NVIDIA

User Guide

NVPerfKit

NVIDIA Performance Toolkit

DEVELOPMENT

Table of Contents

Introduction

Please read this entire document before you get started with NVPerfKit. Several important issues are covered in this document that will help get things running smoothly. Also, some counter names have changed since NVPerfKit 1.0 was released.

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 through NVPerfSDK using the NVPerfAPI, as well as through PerfMon and the Windows Management Instrumentation (WMI) Performance Data Helper (PDH) interface. Also, we have included a plug-in for Microsoft PIX for Windows to export the data while running Microsoft PIX experiments.

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
- NVPerfHUD 4.0 (See separate NVPerfHUD documentation)
- PIX for Windows NVIDIA Plug-in
- NVPerfSDK
  - NVPerfAPI libraries, includes, and sample code
  - PDH based interface
    - NVDevCPL Control Panel applet
    - Sample code and helper classes
- gDEBugger (30 day trail version)

System Requirements

- NVIDIA instrumented display driver, version 83.60 or later on Windows XP
- NVPerfKit signals are available on all NVIDIA GPUs listed below:
  - GeForce 7800 GTX 512
  - GeForce 7800 GTX

- GeForce 6800 Ultra
- GeForce 6800 GT
- GeForce 6600

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

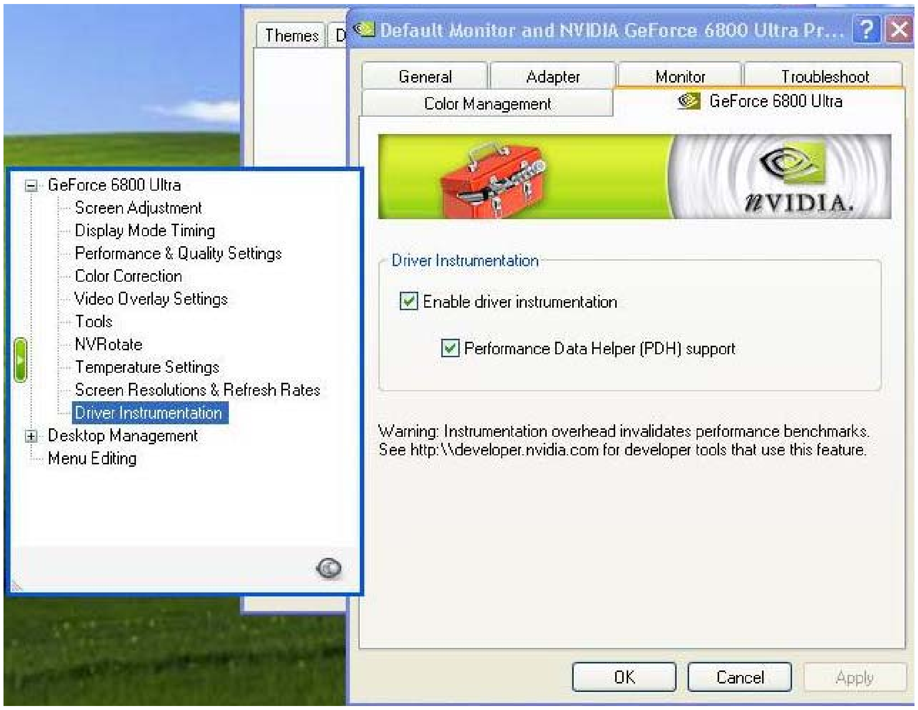
NVPerfKit
Getting Started

Installing NVPerfKit

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

1. Install NVPerfKit by double clicking on the NVPerfKit.exe file downloaded from the NVIDIA developer web site. This will install the Instrumented Driver, NVPerfHUD, and NVPerfSDK.

2. Ensure that driver instrumentation is enabled from the ForceWare driver control panel. Both the **Enable driver instrumentation** and **Performance Data Helper (PDH) support** should be checked.



Using NVPerfSDK

There are now two ways to access the GPU and driver data made available with NVPerfKit from within your own application. The first is using the NVPerfAPI, and the second is through the Performance Data Helper (PDH) interface introduced with NVPerfKit 1.0 and is described in the next section. Finally, the NVIDIA Plugin for Microsoft PIX for Windows is described last.

Using NVPerfAPI

The NVPerfAPI implementation is provided via the NVPerfSDK.h and NVPerfSDK.lib files included in the NVPerfKit distribution. This API provides the developer with greater access to the capabilities of the GPU and driver counters, as well as providing an interface to Simplified Experiments (SimExp), which give even more detailed yet easy to use information about GPU performance.

The typical application that wants to sample GPU (using round robin sampling) and driver counters requires just a few source code changes. During setup, make a call to ***NVPMInit()*** with a similar call to ***NVPMShutdown()*** during cleanup and shutdown. To add a counter, simply call ***NVPMAddCounterByName("gpu_idle")***, substituting the counter of interest for "gpu_idle" in this example. Finally, once per frame, call ***NVPMSample(NULL, &nCount)*** to sample the currently active counters and ***NVPMGetCounterValueByName("gpu_idle", 0, &value, &cycle)*** to retrieve the resulting counter value. Any number of driver counters can be enabled concurrently and will be updated every frame. GPU counters, however, are a more limited resource, and can only sample a certain number of counters per frame. The counter values can always be queried, but they will be refreshed in a round robin fashion as they are sampled.

