GPU Performance Optimization with NVPerfHUD

NVPerfHUD 3.1

Heads-Up Display for Performance Analysis

DU-01231-001_v01
June 2005

DEVELOPMENT

Table of Contents

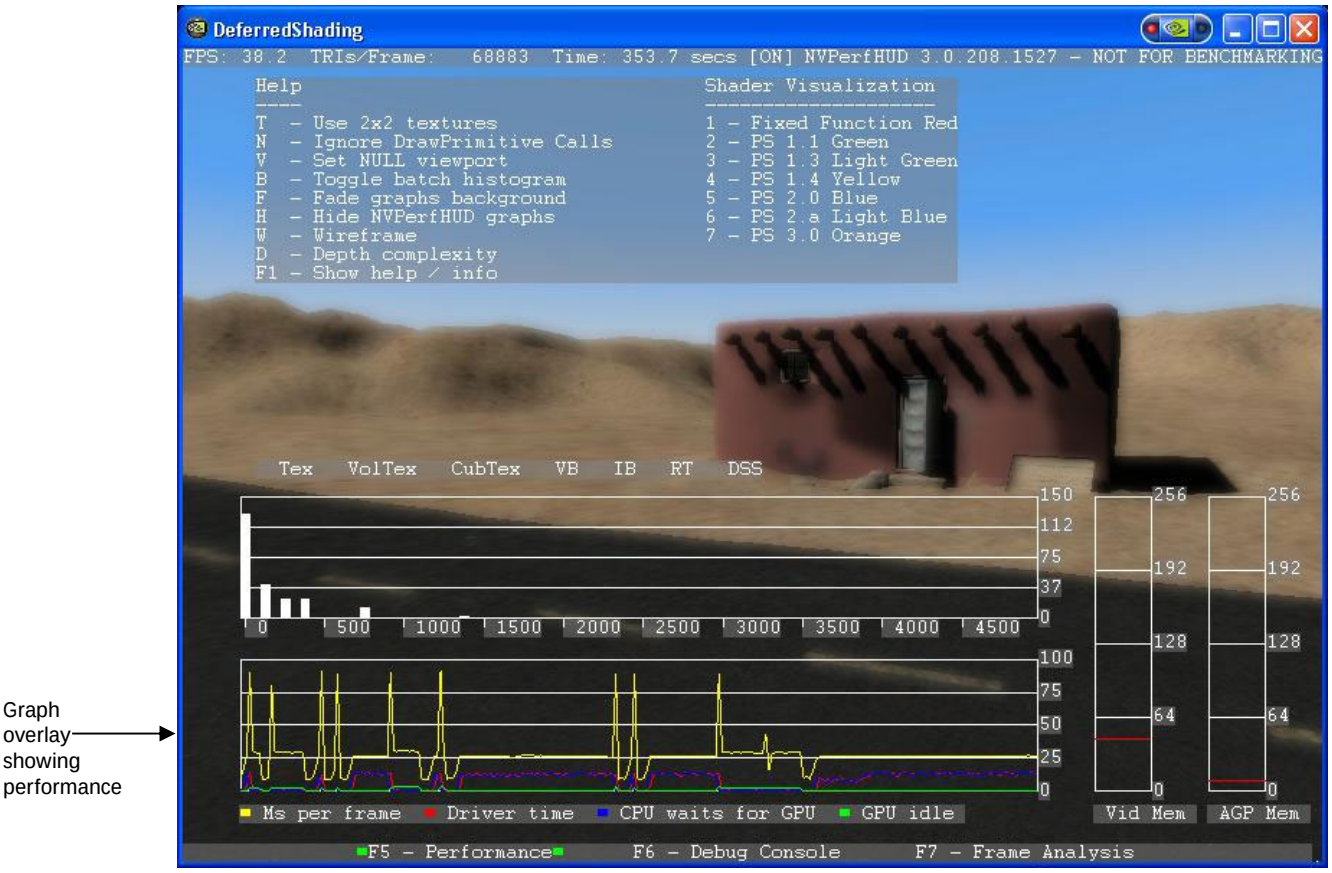
List of Figures

Chapter 1. About NVPerfHUD

Modern graphics processing units (GPUs) generate images through a pipelined sequence of operations. A pipeline runs only as fast as its slowest stage, so tuning graphical applications for optimal performance requires a pipeline-based approach to performance analysis. NVPerfHUD helps improve the overall performance of your application by identifying the slowest stage in your graphics pipeline.

NVPerfHUD allows you to analyze application performance one stage at a time and displays real time statistics that can be used to diagnose performance bottlenecks and functional problems at any stage of a Direct3D 9 application.

When you enable NVPerfHUD, a graphic overlay is displayed on top of your Direct3D application, as shown in Figure 1 below.



Graph overlay showing performance

Figure 1. Direct3D Application with NVPerfHUD Enabled

1.1. System Requirements

- Any NVIDIA GPU (GeForce 3 or better)
  GeForce 6 Series or better recommended
  Older GPUs supported with reduced functionality
- NVIDIA display drivers 77.72 or later
- Microsoft DirectX 9.0c
- Windows XP

1.2. Recommended Reading

- NVIDIA Developer Web Site
  http://developer.nvidia.com
  - *Balancing the Graphics Pipeline for Optimal Performance* – whitepaper [Link]
  - *NVIDIA GPU Programming Guide* – all the latest tips and tricks [Link]
  - NVIDIA FX Composer – shader development environment [Link]
  - NVShaderPerf – shader performance analysis utility [Link]
  - NVIDIA SDK – hundreds of code samples & effects [Link]
  - NVTriStrip – creates strips that are vertex cache aware [Link]
- GPU Gems: *Programming Techniques, Tips, and Tricks for Real-Time Graphics* [Link]
  Several of the performance-related chapters are particularly helpful
- GPU Gems 2: *Programming Techniques for High-Performance Graphics and General-Purpose Computation* [Link]
  Microsoft DirectX web site [Link]
- Microsoft Developer Network (MSDN) web site [Link]
  Search for "performance" and "optimization"
- Microsoft DirectX SDK documentation – in the Start menu after installation

Chapter 2. Getting Started

NVPerfHUD uses special performance monitoring routines in the display driver that collect metrics directly from the GPU and within the driver itself. NVPerfHUD also uses API interception to collect various metrics and interact with your application. These instrumentation techniques are required for NVPerfHUD to function properly, and introduce some small additional overhead (less than 7%).

When you use NVPerfHUD to launch your Direct3D application, you will see the NVPerfHUD user interface displayed on top of your DirectX graphics. An activation hotkey allows you to switch between interacting with your application and interacting with NVPerfHUD.

Enable your application for NVPerfHUD analysis and follow the Quick Start procedures below to get up and running with NVPerfHUD.

2.1. Prepare Your Application

NVPerfHUD is a powerful performance analysis tool that helps you understand the internal functions of your application. To ensure that unauthorized third parties do not analyze your application without your permission, you must make a minor modification to enable NVPerfHUD analysis. Additional information about making your application work well with NVPerfHUD is detailed in the Troubleshooting section at the end of this document.

Note: Be sure you disable NVPerfHUD analysis in your application before you ship. Otherwise, anyone will be able use NVPerfHUD on your application!

One of the first things you do when setting up your graphics pipeline is call the Direct3D CreateDevice() function to create your display device. In your application it probably looks something like this:

```
HRESULT Res;
Res = g_pD3D->CreateDevice(   D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
                              hWnd, D3DCREATE_HARDWARE_VERTEXPROCESSING,
                              &d3dpp, &g_pd3dDevice );
```

When your application is launched by NVPerfHUD, a special **NVIDIA NVPerfHUD** adapter is created. Your application can give NVPerfHUD permission to analyze it by selecting this adapter. In addition, since some applications might select the **NVIDIA NVPerfHUD** adapter ID unintentionally and expose themselves to unauthorized analysis, you must select "D3DDEVTYPE_REF" as the device type. Your application will not actually use the reference rasterizer as long as you have selected the NVPerfHUD adapter.

