AN OPENGL AND OPENGL ES DEBUGGER AND PROFILER

gDEBugger

REALIZE
YOUR
OGL
POTENTIAL

YOUR GRAPHIC APPLICATIONS PERFORMANCE TUNING GUIDE

ANALYZE AND OPTIMIZE THE PERFORMANCE AND RELIABILITY OF YOUR OGL BASED APPLICATIONS WITH GDEBUGGER

www.gremedy.com
info@gremedy.com

graphicREMEDY
SOFTWARE SOLUTIONS FOR THE 3D GRAPHICS INDUSTRY

THIS GUIDE HELPS YOU TO:

SAVE TIME BY TAKING THE GUESS WORK OUT OF PROFILING

Know how the graphic pipeline operates and be aware of the possible performance bottlenecks.

BUILD A BETTER PRODUCT. GAIN BETTER PRODUCT PERFORMANCE

gDEBugger provides you with the precise information you need to find performance bottlenecks and optimize application performance. No more 'performance killer' or redundant OpenGL calls.

BECOME A BETTER OGL DEVELOPER WITH GDEBUGGER

See how changes you make affect OpenGL usage, the application's visual display, performance and accuracy.

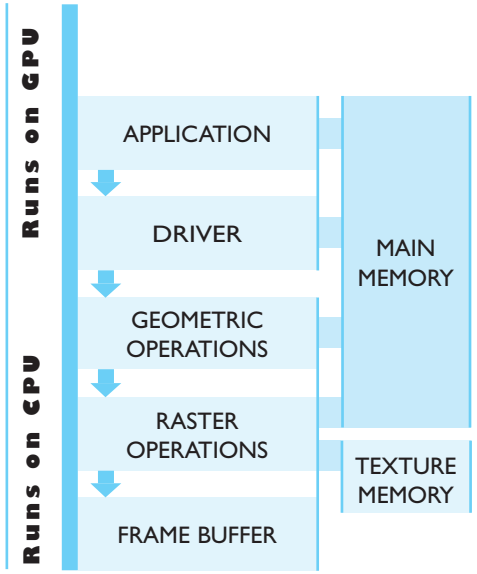
GRAPHIC PIPELINE OVERVIEW, GET FAMILIAR WITH YOUR HARDWARE

1 Graphic Pipeline Bottlenecks

The graphics system generates images through a pipelined sequence of operations. A pipeline runs only as fast as its slowest stage. The slowest stage is often called the pipeline bottleneck. A single graphic primitive (for example, a triangle) has a single graphic pipeline bottleneck. However, the bottleneck may change when rendering a graphic frame that contains multiple primitives. For example, if the application first renders a group of lines and afterwards a group of lit and shaded triangles, we can expect the bottleneck to change.

2 The OpenGL Pipeline

The OpenGL pipeline is an abstraction of the graphics system pipeline. Here is a very rough sketch of the OpenGL pipeline:



OPENGL GRAPHIC PIPELINE (ROUGH SKETCH)

Some of the pipeline stages are executed on the CPU; other stages are executed on the GPU. Most operations that are executed on top of the GPU are executed in parallel.

Application – the graphical application executed on the CPU and calls OpenGL API functions.

Driver – the graphics system driver runs on the CPU and translates OpenGL API calls into actions executed on either the CPU or the GPU.

Geometric operations – the operations required to calculate vertices attributes and position within the rendered 2D image space. This includes: multiplying vertices by the model view and projection matrices, calculating vertices lighting values, executing vertex shaders, etc.

Raster operations – operations operating on fragments / screen pixels: reading and writing color components, reading and writing depth and stencil buffers, performing alpha blending, using textures, executing fragment shaders, etc.

Frame buffer – a memory area holding the rendered 2D image.

OPTIMIZING GRAPHICAL APPLICATIONS PERFORMANCE USING GDEBUGGER

Step 1

"Clean" your OpenGL usage

The first step that should be done before optimizing your application performance is to "clean" your application's OpenGL usage: make sure that your application uses OpenGL correctly and performs the OpenGL API calls you expect it to perform.

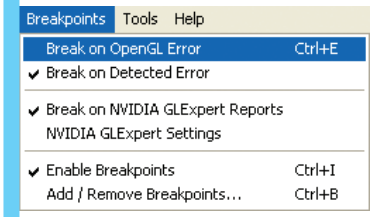
a. Remove OpenGL errors:

When an OpenGL error occurs, in most cases, OpenGL ignores the API call that generated the error. This means that when your application generates OpenGL errors, OpenGL ignores actions that you would like it to perform. These actions may have significant impact on render performance. Therefore, it is important to remove all OpenGL errors before starting to optimize your code.