Simplified Experiments (SimExp)

One of the new features provided by NVPerfAPI is the ability to run directed experiments on the individual units of the GPU and gather performance characteristics, called Simplified Experiments. For 8 locations in the GPU pipeline, SimExp provides a "Speed of Light" (SOL) and a "Bottleneck" value. The speed of light of a unit can be thought of as a utilization measurement. The "value" returned is a count for how many cycles during the experiment the unit was active, and the "cycle" returned gives the amount of time the experiment took to run. Both of these values are in picoseconds. If you take the value returned and divide it by the cycles, you get percentage utilization. Similarly, when running a Bottleneck experiment, the value roughly represents the amount of time this unit was a bottleneck and the cycles is the experiment duration. Divide value by cycle and you get a percentage of time that this unit was the bottleneck.

Finally, there is an additional counter that will run all of the experiments needed to determine what unit in the GPU is the bottleneck. It runs all of the speed of light and bottleneck experiments, passes the results through an expert system, and returns an ordinal value for the unit that is the bottleneck. You can translate that into a string name using ***NVPMGetGPUBottleneckName(value, name)***.

Since the Simplified Experiments require collecting data from multiple counters in the GPU, they require multiple passes across the **same scene data** (as if the game and all animations were paused) to complete the experiment. From a paused frame in the application, this is accomplished using the ***NVPMBeginExperiment()/NVPMEndExperiment()*** mechanism, detailed below. As always, you still setup NVPerfAPI using ***NVPMInit()*** and enable the counter of interest using ***NVPMAddCounterByName("GPU Bottleneck")***. Then, inside of your drawing loop, you would do the following:

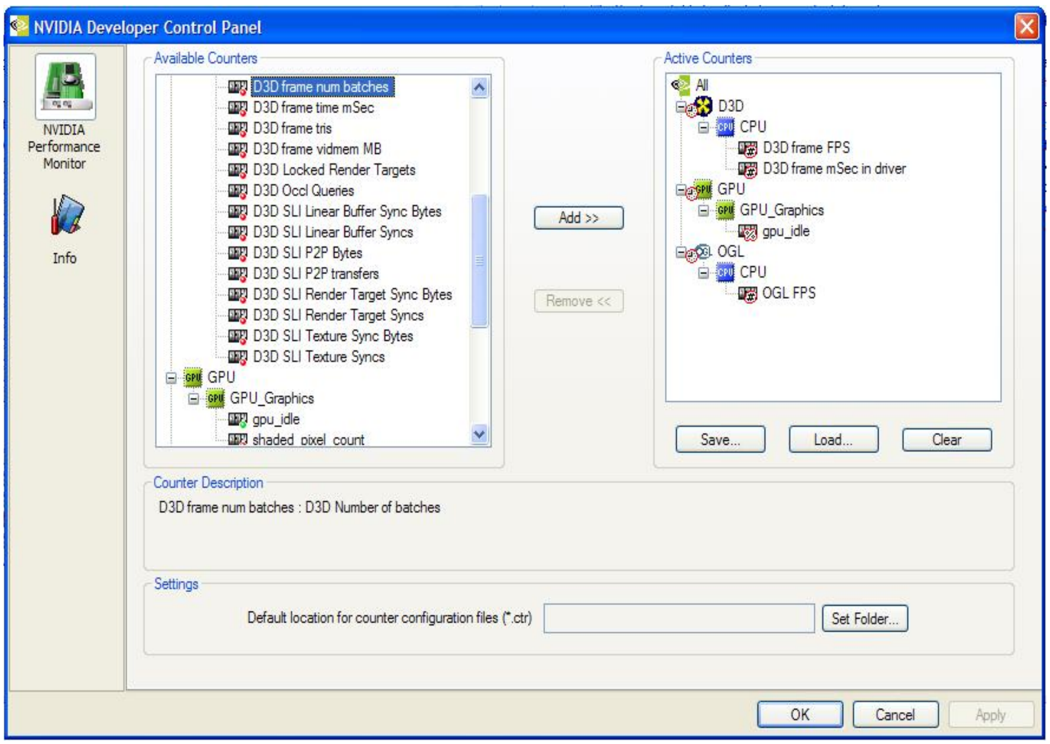
```
NVPMBeginExperiment(&nCount);
for(i = 0; i < nCount; i++) {
    NVPMBeginPass(i);
          // Draw the scene, including Present or
SwapBuffers
    NVPMEndPass(i);
}
NVPMEndExperiment();
```

Once this is completed, you can query ***NVPMGetCounterValueByName("GPU Bottleneck", 0, &value, &cycles)*** to determine which unit is the bottleneck. Because all of the underlying speed of light experiments and bottleneck experiments are run in order to determine this value, you can also query those values when the experiment is over.

One of the things that NVPerfHUD does in order to further analyze the scene for performance issues is to group sets of draw calls by the current GPU state (including pixel shader/vertex shader, textures, render target, etc.). This is accomplished by timing each individual draw call and collecting similar draw calls into "state buckets". Each draw call can be timed using the ***NVPMBeginObject()/NVPMEndObject()*** mechanism. Once you know how many draw calls are in your scene, allocate space for them using the ***NVPMAllocObjects(count)*** call. Then, inside of the ***NVPMBeginPass() NVPMEndPass()*** pair, add calls to ***NVPMBeginObject(objectId) NVPMEndObject(objectId)*** around the draw call, and call Present or SwapBuffers after the last ***NVPMEndObject()*** but before ***NVPMEndPass()***. See Appendix C for further details on the NVPerfAPI specification.