A minimal code change that will enable NVPerfHUD analysis in your application would be something like this:

```
HRESULT Res;
Res = g_pD3D->CreateDevice( g_pD3D->GetAdapterCount()-1,
                            D3DDEVTYPE_REF, hWnd,
                            D3DCREATE_HARDWARE_VERTEXPROCESSING,
                            &d3dpp, &g_pd3dDevice );
```

Using the last adapter (by calling **GetAdapterCount()-1** as shown above) assumes that the **NVIDIA NVPerfHUD** adapter identifier created by NVPerfHUD will be the last in the list.

Note: Selecting the"NVIDIA NVPerfHUD adapter and setting the **DeviceType** flag to **D3DDEVTYPE_REF** are the only changes required to activate NVPerfHUD analysis for your application. If you change only one of these parameters, NVPerfHUD analysis will not be enabled.

As you will note, this quick and dirty implementation will cause your application to use the software reference rasterizer when the **NVIDIA NVPerfHUD** adapter is not available. To avoid this problem, we recommend that you replace the call to **CreateDevice()** in your application with the following code:

```
// Set default settings
UINT AdapterToUse=D3DADAPTER_DEFAULT;
D3DDEVTYPE DeviceType=D3DDEVTYPE_HAL;

#if SHIPPING_VERSION
// When building a shipping version, disable NVPerfHUD (opt-out)
#else
// Look for 'NVIDIA NVPerfHUD' adapter
// If it is present, override default settings
for (UINT Adapter=0;Adapter<g_pD3D->GetAdapterCount();Adapter++)
{
      D3DADAPTER_IDENTIFIER9  Identifier;
      HRESULT                 Res;

      Res = g_pD3D->GetAdapterIdentifier(Adapter,0,&Identifier);
      if (strcmp(Identifier.Description,"NVIDIA NVPerfHUD") == 0)
      {
            AdapterToUse=Adapter;
            DeviceType=D3DDEVTYPE_REF;
            break;
      }
}
```

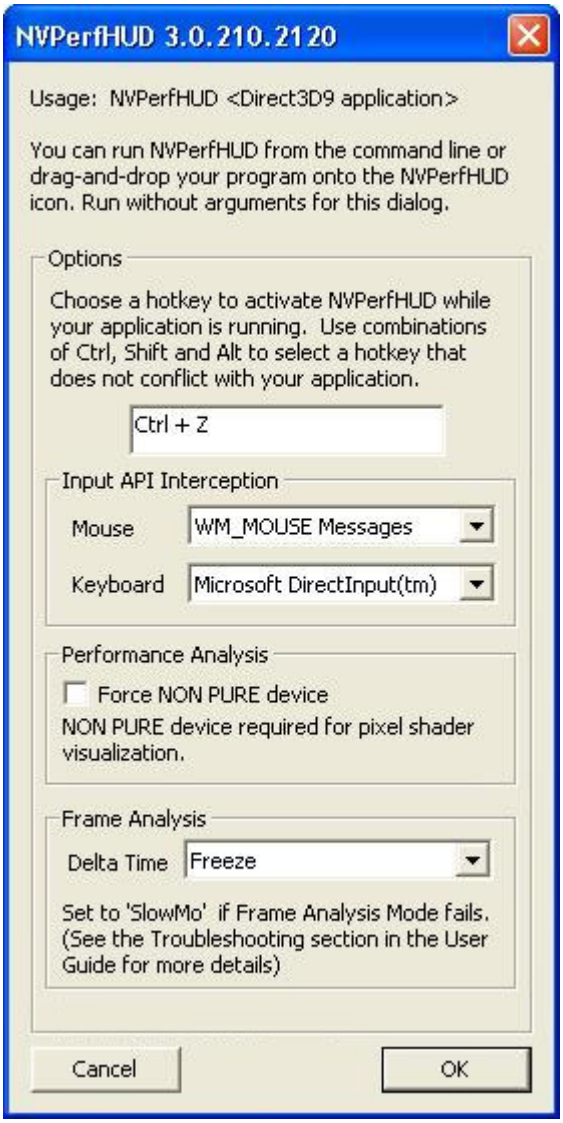
```
#endif

if (FAILED(g_pD3D->CreateDevice( AdapterToUse, DeviceType, hWnd,
            D3DCREATE_HARDWARE_VERTEXPROCESSING,
            &d3dpp, &g_pd3dDevice) ) )
{
      return E_FAIL;
}
```

This will enable NVPerfHUD analysis when you want to use it, and ensure that your application does not use the software reference rasterizer when run normally. Please follow the Quick Start procedures below to get your application up and running with NVPerfHUD.

2.2. Quick Start

When you run your application with NVPerfHUD, a default set of graphs and information are displayed on top of your application. The advanced features of NVPerfHUD are available via the activation hotkey you provide during the setup process described below.



1. **Install NVPerfHUD**
   The installer places a new icon on the desktop.
2. **Run NVPerfHUD**
   The first time you run the NVPerfHUD Launcher, a configuration dialog is displayed automatically. You can see the configuration dialog at any time by running the launch without specifying an application to analyze.
3. **Select an activation hotkey**
   Make sure it does not conflict with the keys used by your application.
4. **Configure API Interception**
   Tell NVPerfHUD how it should capture your mouse and keyboard events. You will not be able to use the keyboard and/or mouse if your application uses an unsupported method.

> **Note:** When NVPerfHUD is activated using the activation hotkey, all subsequent keyboard events are intercepted by NVPerfHUD.

5. **Optional: Enable the Force NON PURE device feature of NVPerfHUD.**
   The **Force NON PURE device** check box must be checked if your application is using a PURE

device. If your application is using a PURE device and the box is *not* checked, you are not able to use the Frame Analysis Mode or pixel shader visualization in the Performance Mode.

6. **Drag-and-drop your application onto the NVPerfHUD desktop icon.** Click **OK** to confirm your configuration options and then drag-and-drop your `.EXE`, `.BAT` or `.LNK` (shortcut) file onto the NVPerfHUD Launcher Icon. You can also run NVPerHUD.exe from the command line and specify the application to analyze as a command line argument. Some developers choose to create batch files or modify their IDE settings so this happens automatically.
7. **Optional: Change the "Delta Time" setting to SlowMo if your application crashes or has problems in Frame Analysis Mode.** This will help determine whether the problem is in NVPerfHUD or your application. See the Troubleshooting section for more details.

> **Note:** Access the configuration dialog at any time by running NVPerfHUD without specifying an application.

You should now see your application running with the default set of NVPerfHUD graphs and information displayed on top of your application. Use the activation hotkey you selected to interact with NVPerfHUD and then press **F1** to display the on-screen help. Use your hotkey again to return control to your application.

> **Note:** NVPerfHUD forces vertical refresh synchronization OFF by setting the PresentationInterval to D3DPRESENT_INTERVAL_IMMEDIATE, ensuring that you are able to accurately identify bottlenecks in your application.

2.3. Basic Workflow

Once NVPerfHUD is activated, you can perform graphics pipeline experiments, display graphs of performance metrics, and explore potential problems using several performance visualization modes. You can also switch from the default Performance Analysis Mode to either the Debug Console Mode or the Frame Analysis Mode. The Debug Console displays messages from the DirectX Debug runtime, warnings from NVPerfHUD and custom messages from your application. In the Frame Analysis Mode you have access to advanced state inspectors that show details for each stage in the graphics pipeline.

