User Guide

NVPerfKit

NVIDIA Performance Toolkit

Table of Contents

Introduction

Please read this entire document before you get started! Important issues are covered in this document that will help get things running smoothly.

NVPerfKit gives every graphics application developer access to low-level performance counters inside the driver and hardware counters inside the GPU itself. The performance counters are available in PerfMon through the Windows Management Instrumentation (WMI) Performance Data Helper (PDH) interface and in Microsoft PIX for Windows via the PIX for Windows NVIDIA Plug-in.

The counters can be used to determine exactly how your application is using the GPU, identify performance issues, and confirm that performance problems have been resolved. Now, for the first time ever, this confidential information is available to third party developers.

NVPerfKit consists of the following components:

- Instrumented display driver
- PIX for Windows NVIDIA Plug-in
- NVDevCPL Control Panel applet
- Sample code and helper classes

Appendix A contains a breakdown of the files that are installed on your system.

System Requirements

- NVIDIA instrumented display driver, version 77.40 or later on Windows XP
- NVPerfKit signals are available on all NVIDIA GPUs listed below:
  - GeForce 6800 Ultra
  - GeForce 6800 GT
  - GeForce 6600

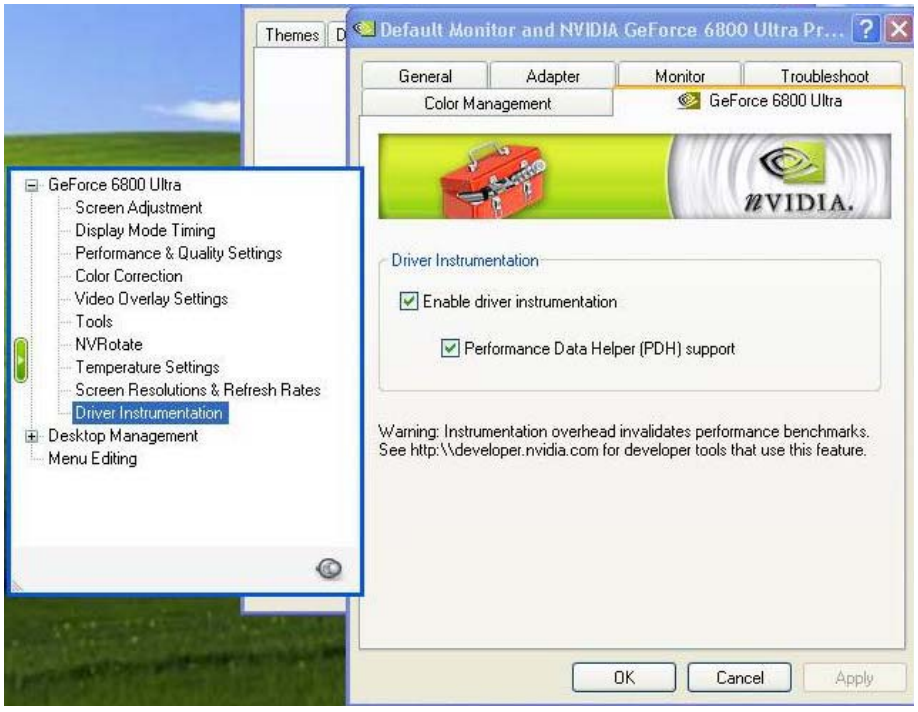
NVPerfKit signals may or may not be available on other NVIDIA GPUs.