Using NVPerfSDK with PDH

When using PDH, you first need to tell the driver and PDH subsystem what counters you are interested in collecting. This is done through the NVIDIA Developers Control Panel (NVDevCPL). To start the 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.

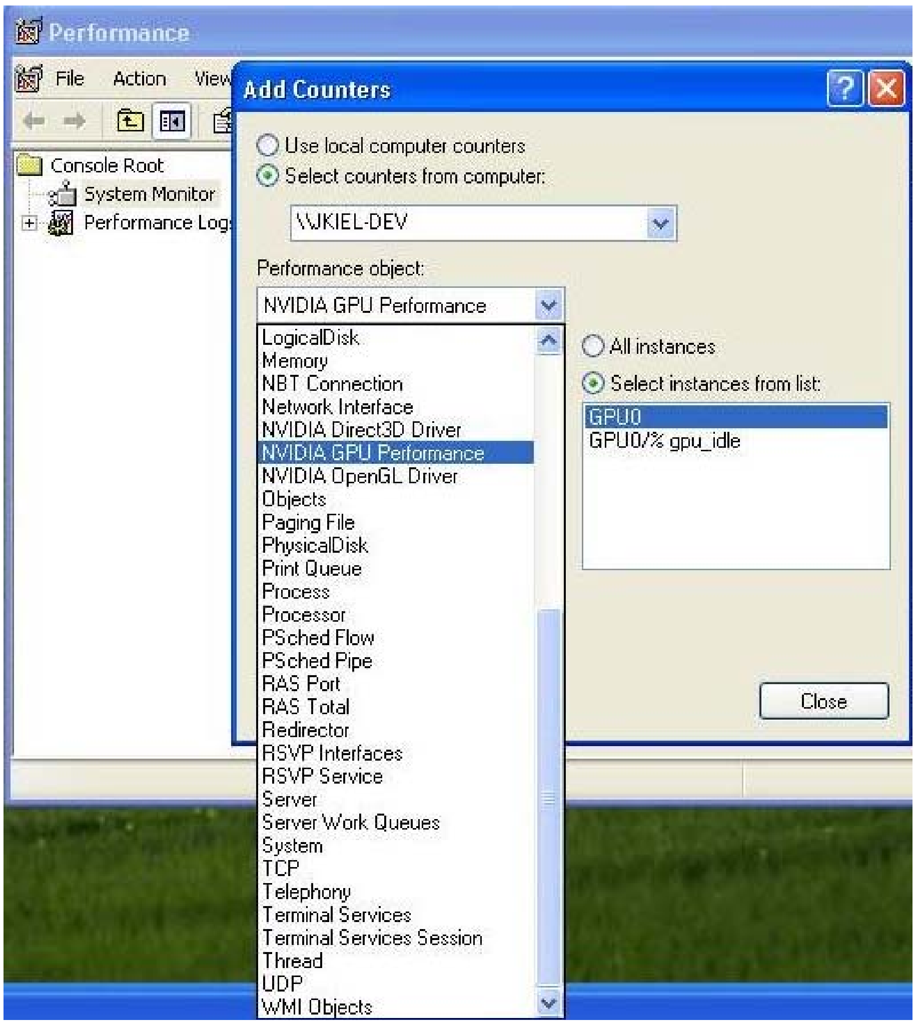


Before you try to sample a counter, make sure you have added it to the list of **Active Counters**. The GPU can sample a pre-set number of counters per clock, and this number can vary from GPU to GPU. If you choose more than this number of counters, the GPU counters are sampled in a round robin fashion, 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 D 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.

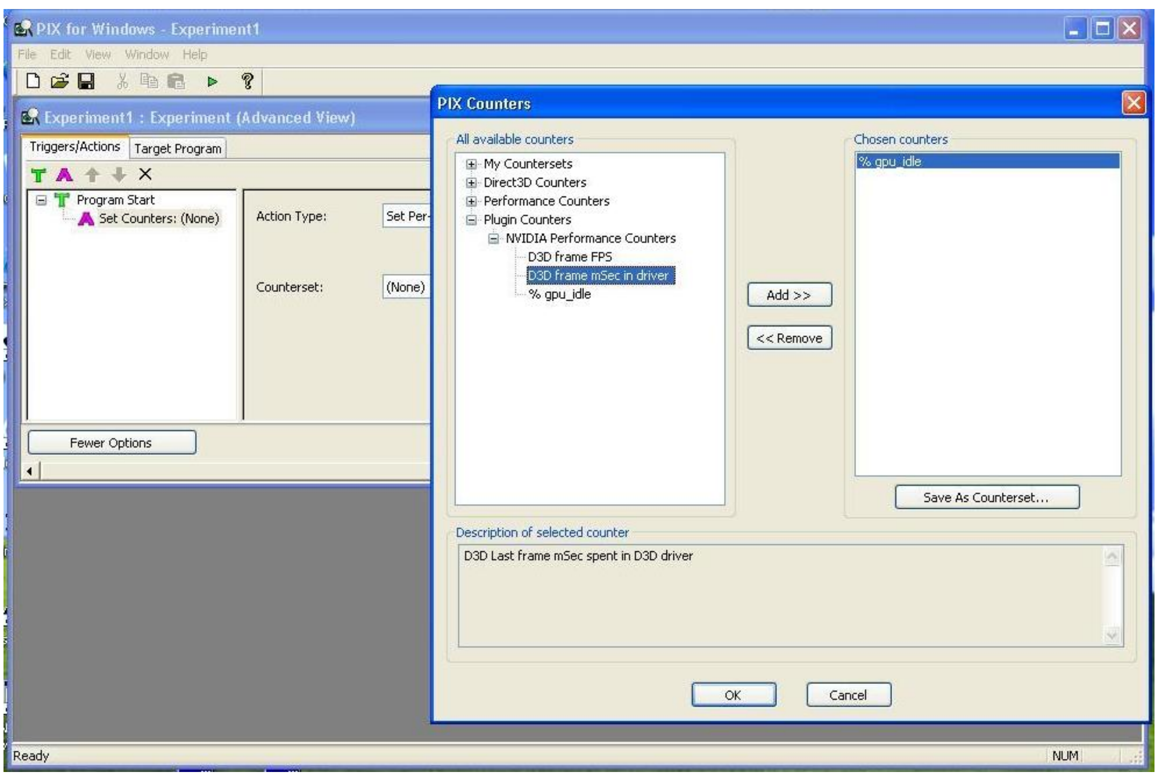
Following is the sample code for setting up PDH:

```
// 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
```