You can switch between these modes while NVPerfHUD is activated by pressing the following keys:

F5 **Performance Analysis**
Use timing graphs and directed experiments to identify bottlenecks.

F6 **Debug Console**
Review messages from the DirectX Debug runtime, NVPerfHUD warnings and custom messages from your application.

F7 **Frame Analysis**
Freeze the current frame and step through it one draw call at a time, drilling down to investigate the setup for each stage of the graphics pipeline using the State Inspectors described below.

Chapter 3.
Performance Analysis Mode

This chapter teaches you how to interpret the information displayed by NVPerfHUD in Performance Analysis Mode and how to use targeted performance experiments to identify and optimize performance bottlenecks in your application.

Figure 2 shows the graphs and overlays available in Performance Analysis Mode.

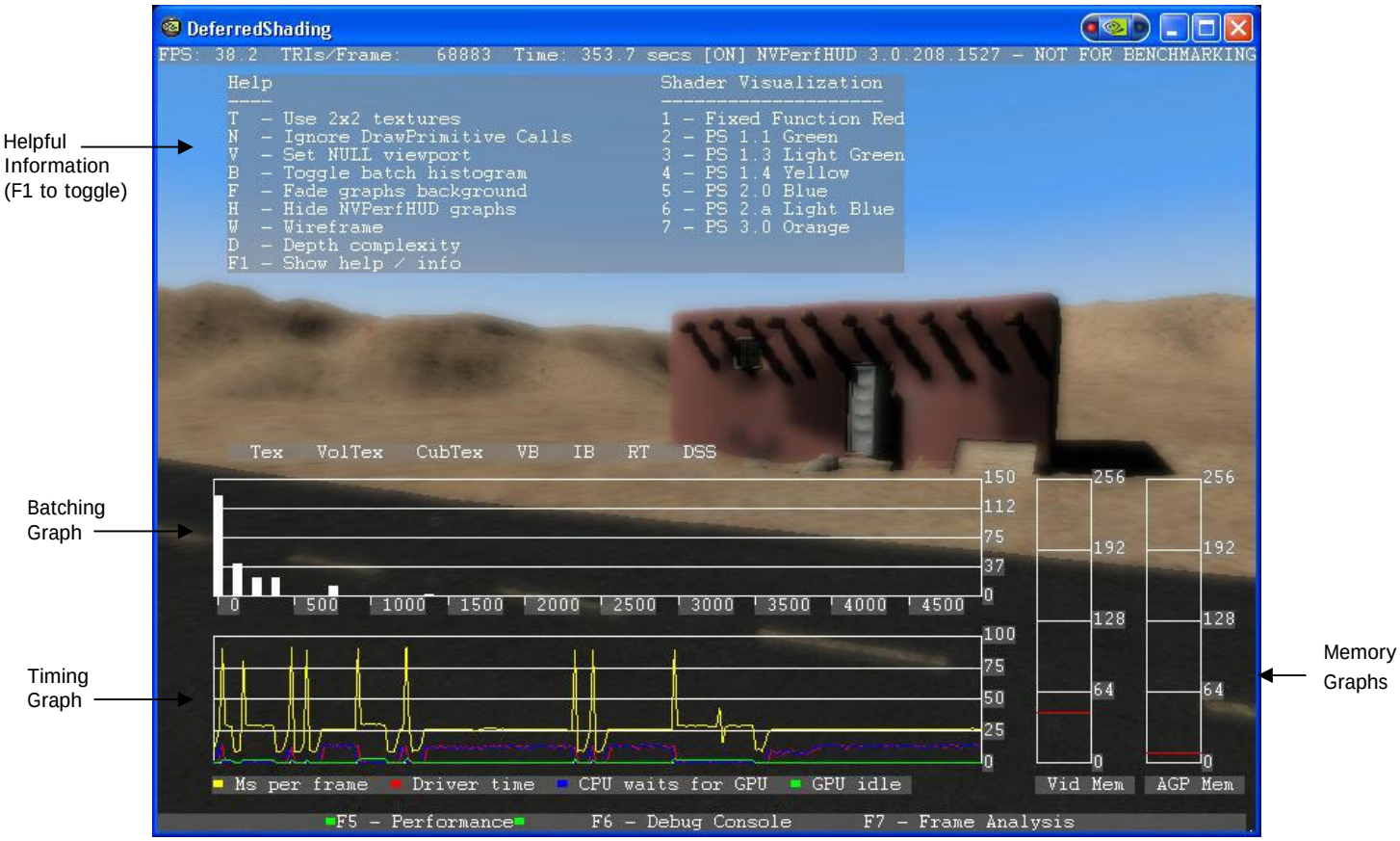


Figure 2. NVPerfHUD Performance Analysis Mode

3.1. Performance Graphs

NVPerfHUD displays several graphs and basic performance metrics by default when you first start your application.

Additional graphs and information can be displayed as needed, using the activation hotkey and the following options:

- **F1** Cycle display of helpful information
- **B** Toggle display of batch size histogram
- **F** Fade the background to improve graph readability
- **H** Hide graphs
- **W** Wireframe
- **D** Depth Complexity

Note: When the DirectX Debug Runtime is enabled, the timing graphs and directed tests are disabled. The additional overhead and performance characteristics of the Debug Runtime environment make it an inappropriate configuration for performance analysis. A warning message is displayed, but the draw calls graph, batch histogram and memory graphs are still functional.

3.1.1. Reading the Graphs

The data displayed in this mode is divided into the following informational areas:

FPS and Triangles/Frame

Basic performance metrics are displayed on the top left corner of the screen (see Figure 2). Together these two numbers provide a measure of how quickly your application is accomplishing its workload.

Figure 3. NVPerfHUD Info Strip (top)

Scrolling Graphs

These graphs behave like a heart rate monitor, scrolling from right to left every frame so you can see changes over time.

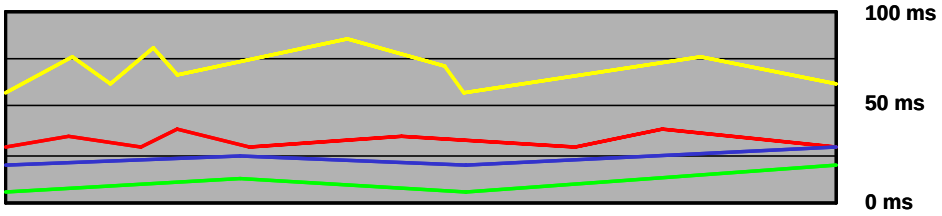


Figure 4. Performance Graphs

- **GREEN** = GPU_IDLE
  Total amount of time per frame that the GPU was idle
- **BLUE** = DRIVER_WAITS_FOR_GPU
  Accumulated elapsed time when the driver had to wait for the GPU (See Appendix A for more information on why this happens)
- **RED**= TIME_IN_DRIVER
  Total amount of time per frame that the CPU is executing driver code, including DRIVER_WAITS_FOR_GPU (**BLUE**)
- **YELLOW** = FRAME_TIME
  Total elapsed time from the end of one frame to the next - you want to keep the FRAME_TIME (**YELLOW**) line as low as possible.

FRAME_TIME	17ms	34ms	50ms	75ms	100ms
FPS	60	30	20	13	10

Note: The time gap between the FRAME_TIME (**YELLOW**) line and the TIME_IN_DRIVER (**RED**) line is the time consumed by application logic and the OS.

You might see occasional spikes caused by some operating system processes running in the background. The graph below shows a sudden frame rate hit that is caused by a hard disk access, texture upload, operating system context switch, etc.