NVPerfKit
Getting Started

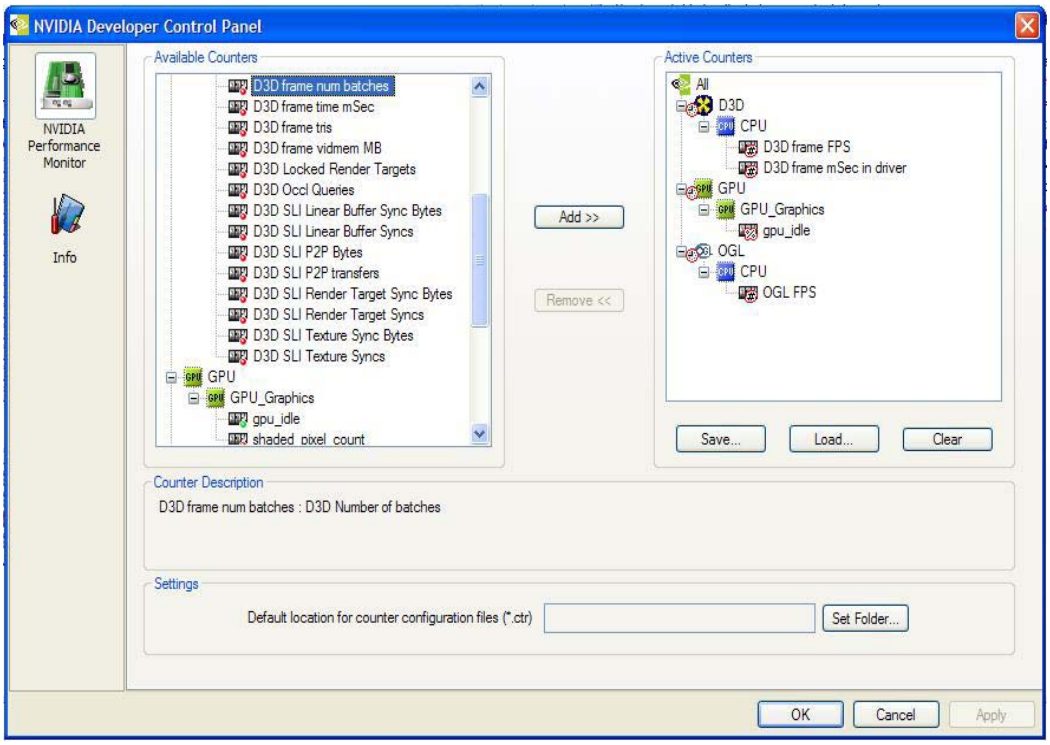
Installing the Instrumented Driver and NVPerfKit

Follow the instructions below to install the instrumented driver and get started using NVPerfKit.

1. Install the instrument driver by unzipping the driver zip file into a directory and running **setup.exe.** You will need to reboot following the driver installation.
2. Install NVPerfKit, by double clicking on **NVPerfKit_<version>.exe** and following the installer prompts.
3. Enable driver instrumentation from the ForceWare driver control panel. Check both the **Enable driver instrumentation** and **Performance Data Helper (PDH) support** check boxes.



4. To start the NVIDIA Developer Control Panel (NVDevCPL), open the Windows Control Panel (from the Windows Start Menu) and double click on the NVIDIA Developer Control Panel icon. Once it is open, you can select which signals to report while the application is running. Note that turning on signals incurs overhead so only enable signals you are interested in for the given experiment.