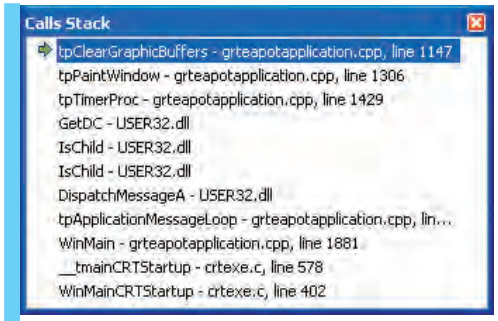
gDEBugger offers two mechanism for doing that:

1. BREAK ON OPENGL ERRORS: Breaks the application run whenever an OpenGL error occurs.

2. BREAK ON DETECTED ERRORS: Let gDEBugger perform error tests that OpenGL drivers cannot afford to perform (performance wise). This functionality turns on the gDEBugger detected error mechanism, breaking the application run whenever a detected error occurs.



After the application run was suspended, you can use the Call Stack View to view the call stack and source code that led to the OpenGL or Detected error.



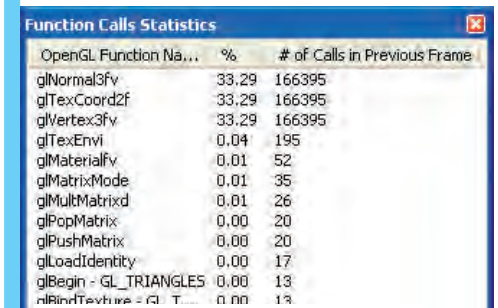
b. Remove redundant OpenGL calls:

Most OpenGL based applications generate a lot of redundant OpenGL API calls. Some of these calls may have significant impact on render performance. It is important to view the log of the OpenGL calls that your application generates and remove the redundant API calls that seems to have significant impact on render performance. Such redundant API calls may be: redundant state changes, repeatedly turning on and off the same OpenGL mechanisms, using immediate mode rendering, etc.

gDEBugger offers few mechanisms for doing that:

1. OPENGL FUNCTION CALLS HISTORY VIEW: Displays a log of OpenGL, OpenGL ES, extensions, WGL and EGL function calls executed in each render context.

2. OPENGL FUNCTION CALLS STATISTICS VIEW: Allows viewing the number of times each OpenGL function call was executed in the previously rendered frame and it's percentage from the total functions execution.



Step 2

Remove performance bottlenecks

As mentioned in the "Graphic Pipeline Bottlenecks" section, the graphics system generates images through a pipelined sequence of operations. The pipeline runs only as fast as its slowest stage, which is often called "the pipeline bottleneck".

The process for removing performance bottlenecks contains the following stages:
1. IDENTIFYING THE BOTTLENECK – Locate the pipeline state that is the current graphic pipeline bottleneck.
2. OPTIMIZING – Reduce the workload done in that pipeline stage until performance stops improving or you have achieved the desired performance level.

* Steps 1 and 2 should be repeated until the desired performance level is reached.

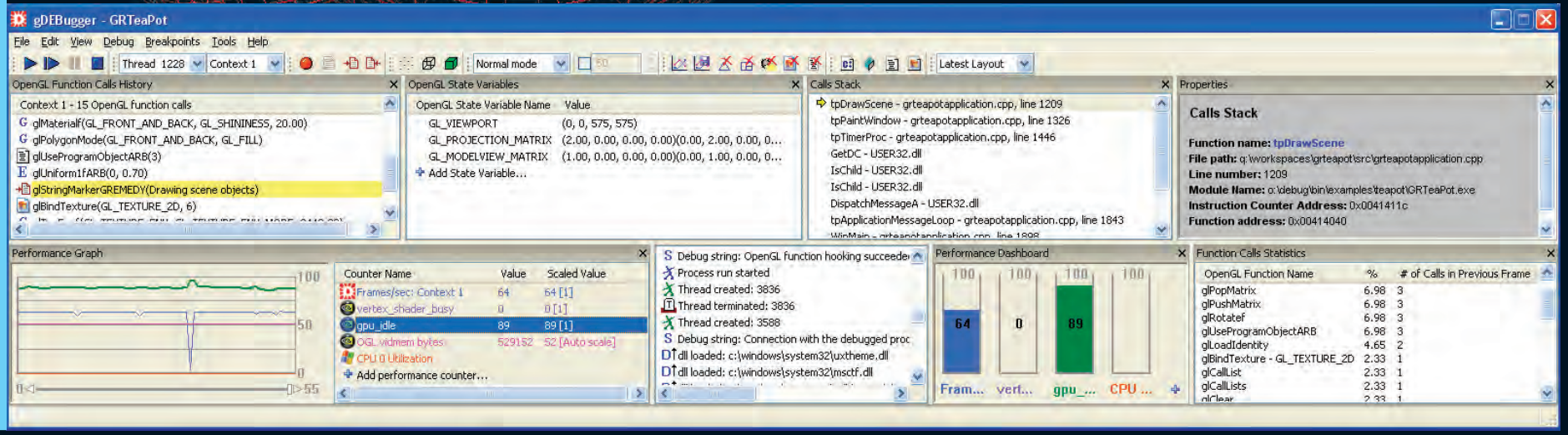
Notice that after your performance optimizations are done, or you have reached a bottleneck that you cannot optimize anymore, you can start adding workload to pipeline stages that are not fully utilized without affecting render performance (example: use more accurate textures, perform more complicated vertex shader operations, etc).