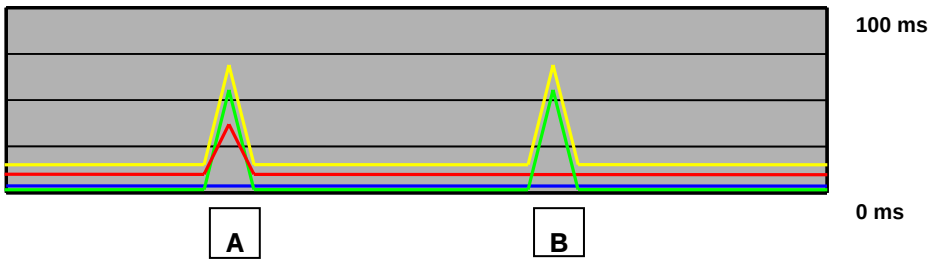


Figure 5. Occasional Spikes

This situation is normal if it is sporadic, but you should understand why the spikes are happening. If they occur regularly, your application may be performing CPU-intensive operations inefficiently.

- Type **A**:
  If the TIME_IN_DRIVER (**RED**) and FRAME_TIME (**YELLOW**) lines spike simultaneously it is likely because the driver is uploading a texture from the CPU to the GPU.
- Type **B**:
  If the FRAME_TIME (**YELLOW**) line spikes and the TIME_IN_DRIVER (**RED**) line does not, your application is likely performing some CPU-intensive operation (like decoding audio) or accessing the hard disk. This situation may also be caused by the operating system attending to other processes.

Please note that the green line may spike in either case because you are not sending data to the GPU.

Draw Primitives Graph

This graph displays the number of `DrawPrimitive, DrawPrimitiveUP` and `DrawIndexedPrimitives` (DP) calls per frame. Using this information to identify performance bottlenecks is discussed in Pipeline Experiments on the next page and in Chapter 6.

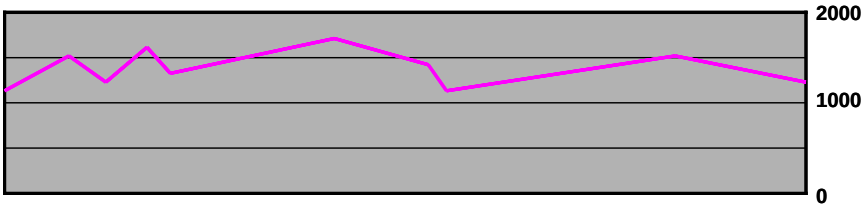


Figure 6. DP Calls Graph

Batch Size Histogram

The batch size graph only shows up when you turn it on by activating NVPerfHUD (using your activation hotkey) and pressing the **B** key. The units are number of batches. The first column on the left represents the number of batches that have between 0 and 100 triangles, the second between 100 and 200 and so on. If you use many small batches, the bar on the far left will be high.

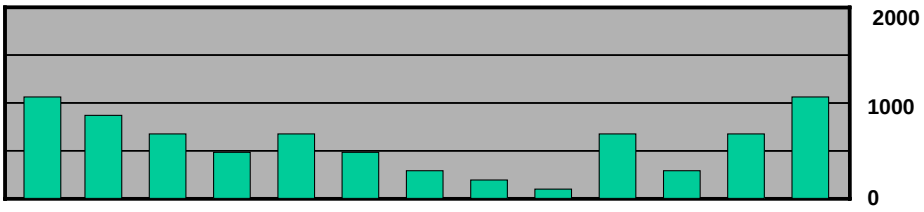


Figure 7. DP Calls Histogram

Memory Graphs

This graph displays megabytes of AGP and video memory allocated by the driver.

Figure 8. Memory Graphs

3.1.2. Resource Creation Monitor

The Resource Creation Monitor indicators blink each time a resource is created. Dynamic creation of resources in Direct3D is generally bad for performance and should be avoided whenever possible.

Figure 9. Resource Creation Monitor

The types of resource creation events monitored are:

Tex	2D textures created using CreateTexture()
VolTex	Volume textures created using CreateVolumeTexture()
CubTex	Cubemap textures created using CreateCubeTexture()
VB	Vertex buffers created using CreateVertexBuffer()
IB	Index buffers created using CreateIndexBuffer()

RT	Render targets created using CreateRenderTarget()
DSS	Depth stencil surfaces created using CreateDepthStencilSurface()

Note: Resource creation events are also logged in the Debug Console (F6) so you can see what is causing the indicators to blink.

3.2. Pipeline Experiments

Identifying performance bottlenecks requires focusing on certain stages of the graphics pipeline one stage at a time. NVPerfHUD allows you to perform the following experiments:

- **T—Isolate the texture unit**
  Force the GPU to use 2×2 textures, if the frame rate increases dramatically your application performance is limited by texture bandwidth.
- **V—Isolate the vertex unit**
  Use a 1×1 scissor rectangle to clip all rasterization and shading work in pipeline stages after the vertex unit. This approach approximates truncating the graphics pipeline after the vertex unit and can be used to measure whether your application performance is limited by vertex transforms, CPU workload and/or bus transactions.
- **N—Eliminate the GPU**
  This feature approximates having an infinitely fast GPU by ignoring all DrawPrimitive() and DrawIndexedPrimitives() calls. This approximates the frame rate your application would achieve if the entire graphics pipeline had no performance cost. Note that CPU overhead incurred by state changes is also omitted.

You can also selectively disable pixel shaders by version and visualize how they are used in your application. When a particular shader version is disabled, all shaders in that group are represented by the same color. The shader visualization options provided by NVPerfHUD are listed below:

1 — Fixed function (**RED**)
2 — 1.1 Pixel shaders (**GREEN**)
3 — 1.3 Pixel shaders (**LIGHT GREEN**)
4 — 1.4 Pixel shaders (**YELLOW**)
5 — 2.0 Pixel shaders (**BLUE**)
6 — 2.a Pixel shaders (**LIGHT BLUE**)
7 — 3.0 Pixel shaders (**ORANGE**)

Note: Shader visualization only works when a Direct3D device is created as a NON PURE device. You can force the device to be created in NON PURE device mode in the NVPerfHUD configuration settings.

Chapter 4. Debug Console Mode

This chapter describes the information available to you in the Debug Console mode. The Debug Console screen is shown in Figure 10.