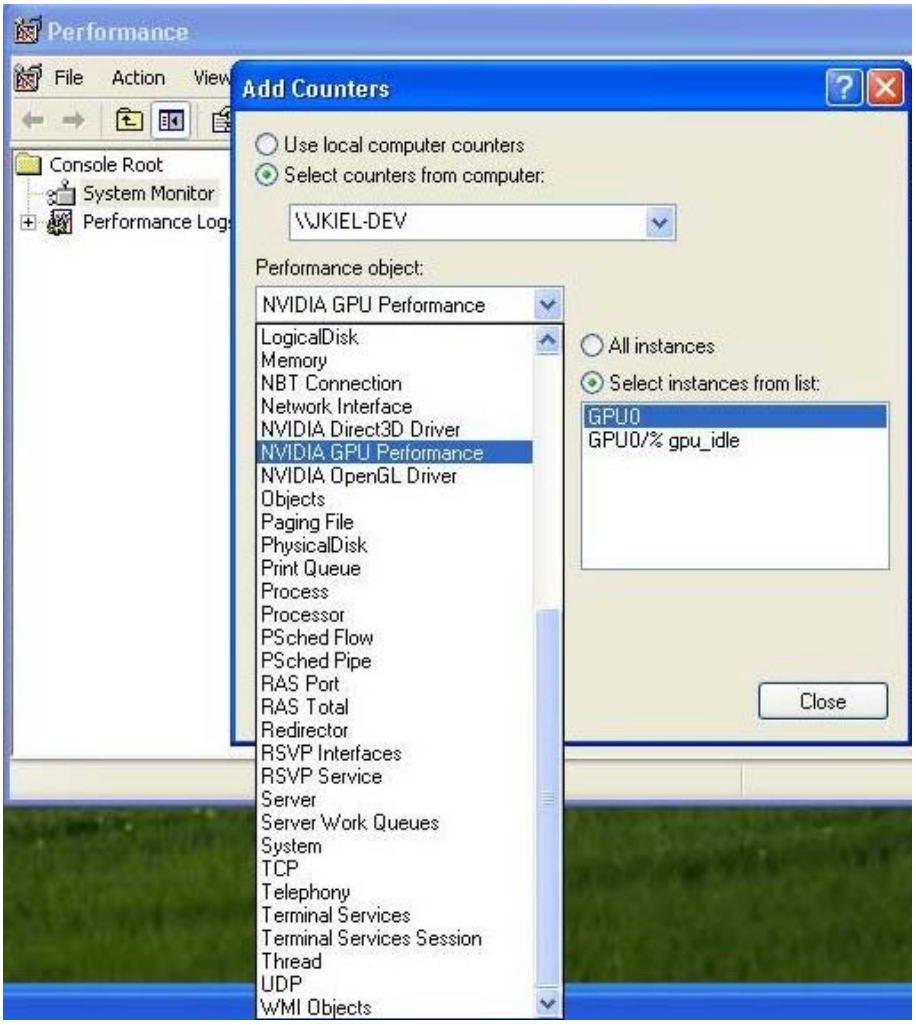


Before you try to sample a counter, make sure you have added it to the list of **Active Counters**. The GPU can sample a pre-set number of counters per clock, and this number can vary from GPU to GPU. If you choose more than this number of counters, the GPU counters are sampled in a round robin fashion, and the list on the right will show an *approximately equal* icon to reflect the reduced accuracy.

If you run your application in a window, you can interactively enable/disable GPU counters. This allows you to set your application up to sample all of the counters of interest and only look at one or two at a time without having to shut down the application, rerun NVDevCPL, restart, etc. This can greatly reduce the configuration turn-around time during performance profiling runs. For a complete list of counters and a description of their use, see Appendix B.

Graphing the Results

One way to see the counters is through the Windows system utility called PerfMon. This helpful utility graphs PDH information over time. Once you have used the NVDevCPL to enable the counters you want to sample, you can add them to the PerfMon graph using the + toolbar button. You need to select one of the NVIDIA performance objects from the drop-down list (Direct3D Driver, GPU Performance, or OpenGL Driver), and then the instance you want to graph.



If you want to use the counters in your own application, use the helper classes supplied with NVPerfKit, which include a PDH interface as well as a simple, API agnostic graphing library (see Appendix C for details). Consult the sample code for hints on how to use these. You can also call PDH directly and use the sampled values in any way that makes sense for your application.

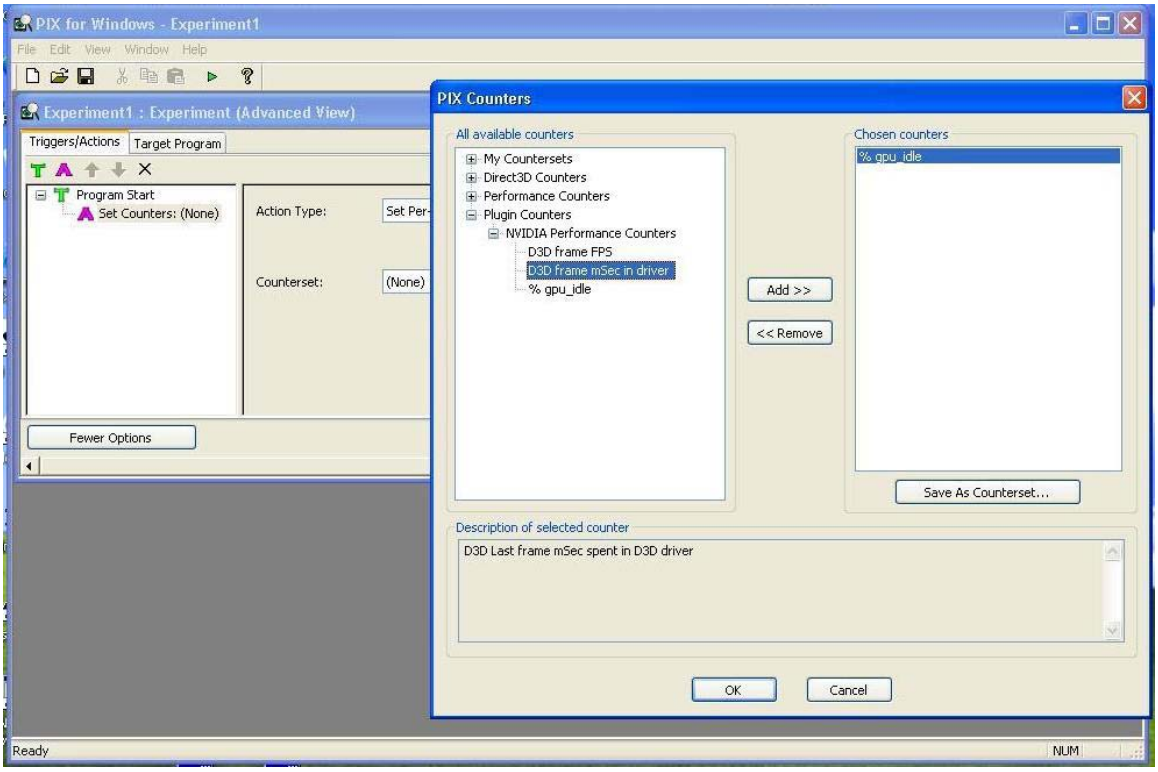
Following is the sample code for setting up PDH:

```
// Setup
PDH_HQUERY hQuery;
PDH_COUNTER hCounter;

PDH_STATUS status = PdhOpenQuery(0,0,&hQuery);
PdhAddCounter(hQuery,
"\\NVIDIA GPU Performance(GPU0/% gpu_idle)\\GPU
Counter Value",0,&hCounter));

// Periodically...
PDH_STATUS status = PdhCollectQueryData(hQuery);
PDH_FMT_COUNTERVALUE cvValue;
PdhGetFormattedCounterValue(hCounter,
PDH_FMT_DOUBLE|PDH_FMT_NOCAP100|PDH_FMT_NOSCALE,0,
&cvValue);

// cvValue.doubleValue
```