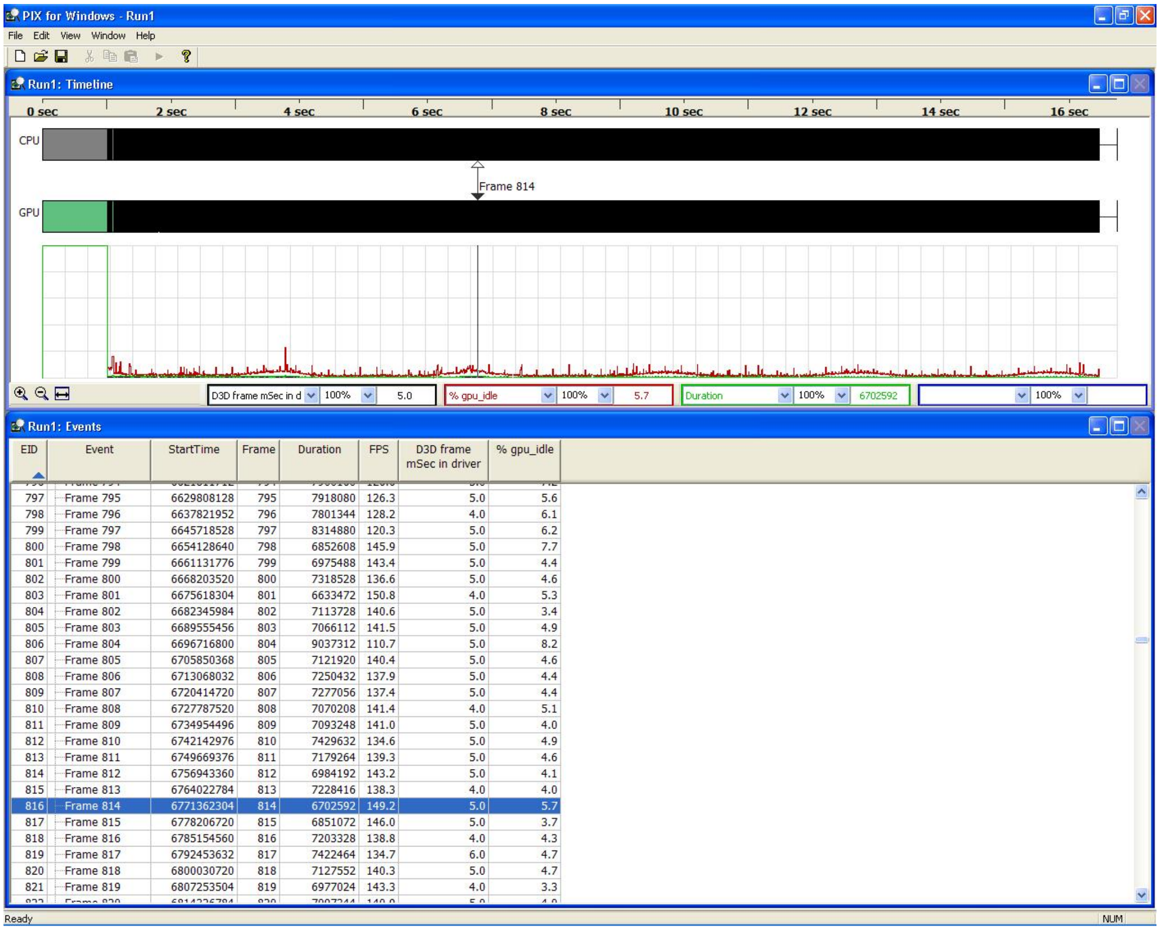
NVIDIA Plug-in for Microsoft PIX for Windows

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 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 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.
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 three 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. Simplified Experiments are multipass experiments that give detailed information about the state of the GPU. All of the GPU counters give results accumulated from the previous time the GPU 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 be 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 (sub-framerate? sub-frame rate? …?) frequency, the software counters will hold the same data (from the previous frame) until the end of the current frame.

When using the PDH interface, counters can be reported in one of two methods: raw and percentage. Raw counters count events (triangles, pixels, milliseconds, etc.) since the last call. Percentage counters are event counts based on the clock rate; event counts divided by the number of cycles since the last sample. For example, gpu_idle 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.

In contrast, sampling the GPU counters with the NVPerfAPI will return raw values for the event and cycle counts. The cycle count always represents the number of GPU clock cycles that the given experiment was run. The event count, in the case of utilization and stall counters, is the number of cycles that triggered a "true" value when sampling. The easiest way to use this data is to divide by the cycle count to get a % utilization or a % stalling number. For the raw counts (vertex count, triangle count, etc.), the event count is the number of triangles (elements?) that were processed.