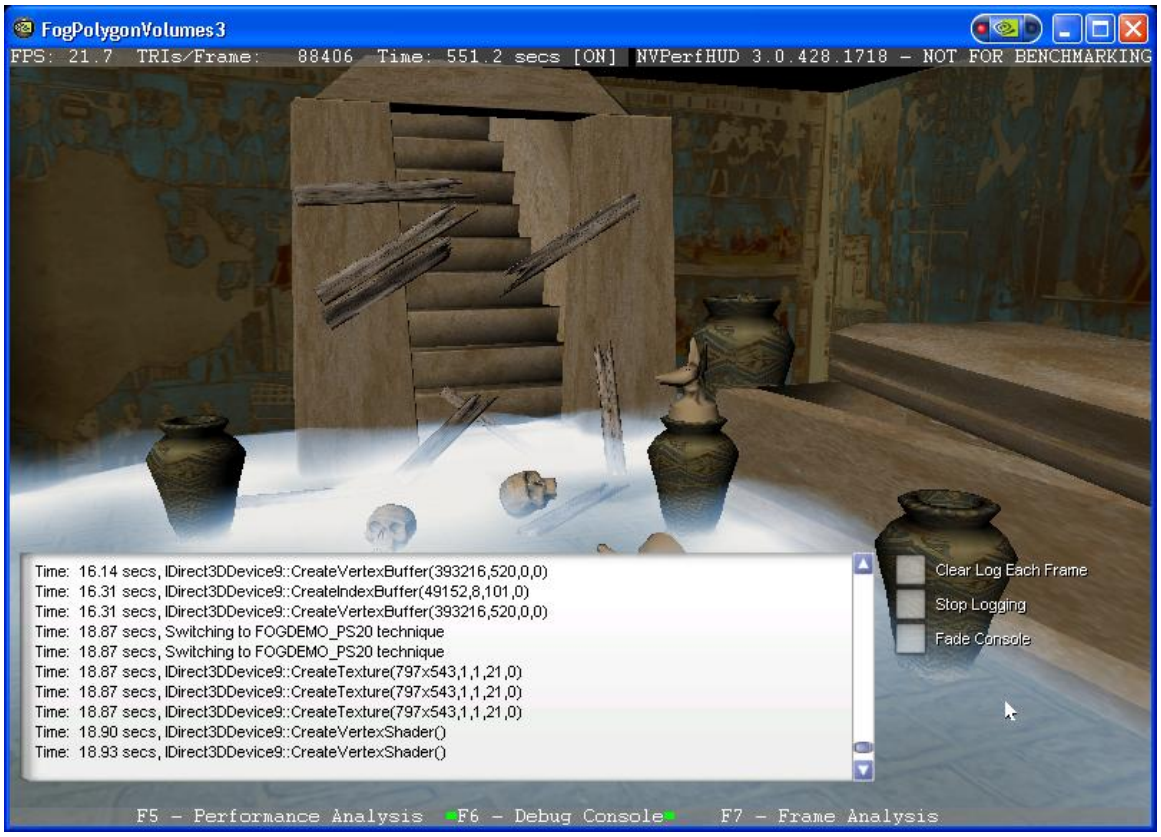


Figure 10. Debug Console Mode

The Debug Console shows all the messages reported via the DirectX Debug runtime, messages reported by your application via the OutputDebugString() function and any additional warnings or errors detected by NVPerfHUD.

Resource creation events and warnings detected by NVPerfHUD are also logged in the console window.

You can use the options below to customize how the Debug Console works:

- **C** Clear Log Each Frame
- **S** Stop Logging
- **F** Fade Console

Enabling the "Clear Log Each Frame" checkbox causes the contents of the console window to be cleared at the beginning of the frame so you only see the warnings generated by the current frame. This is useful when your application generates more warnings per frame than fit in the console window.

Enabling the "Stop Logging" checkbox causes the console to stop displaying new messages.

Chapter 5.
Frame Analysis Mode

This chapter explains how to get the most out of Frame Analysis Mode and its advanced graphics pipeline State Inspectors. Figure 11 shows the Frame Analysis Mode screen.

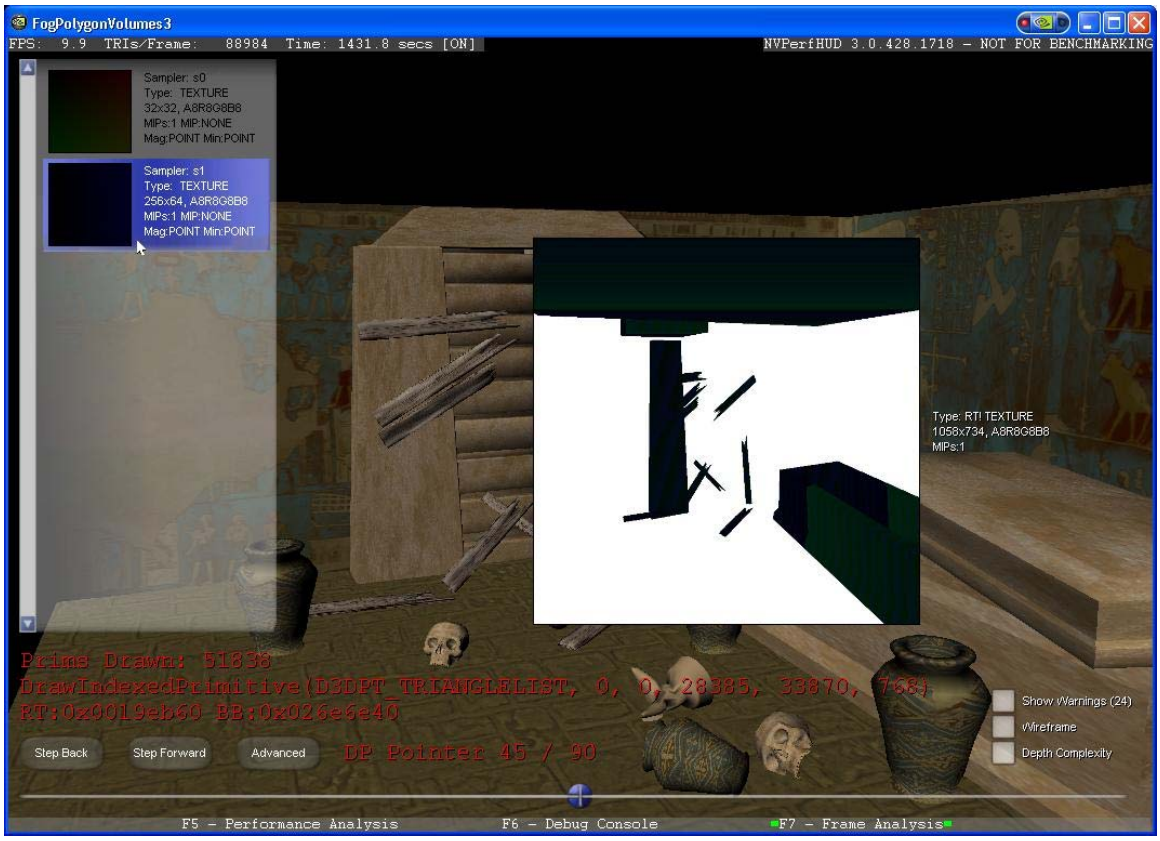


Figure 11. Frame Analysis Mode

You can use the **left/right arrow keys** to display the previous/next draw call, and the options below to configure Frame Analysis Mode:

- **A** Toggle Advanced / Simple display
- **S** Show Warnings
- **W** Wireframe
- **D** Depth Complexity

5.1. Rendering Decomposition

When you have identified a frame that has rendering artifacts, verify the order in which your scene gets drawn or learn more about what is causing the warnings, you can use Frame Analysis Mode to navigate to any Draw Call within a frame. When you switch to Frame Analysis Mode, NVPerfHUD stops the clock for your application and you can perform in-depth analysis of the current frame while it is frozen. If your application uses frame-based animation, freezing time will have no effect on animated objects.

Note: To use Frame Analysis Mode effectively, your application must behave in a way that NVPerfHUD can control it. Several requirements are described below. See the Troubleshooting section at the end of this document for additional issues.

Frame Analysis Mode requires that your application use and rely on the QueryPerformanceCounter() or timeGetTime() win32 functions. Your application must be robust in handling elapsed time (dt) calculations, especially the case where dt is zero. In other words, your program should not divide by dt.

While your application is frozen, use the **Next (right arrow)** and **Previous (left arrow)** keyboard buttons to step through all the draw calls in the frame. You can also drag the slider at the bottom of the screen back and forth or use **PgUp / PgDn** for quick navigation to a particular draw call. If mouse or keyboard event interception isn't working properly, exit your application and select an alternate API interception option in the NVPerfHUD configuration dialog.

The information displayed for each draw call includes:

- Which draw call was just drawn and how many there are in total
- The function name and parameters of the last draw call. If the draw call generated a warning then the warning message is displayed as well.

5.1.1. Show Warnings

When the **Show Warnings (S)** is enabled, a list of warnings is shown in a list box at the top of the screen, including the most expensive vertex buffer (VB) locks by DP call. Clicking on a warning will jump you to the associated DP call. You can also use the **Up / Down arrows** to scroll through the list. Clicking **Next (right arrow)** or **Previous (left arrow)** while warnings are displayed moves the slider at the bottom of the screen to the next/previous draw call that caused a warning.