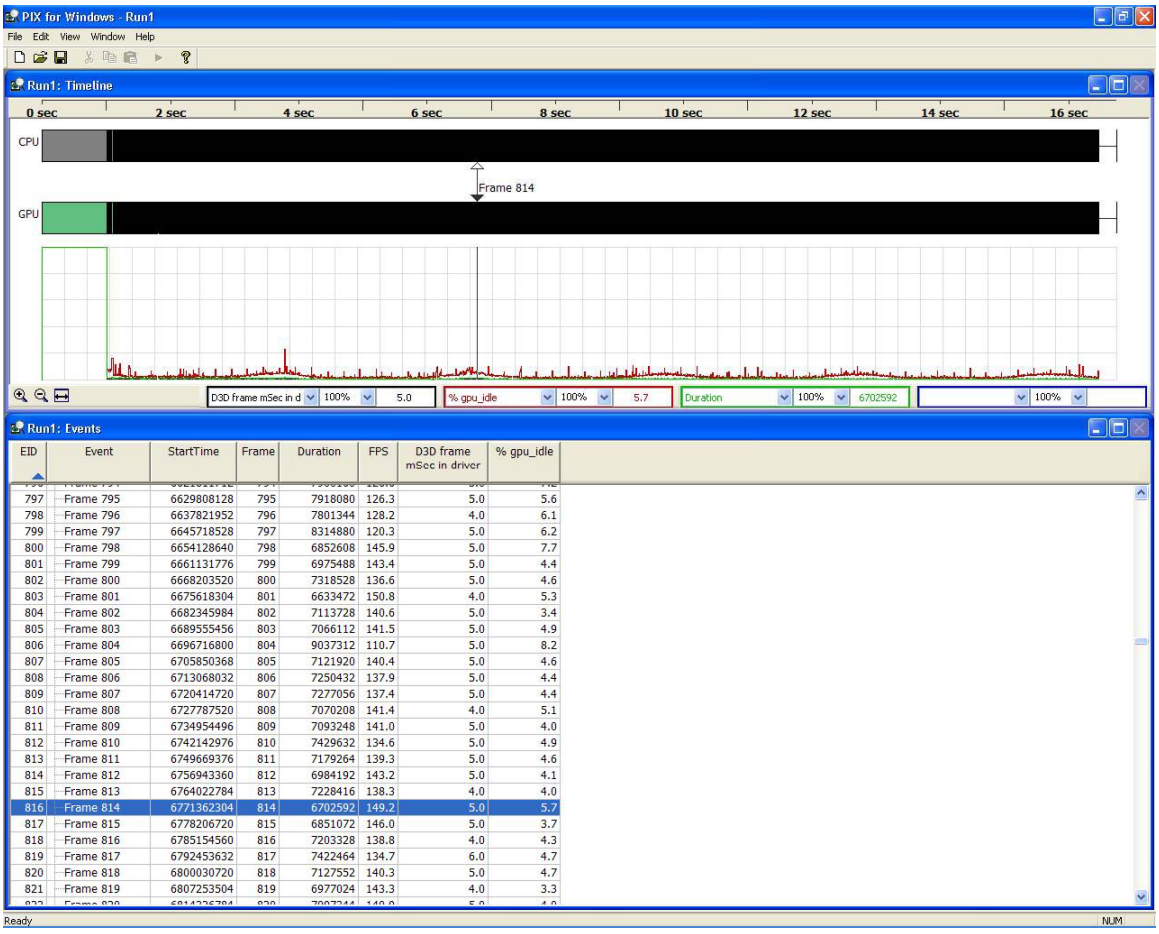
NVIDIA Plug-in for Microsoft PIX for Windows