The Simplified Experiments report the results in a hybrid fashion. The event is the integer percentage of the counter (*XXX* SOL, *XXX* Bottleneck) representing % utilization and % of the time the unit was a bottleneck, respectively. The cycle count is the number of picoseconds that the experiment was run. Finally, the result of GPU Bottleneck is an integer in the event count that is the unit that is determined to be the system bottleneck.

Table 1 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 using the counters with NVPerfAPI, you can use the "Official Name" as denoted in the chart. When configuring your application to use PDH counters, you need to construct the identifier string for PDH using the Official Name. 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

Table 1. Direct3D Counters

Direct3D Counter Description	Official Name
FPS (#)	D3D FPS
Frame Time (1/FPS) (#) in mSec	D3D frame time
Driver Time (#) in mSec	D3D time in driver
Driver Sleep Time (all reasons) (#) in mSec	D3D driver sleeping
Triangle Count (#)	D3D triangle count
Triangle Count Instanced (#)	D3D triangle count instanced
Batch Count (#)	D3D batch count
Locked Render Targets Count (#)	D3D Locked Render Targets
AGP/PCIE Memory Used in Integer MB (#)	D3D agpmem MB
AGP/PCIE Memory Used in Bytes (#)	D3D agpmem bytes
Video Memory Used in Integer MB (#)	D3D vidmem MB
Video Memory Used in Bytes (#)	D3D vidmem bytes
Total video memory available in bytes (#)	D3D vidmem total bytes
Total video memory available in integer MB (#)	D3D vidmem total MB
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

PDH Syntax:

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

PDH Example: FPS

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

Note that "D3D triangle count" will return the total number of primitives, summed up from the primitive count sent in the DrawPrimitive call, not taking into account instancing. "D3D triangle count instanced" takes into account the frequency divider and returns the total number of triangles sent to the GPU.

OpenGL Counters

Table 2. OpenGL Counters

OpenGL Counter Description	Official Name
FPS (#)	OGL FPS
Frame Time (1/FPS) (#) in mSec	OGL frame time
Driver Sleep Time (waits for GPU) (#) in mSec	OGL driver 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)
Total amount of video memory in bytes	OGL vidmem total bytes
Total amount of video memory in integer MB	OGL vidmem total MB
Number of batches in the frame	OGL Frame Batch Count
Number of vertices in the frame	OGL Frame Vertex Count
Number of primitives in the frame	OGL Frame Primitive Count

PDH Syntax:

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

PDH Example: FPS:

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

GPU Counters

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
ROP Stalls (%)	shader_waits_for_rop

ROP Utilization (%)	rop_busy
FastZ Utilization (#)	fast_z_count
Vertex Attribute Count (#)	vertex_attribute_count
Vertex Shader Utilization (%)	vertex_shader_busy
Pixel Count (#)	shaded_pixel_count
Vertex Count (#)	vertex_count
Triangle Count (#)	triangle_count
Primitive Count (#)	Primitive_count
Culled Primitive Count (#)	culled_primitive_count

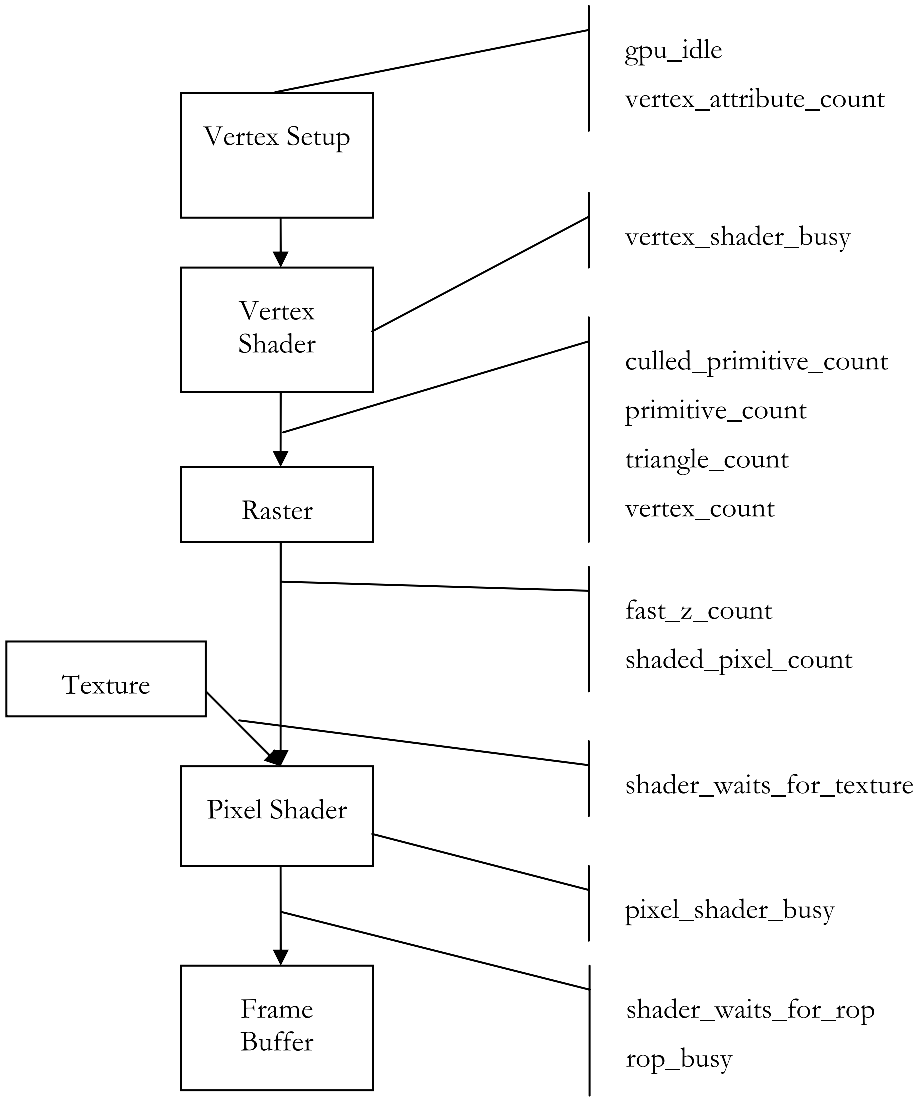
Syntax:

\\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 could simplify your geometry and use bump/displacement maps, for example.

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 if there is poor 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 or 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.

rop_busy: % of time that the ROP unit is actively doing work. This can be high if alpha blending is turned on, of overdraw is high, etc.

Simplified Experiments (SimExp)

Table 4 lists the Simplified Experiments. The value returned is picoseconds that the unit was utilized or the bottleneck and the cycles returned is picoseconds that the experiment was run. Divide value by cycles to get % bottleneck and % utilization.

Table 4. Simplified Experiments

SimEXP Counter Description	Official Name
2D Unit (blit) is Bottleneck	2D Bottleneck
2D Unit (blit) utilization	2D SOL
Index Unit is Bottleneck	IDX Bottleneck
Index Unit utilization	IDX SOL
Vertex Shader Unit is Bottleneck	GEOM Bottleneck
Vertex Shader Unit utilization	GEOM SOL
ZCull Unit is Bottleneck	ZCULL Bottleneck
ZCull Unit utilization	ZCULL SOL
Texure Unit is Bottleneck	TEX BOTTLENECK
Texture Unit utilization	TEX SOL
Raster Operations Unit is Bottleneck	ROP BOTTLENECK
Raster Operation Unit utilization	ROP SOL
Pixel Shader Unit is Bottleneck	SHD Bottleneck
Pixel Shader Unit utilization	SHD SOL
Frame Buffer Unit is Bottleneck	FB Bottleneck
Frame Buffer Unit utilization	FB SOL
Index for GPU Bottleneck	GPU Bottleneck

Appendix C. NVPerfAPI Specification

All functions return NVPM_OK if everything worked out just fine. They can also return NVPM_ERROR_INTERNAL for internal errors. If this happens, please send email to NVPerfKit@nvidia.com with the result from NVPMGetExtendedError(). Please note that all of the NVPM_WARNING_* messages have not been implemented yet, and will be supported in a future release.

Setup NVPerfAPI:

```
NVPMRESULT NVPMInit();
```

Error return values:

NVPM_ERROR_DRIVER_MISMATCH: NVPerfAPI version and driver version do not match

Shutdown NVPerfAPI:

```
NVPMRESULT NVPMShutdown();
```

Error return values:

NVPM_ERROR_NOT_INITIALIZED: NVPMInit wasn't called or didn't complete sucessfully

Enumerate available counters:

The callback function will continue to be called until all of the counters are enumerated or until anything but NVPM_OK is returned.

typedef NVPMRESULT (*NVPMEnumFunc)(UINT unCounterIndex, char *pcCounterName);

```
NVPMRESULT NVPMEnumCounters(NVPMEnumFunc
pEnumFunction);
```

Error return values:

NVPM_ERROR_BAD_ENUMERATOR: A bad/NULL pointer was sent for the enumerator function

NVPM_WARNING_ENDED_EARLY: Enumeration was stopped before the end of the counter list was reached

Retrieve the number of counters available:

```
NVPMRESULT NVPMGetNumCounters(UINT *punCount);
```

Get various counter information:

Passing NULL for pcString and a valid pointer for punLen will return the length of the name in punLen. Passing a pointer in pcString and a buffer size in punLen will attempt to write the name (\0 term) to pcString. If the buffer is too small, nothing is written and punLen is set to the string length needed.

```
NVPMRESULT NVPMGetCounterName(UINT unCounterIndex,
char *pcString, UINT *punLen);
NVPMRESULT NVPMGetCounterDescription(UINT
unCounterIndex, char *pcString, UINT *punLen);
NVPMRESULT NVPMGetCounterAttribute(UINT
unCounterIndex, UINT unAttribute, UINT *punValue);
```

Error return values:

NVPM_ERROR_STRING_TOO_SMALL: pcString is too small based on size passed in punLen

Enable a counter for sampling:

```
NVPMRESULT NVPMAddCounter(char *pcName);
NVPMRESULT NVPMAddCounter(UINT unIndex);
NVPMRESULT NVPMAddCounters(UINT unCount, UINT
*punIndices);
```

Error return values:

NVPM_ERROR_INVALID_COUNTER

Disable a counter(s):

```
NVPMRESULT NVPMRemoveCounter(char *pcName);
NVPMRESULT NVPMRemoveCounter(UINT unIndex);
NVPMRESULT NVPMRemoveCounters(UINT unCount, UINT
*punIndices);
NVPMRESULT NVPMRemoveAllCounters();
```