You should inspect each VB lock to make sure that the time spent locked is understood and as small as possible. This should help you understand where your application is spending CPU time to setup Vertex Buffers.

5.1.2. Texture Unit and RTT Information

Information displayed for each texture unit and off-screen render to texture (RTT) target includes:

- The texture stored in each texture unit and attributes for each:
  - Dimensions
  - Filtering Parameters: minification, magnification and MIP level
  - Texture Format: `RGBA8, DXT1, DXT3, DXT5`, etc…
  - Texture Target: 1D, 2D, Volume Texture, Cube Texture, NP2, etc…
- If the current draw call is rendering to an off-screen texture (RTT) the contents of that texture and its attributes will be displayed in the middle of the screen.

When more textures are used than fit on the screen, use the scroll bar to navigate through the list.

5.1.3. Visualization Options

Frame Analysis Mode supports several visualization options:

- **D Depth Complexity**: this option turns on additive blending with a reddish colored tint. The brighter the screen becomes, the more the frame buffer has been touched. Each frambuffer Read-Modify-Write (RMW) increments the color value by 8, up to a maximum of 32 RMWs. When the screen color is saturated (255), your application is overwriting to the frame buffer too often and may lead to fill limited performance bottlenecks.
- **W Wireframe**: this option forces wireframe rendering so you can examine the geometric complexity of the scene.

5.1.4. Advanced State Inspectors

Clicking on the **Advanced… (A)** button activates the advanced State Inspectors. The slider at the bottom of the screen is still available for navigation, but now the top of the screen has several buttons for each stage in the graphics pipeline. You can either click on the button or press a shortcut key to switch between Inspectors:

- **1** Index Unit – fetches vertex data
- **2** Vertex Shader – executes vertex shaders
- **3** Pixel Shader – executes pixel shaders
- **4** Raster Operations – post-shading operations in the frame buffer

> **Note:** For details regarding the state information displayed in each of the NVPerfHUD State Inspectors, refer to the documentation that is installed with the latest version of the DirectX SDK.

Clicking on each stage shows you detailed information about what is happening in that stage during the current draw call. The following sections describe the information displayed by each of the State Inspectors.

5.2. Index Unit State Inspector

When this State Inspector is selected, information about the Index Unit during the current draw call is displayed.

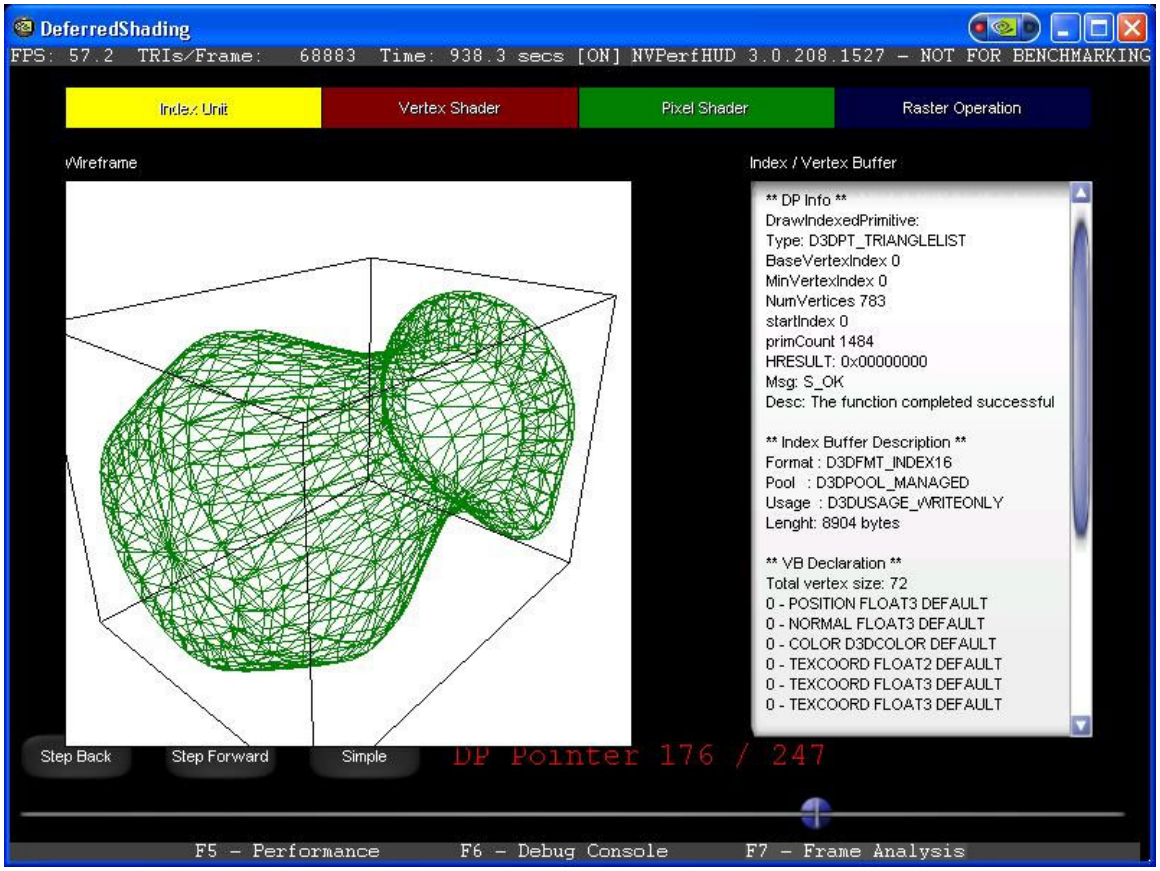


Figure 12. Index Unit State Inspector

In the center of the screen, a rotating wireframe rendering of the geometry associated with this draw call is displayed inside a bounding box.

Next to this, a list box reports all the information used to fetch the vertex data for this draw call, including:

- Draw call parameters and return flags
- Index and Vertex buffer formats, sizes, etc.
- FVFs

Using the Index Unit State Inspector you should look at the wireframe rendering to verify that the batch your application sent is correct. For example, when doing

matrix palette skinning and the rendering is corrupted, you should verify that the reference posture in the vertex buffer/index buffer is correct. If the reference posture is correct, then any rendering corruption of this geometry in your scene is probably caused by the vertex shader or bad vertex weights

You should also verify that the format of the indices is correct, making sure that 16-bit indices are used whenever applicable.

5.3. Vertex Shader State Inspector

When this State Inspector is selected, information about the Vertex Shader during the current draw call is displayed.