NVPerfKit includes a plug-in that allows you to use all the NVPerfKit performance counters in Microsoft PIX for Windows. This PIX plug-in enables you to display driver and GPU counter data alongside the associated Direct3D calls for additional correlation and performance tuning. The NVPerfKit installer places the PIX plug-in in the appropriate directory for PIX to access it. To set up sampling, first remember to enable the counters that you are interested in the NVDevCPL (see Installing the Instrumented Driver and NVPerfKit above). Once this is done, you are ready to enable the counters in PIX.

From the Experiment window in PIX, make sure you select the Advanced View (using the More Options button from the Basic View). Select the Action Type "Set Per-Frame Counters" in the upper combo box and then press the Customize button. This will bring up the PIX Counters dialog with the available counter types on the left. Open the Plug-in Counters element and the NVIDIA Performance Counters sub element to display the counters you enabled in the NVDevCPL. Select the counters of interest and press the Add button. These will now show up in the data stream that PIX produces.



Here is an example of PIX for Windows output:



Appendix A.
Frequently Asked Questions

What are all of these files and where are they installed?

c:\windows\system32
- NVPMAPI driver (Standard ForceWare driver with additional instrumentation)
- nvpmapi.dll (PDH implementation)
- nvprfsmb.h (PDH implementation)
- nvprfctr.ini (PDH implementation)
- nvdevcpl.cpl (Driver Control Panel)
- nvdevtray.dll (Tray extension)

c:\Program Files\NVIDIA Corporation\NVIDIA NVPerfKit
- samples\OpenGL*.*
- samples\Direct3D*.*
- samples\common*.*

What does this error message mean, "HW necessary for GPU counters is unavailable, HW counters are disabled."

Not all GPUs have the features necessary to provide the GPU counter data. NVPerfKit signals are available on *all* NVIDIA GPUs listed under System Requirements. NVPerfKit signals may or may not be available on other GPUs.

Can I run NVPerfKit with multiple monitors?

NVPerfKit is currently not enabled to run in a multiple monitor setup. We are investigating ways to implement this feature.

What does this error message mean, "Performance monitoring has been disabled by PDH."?

PDH has a safe guard mechanism that can disable a data provider. If NVDevCPL detects this flag, you have the option of resetting it. We have not seen this happen in any released version of NVPerfKit, only during testing.

I have discovered a problem that is not listed above. Who should I call?

We want to make sure NVPerKit is a useful tool for developers analyzing their applications. Please let us know if you encounter any problems or think of additional features that would be helpful while using NVPerfKit.

Contact us at: NVPerfKit@nvidia.com

Appendix B. Counters Reference

There two types of counters available through NVPerfKit. Hardware counters provide data directly from various points inside the GPU, while the software counters, both OpenGL and Direct3D, give insight into the state and performance of the driver. All of the GPU counters give results accumulated from the previous time it was sampled. For instance, the triangle_count gives the number of triangles rendered since the last sample was taken. If you are using perfmon to sample these counters, you need to remember that it will be sampling once per second, so to get the average number of triangles per frame you need to divide by the average frame rate during that time. Once you integrate the counters into your own application, and can sample on a per frame basis, the numbers can then correlated to a given frame.

All of the software/driver counters represent a per frame accounting. These counters are accumulated and updated in the driver per frame, so even if you sample at a sub frame rate frequency, the software counters will hold the same data (from the previous frame) until the end of the current frame.

Also, counters can be reported in one of two methods: raw and percentage. Raw is a count of events (triangles, milliseconds, pixels, etc.) since the last call. Percentage counters are divided by the number of cycles since the last sample since they are event counts based on the clock rate. gpu_idle is a example, since it counts the number of clock ticks that the GPU was idle since the last call. When divided by the total number of clock ticks, you get a % of time that the GPU was idle.

The table below shows a description of the available software and hardware counters. A # next a counter denotes a raw counter, while a % denotes a percentage counter.

