(kCGLGOEnableFunctionTrace, GL_TRUE);
...
```

To clear the Trace window:

1. Include the `CGLProfiler.h` header file in your application.
2. Call the function `CGLSetOption` with the constant `kCGLGOResetFunctionTrace` and the value `NULL`.

Listing 4-4 (page 43) shows code that enables trace collection.

**Listing 4-4** Code that clears the Trace window

```
#include <OpenGL/CGLProfiler.h>
...
  CGLSetOption(kCGLGOResetFunctionTrace, NULL);
...
```

## Controlling Statistics Collection

You application can programmatically control when to start and stop collecting statistics. You must make sure that the Statistics window in OpenGL Profiler is open when you profile your application.

To control statistics collection:

1. Include the `CGLProfiler.h` header file in your application.
2. Call the function `CGLSetOption` with the constant `kCGLGOResetFunctionStatistics` and the value `NULL` to first reset counters to 0. This step is optional.
3. Call the function `CGLSetOption` with the constant `kCGLGOResetFunctionStatistics` and the value `GL_TRUE` to start statistics collection.
4. When you are done collecting statistics, call the function `CGLSetOption` with the constant `kCGLGOResetFunctionStatistics` and the value `GL_FALSE`.

Listing 4-5 (page 43) shows code that resets counters, starts statistics collection, and then stops it.

**Listing 4-5** Code that starts and stops statistics collection

```
#include <OpenGL/CGLProfiler.h>
...
  // Reset counters to 0
  CGLSetOption(kCGLGOResetFunctionStatistics, NULL);
  // Start statistics collection
  CGLSetOption(kCGLGOEnableFunctionStatistics, GL_TRUE);
...
  // Stop statistics collection
  CGLSetOption(kCGLGOEnableFunctionStatistics, GL_FALSE);
...
```

# Document Revision History

This table describes the changes to *OpenGL Profiler User Guide*.

| Date | Notes |
|---|---|
| 2008-02-08 | Fixed a link. |
| 2007-12-11 | New document that explains how to assess the efficiency of an OpenGL application. |