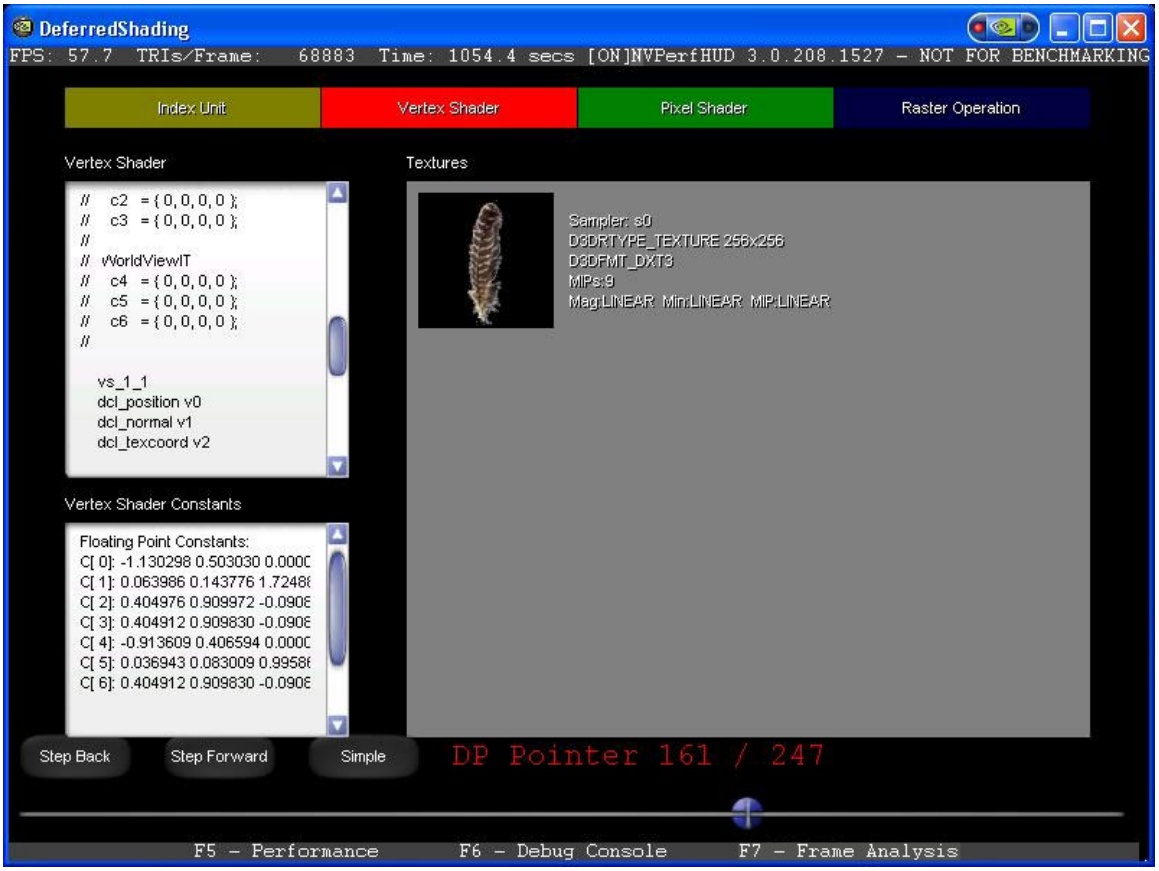


Figure 13. Vertex Unit State Inspector

The vertex shader program, along with any constants and textures used by it, are displayed for inspection. When a vertex shader uses the address register (e.g. matrix palette skinning) all the constants are shown. Information about each of the texture samplers is also displayed for reference. Use the + / – keys to magnify the textures displayed.

Using the Vertex Shader State Inspector you should:

- Verify that the expected vertex shader is applied for the current draw call
- Verify that the constants are not passing **#NAN** or **#INF**

5.4. Pixel Shader State Inspector

When this State Inspector is selected, information about the Pixel Shader during the current draw call is displayed.

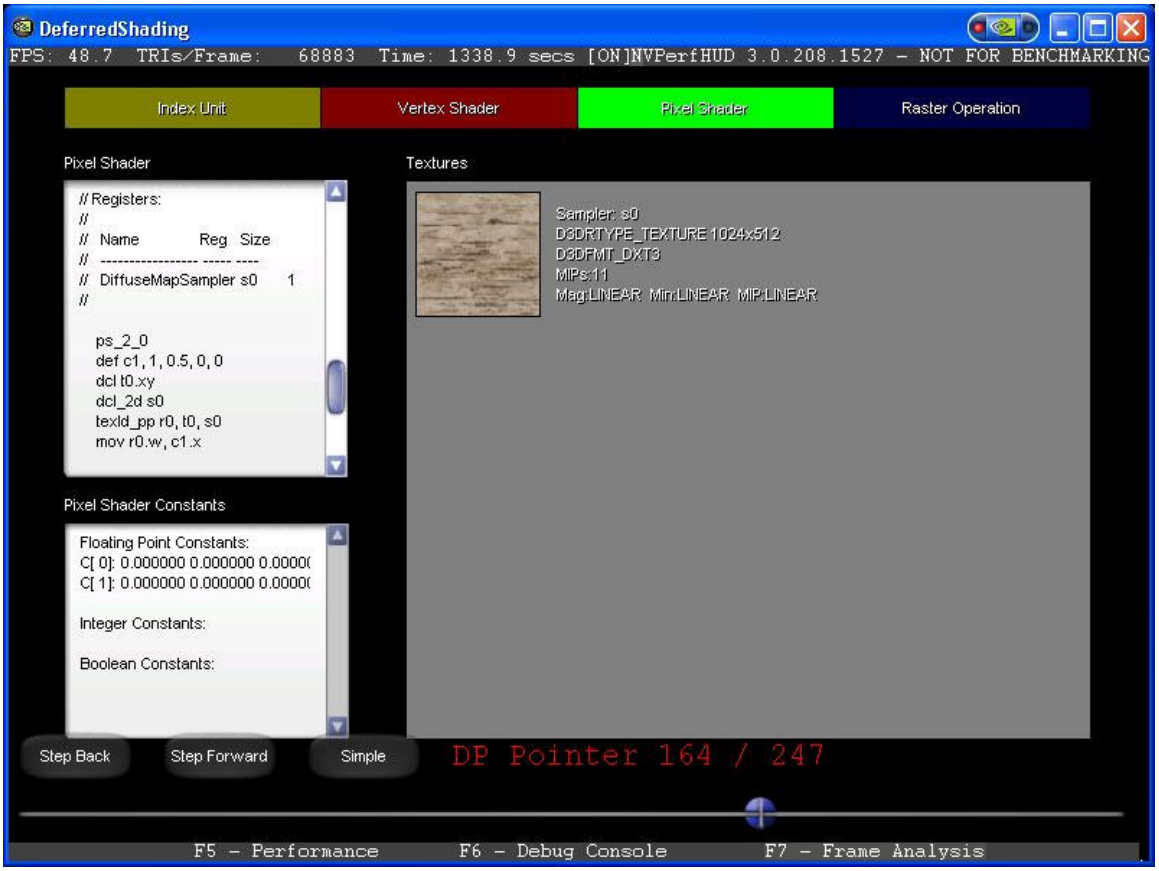


Figure 14. Pixel Shader State Inspector

The pixel shader program, along with any constants and textures used by it, are displayed for inspection. Information about each of the texture samplers is also displayed for reference. Use the + / – keys to magnify the textures displayed.

Using the Pixel Shader State Inspector, you should:

- Verify that the expected pixel shader is applied for the current draw call
- Verify that the constants are not passing **#NAN** or **#INF**
- Verify that the textures and render-to-texture textures are used correctly

5.5. Raster Operations State Inspector

When this State Inspector is selected, information about the Raster Operations (ROP) during the current draw call is displayed.

Note: For additional details regarding the state information displayed in the Raster Operations State Inspector, please see the documentation that is installed with the latest version of the DirectX SDK.