When configuring your application to use PDH counters, you need to know the official name of the counter and how to construct the identifier string for PDH. The tables below show the performance counters available in each counter domain.

The syntax for counters is:

```
\\Machine\PerfObject(ParentInstance/ObjectInstance#I
nstanceIndex)\\Counter Type
```

Direct3D Counters

All of the Table 1 lists the Direct3D counter names and descriptions.

Table 1. Direct3D Counters

Direct3D Counter Description	Official Name
FPS (#)	D3D frame FPS
Frame Time (1/FPS) (#)	D3D frame time mSec
Driver Time (#)	D3D frame mSec in driver
Driver Sleep Time (all reasons) (#)	D3D frame mSec Sleeping
Driver Sleep Time (waiting on GPU) (#)	D3D CPU Wait Total
Driver Sleep Time (index buffer locking) (#)	D3D CPU Wait - IB Lock
Driver Sleep Time (vertex buffer locking) (#)	D3D CPU Wait - VB Lock
Triangle Count (#)	D3D frame tris
Batch Count (#)	D3D frame num batches
Locked Render Targets Count (#)	D3D Locked Render Targets
Number of Occlusion Queries(#)	D3D Occl Queries
AGP/PCIE Memory Used in Integer MB (#)	D3D frame agpmem MB
AGP/PCIE Memory Used in Bytes (#)	D3D frame agpmem bytes
Video Memory Used in Integer MB (#)	D3D frame vidmem MB
Video Memory Used in Bytes (#)	D3D frame vidmem bytes
Total Number of GPU to GPU Transfers (#)	D3D SLI P2P transfers
Total Byte Count for GPU to GPU Transfers (#)	D3D SLI P2P Bytes
Number of IB/VB GPU to GPU Transfers (#)	D3D SLI Linear Buffer Syncs
Byte Count of IB/VB GPU to GPU Transfers (#)	D3D SLI Linear Buffer Sync Bytes
Number of Render Target Syncs (#)	D3D SLI Render Target Syncs
Byte Count of Render Target Syncs (#)	D3D SLI Render Target Sync Bytes
Number of Texture Syncs (#)	D3D SLI Texture Syncs
Byte Count of Texture Syncs (#)	D3D SLI Texture Sync Bytes

Syntax:

\\NVIDIA Direct3D Driver(CPU/Counter name\\D3D Counter Value

Example: FPS

\\NVIDIA Direct3D Driver(CPU/D3D frame FPS\\D3D Counter Value

OpenGL Counters

Table 2 lists the OpenGL counter names and descriptions.

Table 2. OpenGL Counters

OpenGL Counter Description	Official Name
FPS (#)	OGL FPS
Frame Time (1/FPS) (#)	OGL frame time mSec
Driver Sleep Time (driver waits for GPU) (#)	OGL frame mSec Sleeping
% of the Frame Time driver is waiting (%)	OGL % driver waiting
AGP/PCIE Memory Used in Integer MB (#)	OGL AGP/PCI-E usage (MB)
AGP/PCIE Memory Used in bytes (#)	OGL AGP/PCI-E usage (bytes)
Video Memory Used in Integer MB (#)	OGL vidmem usage (MB)
Video Memory Used in bytes (#)	OGL vidmem usage (bytes)

Syntax:

\\NVIDIA OpenGL Driver(CPU/Counter name\\OGL Counter Value

Example: FPS:

\\NVIDIA OpenGL Driver(CPU/OGL FPS\\OGL Counter Value

GPU Counters

Table 3 lists the GPU counter names and descriptions.

Table 3. GPU Counters

GPU Counter Description	Official Name
GPU Idle (%)	gpu_idle
Pixel Shader Utilization (%)	pixel_shader_busy
Shader Stalls (%)	shader_waits_for_texture
Texture Stalls (%)	texture_waits_for_ shader
ROP Stalls (%)	shader_waits_for_rop
FastZ Utilization (#)	fast_z_count
Vertex Attribute Count (#)	vertex_attribute_count
Pixel Count (#)	shaded_pixel_count
Vertex Count (#)	vertex_count
Triangle Count (#)	triangle_count
Primitive Count (#)	Primitive_count
Culled Primitive Count (#)	culled_primitive_count