Error return values:

NVPM_ERROR_INVALID_COUNTER

NVPM_WARNING_COUNTER_NOT_ENABLED

NVPM_WARNING_NO_COUNTERS: No counters were enabled

Experiment interface:

Signals to NVPerfAPI that the user is ready to begin sampling. It returns in pnNumPasses the number of passes it will take to provide data for all of the enabled counters.

```
NVPMRESULT NVPMBeginExperiment(int *pnNumPasses);
NVPMRESULT NVPMEndExperiment();
```

Error return values:

NVPM_ERROR_NO_COUNTERS: No counters are enabled

NVPM_ERROR_NOT_IN_EXPERIMENT: NVPMBeginExperiment not called

NVPM_ERROR_EXPERIMENT_INCOMPLETE: Didn't call the correct number of passes specified by NVPMBeginExperiment

Pass interface:

```
NVPMRESULT NVPMBeginPass(int nPass);
NVPMRESULT NVPMEndPass(int nPass);
```

Error return values:

NVPM_ERROR_NOT_IN_EXPERIMENT: NVPMBeginExperiment() was not called

NVPM_ERROR_PASS_SKIPPED: Passes were not given in sequence

NVPM_ERROR_INVALID_PASS: An pass number not valid for the current experiment was given

NVPM_WARNING_PASS_NOT_ENDED: Previous pass was not ended with NVPMEndPass()

NVPM_ERROR_NOT_IN_EXPERIMENT: NVPMBeginExperiment() was not called

NVPM_ERROR_NOT_IN_PASS: NVPMBeginPass wasn't called or was called with another pass number

NVPM_WARNING_OBJECT_NOT_ENDED: The last NVPMEndObject was not called

NVPM_WARNING_PASS_INCOMPLETE: NVPMBeginObject()/NVPMEndObject() was not called for all allocated objects

Object interface:

Allocate slots for counter data to be put into. If this isn't done, all data is put in "slot 0". Up to NVPM_MAX_OBJECTS (currently 1024) objects are currently supported.

```
NVPMRESULT NVPMAllocObjects(int);
```

Error return values:

NVPM_OUT_OF_MEMORY: Too many objects are trying to be allocated.

```
NVPMRESULT NVPMBeginObject(int nObjectID);
```

NVPM_ERROR_UNKNOWN_OBJECT: Object was not allocated with NVPMAllocObjects()

NVPM_ERROR_NOT_IN_PASS: NVPMBeginPass was not called

NVPM_ERROR_NOT_IN_EXPERIMENT: NVPMBeginExperiment was not called

NVPM_WARNING_OBJECT_NOT_ENDED: NVPMEndObject wasn't called

```
NVPMRESULT NVPMEndObject(int nObjectID);
```

NVPM_ERROR_UNKNOWN_OBJECT: Object was not allocated with NVPMAllocObjects()

NVPM_ERROR_NOT_IN_PASS: NVPMBeginPass was not called

NVPM_WARNING_DRAW_COUNT_CHANGED: The number of DPs for the changed from one pass to the next

Retrieving results:

```
NVPMRESULT GetCounterValueByName(char *pcName, int
nObjectID, UINT64 *pulValue, UINT64 *pulCycles);
NVPMRESULT GetCounterValue (UINT unIndex, int
nObjectID, UINT64 *pulValue, UINT64 *pulCycles);
NVPMRESULT NVPMGetGPUBottleneckName(UINT ulValue,
char *pcName);
```

NVPM_ERROR_COUNTER_NOT_ENABLED: Asked for a counter that isn't currently sampling

NVPM_ERROR_EXPERIMENT_NOT_RUN: No data because a new experiment needs to be run (usually happens when they run an exp, enable a counter, and try and sample the previous experiments)

NVPM_ERROR_EXPERIMENT_RUNNING: Cannot sample while the experiment is running

Misc functions:

```
UINT NVPMGetExtendedError()
```

Appendix D. 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 Direct3D 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

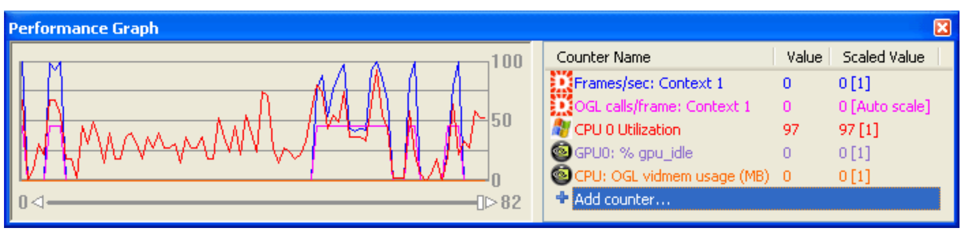
Appendix E. Accessing NVPerfKit in gDEBugger

gDEBugger is an OpenGL and OpenGL ES debugger and profiler which helps you save precious debugging time and boosts your application performance. This tool is available from our partners at Graphic Remedy; a trial version is part of the NVPerfKit installation package.