gDEBugger offers views and toolbars that helps you locate your application's graphic pipeline performance bottlenecks:

a. Performance Graph view:

gDEBugger Performance Graph view displays, in real time, graphics system performance metrics such as Frame per Second, number of OpenGL function calls per frame, vertex and fragment processors utilization, video memory usage, GPU idle, Driver idle, textures data utilization and amount of texels currently loaded into the graphic pipeline. There is no need to make any changes to your source code or recompile your application. The performance counters will be displayed inside the Performance Graph view.

gDEBugger supports NVIDIA's performance counters via NVPerfKit, ATI's Performance metrics and 3Dlabs performance counters via 3DLabs' Hardware Profiling Technology.



b. Performance Analysis toolbar:

The Performance Analysis toolbar offers commands that enable you to pinpoint application performance bottlenecks. These commands let you disable graphics pipeline stages one by one. If the performance metrics improves while turning off a certain stage, you have found a graphics pipeline bottleneck!

These commands include:

Eliminate all draw commands - Identify CPU / BUS performance bottlenecks by ignoring all OpenGL draw commands: all OpenGL commands that push vertices or texture data into OpenGL. When ignoring these commands, the CPU and bus workloads remain unchanged, but the GPU workload is almost totally removed since most GPU activities are triggered by input primitives (triangles, lines, etc).

Eliminate all raster operations - Identify raster operations bottlenecks by forcing the use of a 1x1 pixels view port. Raster operations operate per fragment or pixel. By setting a 1x1 pixels view port most raster operations are eliminated.

Eliminate all fixed pipeline light operations - Identify "fixed pipeline lights" related calculations bottlenecks. This is done by turning off all OpenGL fixed pipeline lights. Notice that this command does not affect fragment shaders that do not use the fixed pipeline lights.

Eliminate all textures data fetch operations - Identify textures memory performance bottlenecks by forcing the application to use 2x2 stub textures instead of the application defined textures. When using such small stub textures, the texture data fetch operations workload is almost completely removed.

Eliminate OpenGL fragment shaders operations - Identify fragment shaders related bottlenecks by forcing a simple stub fragment shader instead of the application defined fragment shaders.

OTHER GDEBUGGER FEATURES

Below is a partial list of other gDEBugger features that were not mentioned in the above sections:

- Launches any OpenGL or OpenGL ES application for debug or profile session.
- Adds breakpoints for any OpenGL, OpenGL ES or extensions entry point.
- Views texture objects, their parameters and the textures' data as an image.
- Saves textures as image files to disk.
- Views program and shader parameters, active uniforms values and shader source code.
- Edits, Saves and Compiles Shaders, Links and Validates Shading Programs "on the fly".
- Views a list of active and deleted OpenGL render contexts.
- Detects OpenGL errors and automatically suspends the application run.
- Views the application's threads call stack and source code.
- Views OpenGL state variables values in the watch view.
- Saves a snapshot of all the OpenGL state variables into a file.
- Compares the current OpenGL state machine values to a saved snapshot automatically.
- Forces OpenGL to render directly into the front buffer and controls the rendering speed.
- Forces the OpenGL Polygon Raster mode to see the rendered geometry.
- Saves performance counters data into a file (.csv). This enables you to do performance and regression tests for your application using different hardware and driver configurations.
- Supports applications that render using multiple threads and multiple render contexts.
- Supports debugging and profiling of OpenGL ES applications.
- gDEBugger ES is being used as an "out of the box" OpenGL ES emulator for the embedded systems on a Windows PC machine.
- Displays implementation-specific OpenGL run-time information such as pixel formats and available extensions.
- Displays NVIDIA GLExpert driver reports, and automatically suspends the application run.
- And more…

gDEBugger works with all current graphic hardware products. It supports NVIDIA GPUs performance counters via NVPerfKit, NVIDIA GLExpert driver reports, ATI Performance Metrics, the latest version of OpenGL and many additional extensions.