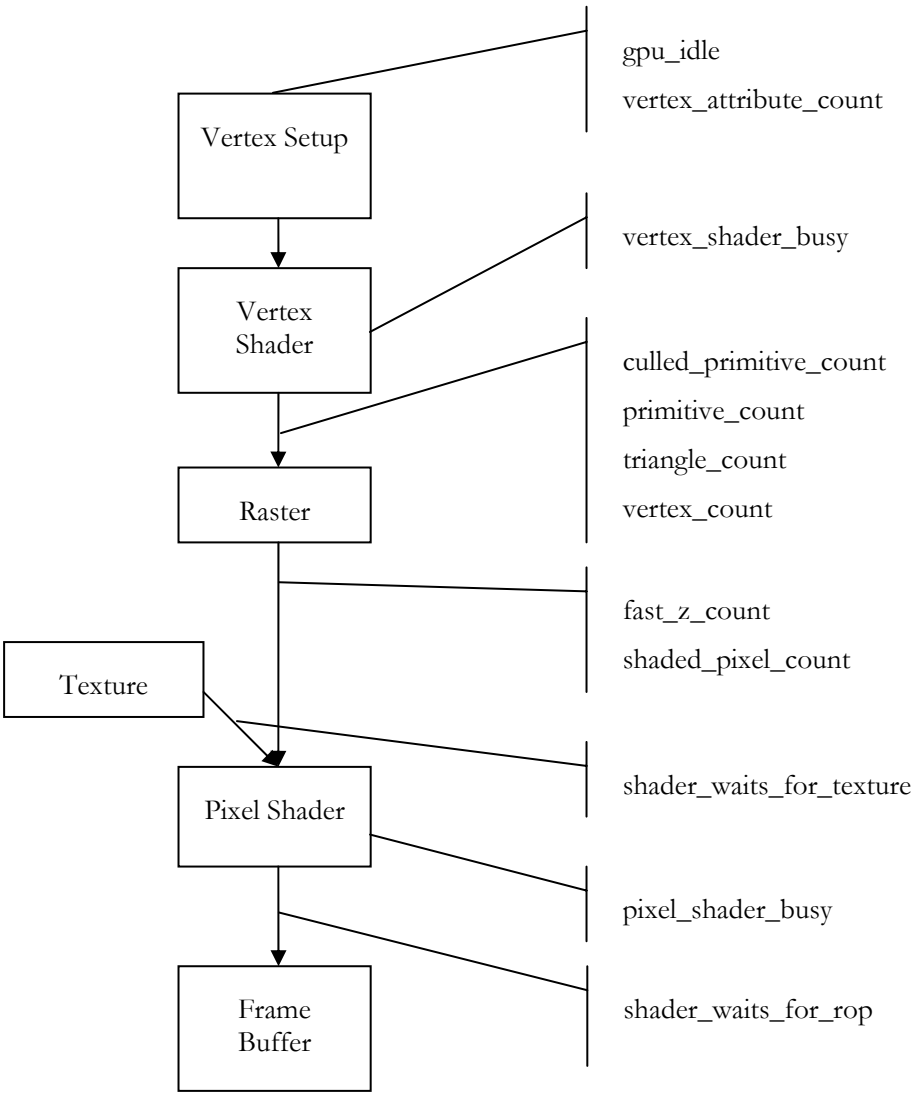
Syntax:

\\NVIDIA GPU Performance(GPU0/Counter name\\GPU Counter Value

Example: GPU Idle:

\\NVIDIA GPU Performance(GPU0/% gpu_idle\\GPU Counter Value

This block diagram shows where in the GPU pipeline each counter falls.



gpu_idle: This is the % of time the GPU is idle since the last call. Obviously, having the GPU idle at all is a waste of valuable resources. In general, you want to balance the GPU and CPU work loads so that no one resource is starved for work. Time management or using multithreading in your application can help balance CPU based tasks (world management, etc.) with the rendering pipeline.

vertex_attribute_count: The number of vertex attributes that are fetched and passed to the geometry unit is returned in this counter. A large the number of

attributes (or unaligned vertices) can hurt vertex cache performance and reduce the overall vertex processing capabilities of the pipeline.

culled_primitive_count: Returns the number of primitives culled in primitive setup. If you are performing viewport culling, this gives you an indication of the accuracy of the algorithm being used, and can give you an idea if you need to improve this culling. This includes primitives culled when using backface culling. Drawing a fully visible sphere on the screen should cull half of the triangles if backface culling is turned on and all the triangles are ordered consistently (CW or CCW).

vertex_shader_busy: This is the % of time that vertex shader unit 0 was busy. If this value is high but, for instance, pixel_shader_busy is low, it is an indication that you may be vertex/geometry bound. This can be from geometry that is too detailed or even from vertex programs that are overly complex and need to be simplified. In addition, taking advantage of the post T&L cache (by reducing vertex size and using indexed primitives) can prevent processing the same vertices multiple times.