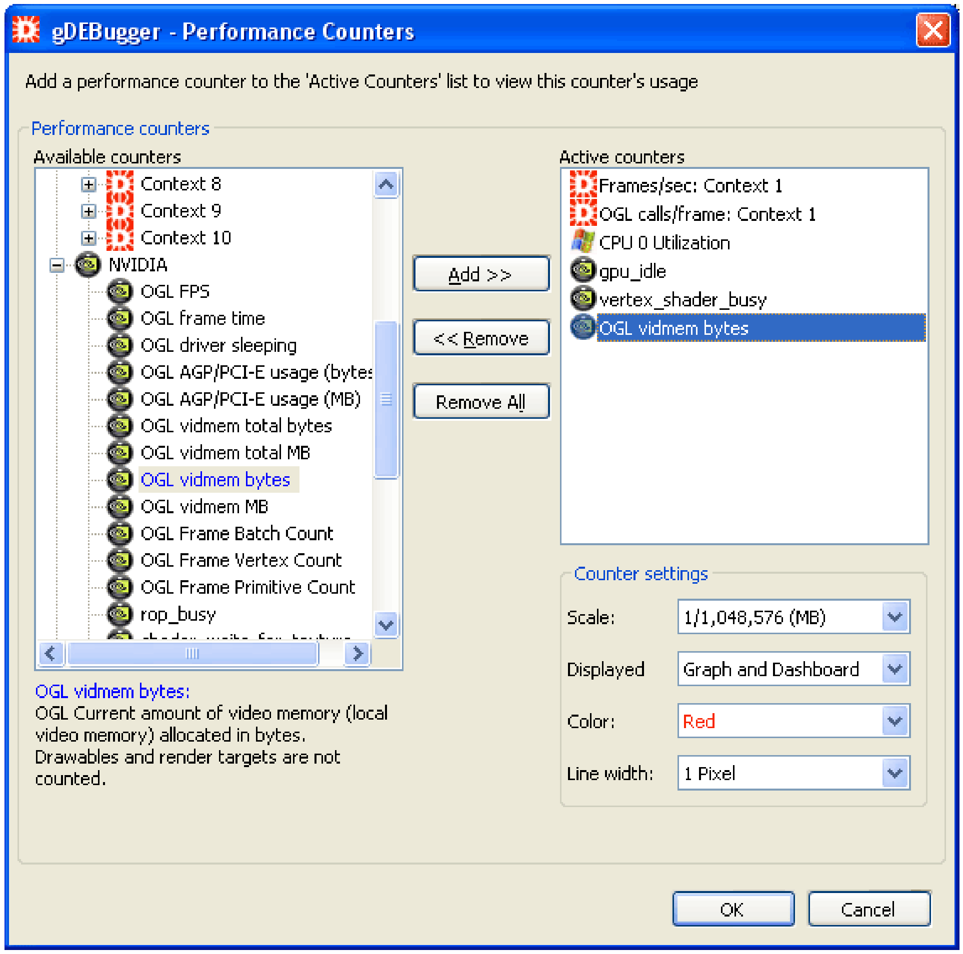
This section explains how to access NVPerfKit's performance counters through gDEBugger.

Accessing GPU Performance Counters

gDEBugger is fully integrated with NVPerfKit. This provides gDEBugger with the ability to display, in real time, the NVIDIA graphics system performance metrics in the Performance Graph view.



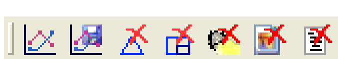
Double clicking on an item in the list opens the Performance Counters dialog where you can add new counters and set the attributes of each counter.



Note: there is no need to enable the counters in the NVDevCPL.

Performance Analysis Toolbar

The gDEBugger Performance Analysis toolbar enables you to pinpoint application performance bottlenecks quickly and easily. The toolbar contains commands which allow you to disable stages of the graphics pipeline one by one. These commands include: eliminate all OpenGL draw commands, force single pixel view port, render using no lights, force 2x2 stub textures and force a stub fragment shader. If the performance metrics improves when a certain stage has been turned off, then you have found a graphics pipeline bottleneck!



Saving Performance Data Counters in a File

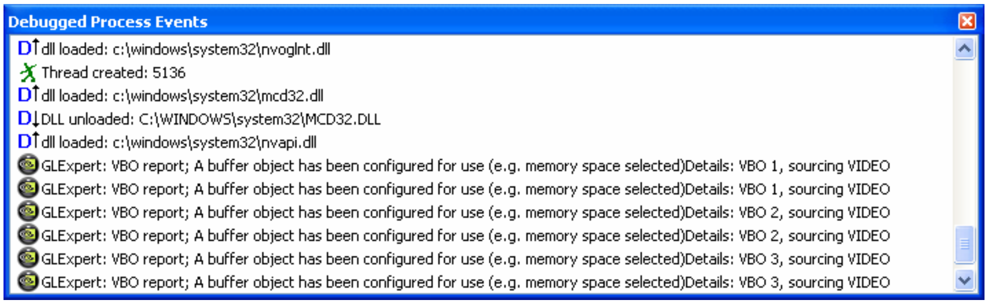
The performance data can be saved in a file (.csv). Saving performance data in a file enables you to compare performance tests for your application using different hardware and driver configurations or to perform regression tests (compare the performance of two versions of your application).

NVIDIA GLExpert and gDEBugger integration

The NVIDIA GLExpert integration enables you to receive all GLExpert reports in gDEBugger. It also enables you to break the application run whenever a GLExpert report is triggered by the debugged application and receive the call stack and source code that caused the GLExpert report.

GLExpert Settings dialog allows you configure all the NVIDIA GLExpert driver reports directly from gDEBugger.

gDEBugger will display all NVIDIA GLExpert reports in the Process Events view whenever they are reported.



Note: the gDEBugger "NVIDIA GLExpert Settings" dialog affects only the debugged process, unlike the NVIDIA Developer Control Panel, which has system wide effect.

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