primitive_count: Returns the number of primitives processed in the geometry subsystem. This experiment counts points, lines, and triangles. To count only triangles, use the triangle_count counter. Balance these counts with the number of pixels being drawn to see if you need to simplify your geometry and use bump/displacement maps.

triangle_count: Returns the number of triangles processed in the geometry subsystem

vertex_count: Returns the number of unique vertices transformed by the geometry. This can give you an idea of how good your vertex sharing is from the use of strips/fans/etc.

fast_z_count: This returns the number of blocks that were processed through the GPU's fastZ hardware. If you are doing z only passes, this will let you know if you are utilizing the hardware optimally.

shaded_pixel_count: Counts the number of pixels generated by the rasterizer and sent to the pixel shader units.

shader_waits_for_texture: This is the amount of time that the pixel shader unit was stalled waiting for a texture fetch. Texture stalls usually happen if textures don't have mipmaps, if a high level of anisotropic filtering is used, or even for bad coherency in accessing textures.

pixel_shader_busy: This returns the % of time that pixel shader unit 0 was busy and is an indication of if you are pixel bound. This can happen in high resolution settings, when pixel programs are very complex

shader_waits_for_rop: This is the % of time that the pixel shader is stalled by the raster operations unit (ROP), waiting to blend a pixel and write it to the frame buffer. If the application is performing a lot of alpha blending, or even if the application has a lot of overdraw (the same pixel being written multiple times, unblended) this can be a performance bottleneck.

Appendix C. Sample Code

The sample code provided with NVPerfKit illustrates how to implement support for the performance counters in your application via PDH.

Note: PDH is the Performance Data Helper interface provided by Microsoft and used by perfmon.exe and others.

The purpose of this sample code is twofold:

- Provide code you can copy/paste into your own applications
- Demonstrate the performance issues associated with using the performance counters

To use this sample code, you must have installed an instrumented driver and also enabled performance instrumentation in the display driver control panel. You must also use the NVIDIA Developer Control Panel to enable the following counters:

- gpu_idle
- D3D frame mSec in driver
- OGL FPS

The OpenGL Demo draws a simple tessellated sphere. The number of tessellations varies smoothly each frame, except every 100th frame it draws the sphere very highly tessellated for that single frame (the D3D demo currently doesn't draw any geometry). While this is happening, the OpenGL Demo displays the values of the counters in various ways on the screen.

The code accompanying this demo includes source code for 3 helper classes and examples of how to use them.

- **`CPDHHelper`** wraps some of the Win32 PDH library's calls for simpler usage.
- **`CTrace`** is similar to a hybrid **`Queue`** and **`CircularQueue`** (it can be used both ways). It is for storing values read from the **`CPDHHelper`** so that a counter's history can be available.
- **`CTraceDisplay`** is a helper class for displaying the trace data in a variety of manners.

Use **CPDHHelper::add()** and the identifier string for each counter you want to monitor. The construction of this string is a bit ugly, so please pay attention to how this is done in the demo. Open perfmon.exe (supplied with windows) and use the add feature to add a new counter. Inspection of the displayed counter name and information along with comparison to the sample strings should be sufficient for your usage. MSDN has further information about the construction of the string, in addition to a few macros and other tools to help with it.

Once counters are added to **CPDHHelper**, call sample() to retrieve values. Then call value**(i)** where **i** is the number of the counter you want to read (0 based, in the order you added them). This returns a win32 structure. The "doubleValue" entry is demonstrated in the Demo code, but you may prefer others.

Values are **insert()'d** into a **CTrace**. Values can be read out either via the **[]** operator or the **()** operator. One *streams* the data, the other *wraps* it, in wrap-around style.

CTraceDisplay can display data in a variety of ways. **LINE_STREAM** uses the **[]** operator for a streaming plot. There are also **BAR** and **NEEDLE** methods. Play around and use your favorites. The display's are in a bounding box provided at creation time or later, with 0,0 being the bottom left corner of the window. A background color may be selected, including alpha values. You can enable blending in the mode of your choice if you want to be able to "see through" the displayed trace. **CTraceDisplay** has sub classes for Direct3D and OpenGL to implement some API specific calls.

Further details are in the sample code.

Contact

Please let us know if you encounter any problems or think of additional features that would improve NVPerfKit. You can reach us at the following Email address:

NVPerfKit@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com