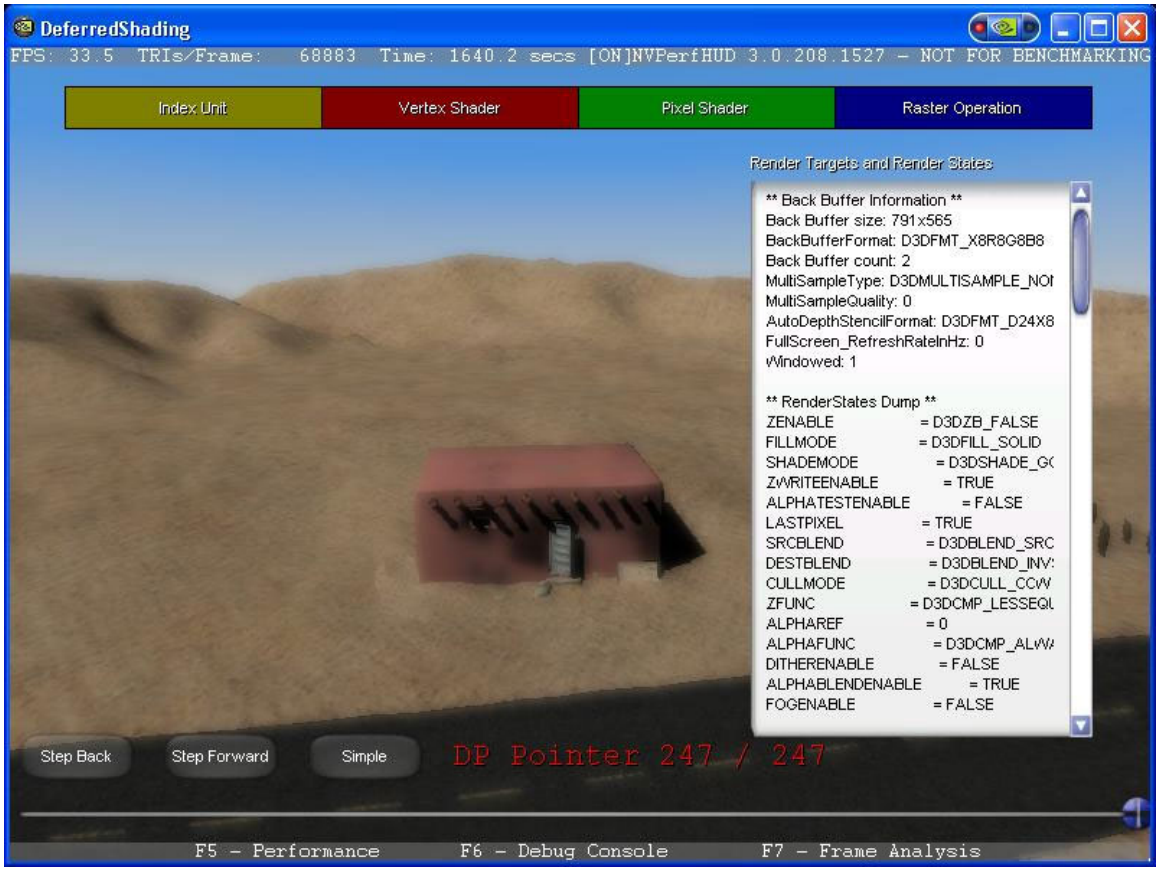


Figure 15. Raster Operations State Inspector

Information about the post-shading raster operations for this draw call is displayed for inspection. The information includes:

- Render target format
- Back buffer format
- Render states that can cause frame buffer processing to be expensive include:
  - **`Zenable`** – **`Z`** compare operation
  - **`Fillmode`** – rasterization mode
  - **`ZWriteEnable`** – Writes **`Z`** in the depth buffer or not
  - **`AlphaTestEnable`** – is alpha test enabled

- **SRCBLEND** and **DSTBLEND** – what is the blend operation
- **AlphablendEnable** – is the application blending in the frame buffer
- **Fogenable** – is fog enabled
- **Stencil enable** – is writing to stencil buffer enabled
- **StencilTes**t….

When you are using the Raster Operations State Inspector, you should:

- Verify that the back buffer format does indeed contains an alpha component when the blending doesn't work right
- Verify that drawing opaque objects is not done with **blendEnable**

Chapter 6. Analyzing Performance Bottlenecks

6.1. Graphics Pipeline Performance

Over the past few years, hardware-accelerated rendering pipelines have substantially increased in complexity, bringing with it increasingly complex and potentially confusing performance characteristics. What used to be a relatively simple matter of reducing CPU cycles of inner loops in your renderer to improve performance, has now become a cycle of determining bottlenecks and systematically optimizing them. This repeating process of *Identification* and *Optimization* is fundamental to tuning a heterogeneous multiprocessor system, with the driving idea being that a pipeline is, by definition, only as fast as its slowest stage. The logical conclusion is that, while premature and unfocused optimization of a single processor system can lead to only minimal performance gains, in a multi-processor system it very often leads to *zero* gains.

Working hard on graphics optimization and seeing zero performance improvement is no fun. The goal of this chapter is to explain how NVPerfHUD should be used to identify performance bottlenecks and save you from wasting time.

6.1.1. Pipeline Overview

At the highest level, the pipeline is broken into two parts: the CPU and GPU. Figure 16 shows that within the GPU there are a number of functional units operating in parallel, which can essentially be viewed as separate special purpose processors, and a number of spots where a bottleneck can occur. These include vertex and index fetching, vertex shading (transform and lighting), pixel shading, texture loading, and raster operations (ROP).

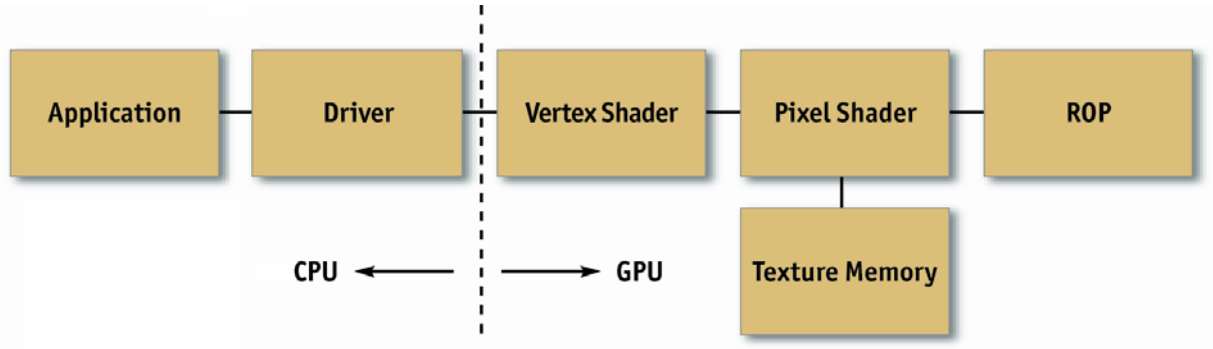


Figure 16. Pipeline Overview

6.1.2. Methodology

Optimization without proper bottleneck identification is the cause of much wasted development effort, and so we formalize the process into the following fundamental identification and optimization loop:

- **Identify**
  For each stage in the pipeline use an NVPerfHUD experiment to isolate that stage. If performance varies, you've found a bottleneck. You can also try implementing similar experiments on your own by modifying the application to vary its workload,
- **Optimize**
  Given the bottlenecked stage, reduce its workload until performance stops improving, or you achieve your desired level of performance.
- **Repeat**
  Repeat steps 1 and 2 until the desired performance level is reached

6.2. Identifying Bottlenecks

Locating the bottleneck is half the battle in optimization, as it enables you to make intelligent decisions on focusing your actual optimization efforts. Figure 17 shows a flow chart depicting the series of steps required to locate the precise bottleneck in your application. Note that we start at the back end of the pipeline, with the frame buffer operations (also called raster operations) and end at the CPU. Note also that, while any single primitive (usually a triangle), by definition, has a single bottleneck, over the course of a frame the bottleneck most likely changes, so modifying the workload on more than one stage in the pipeline often influences performance. For example, it's often the case that a low polygon skybox is bound by pixel shading or frame buffer access, while a skinned mesh that maps to only a few pixels on screen is bound by CPU or vertex processing. For this reason, it often helps to vary workloads on an object-by-object, or material-by-material, basis.

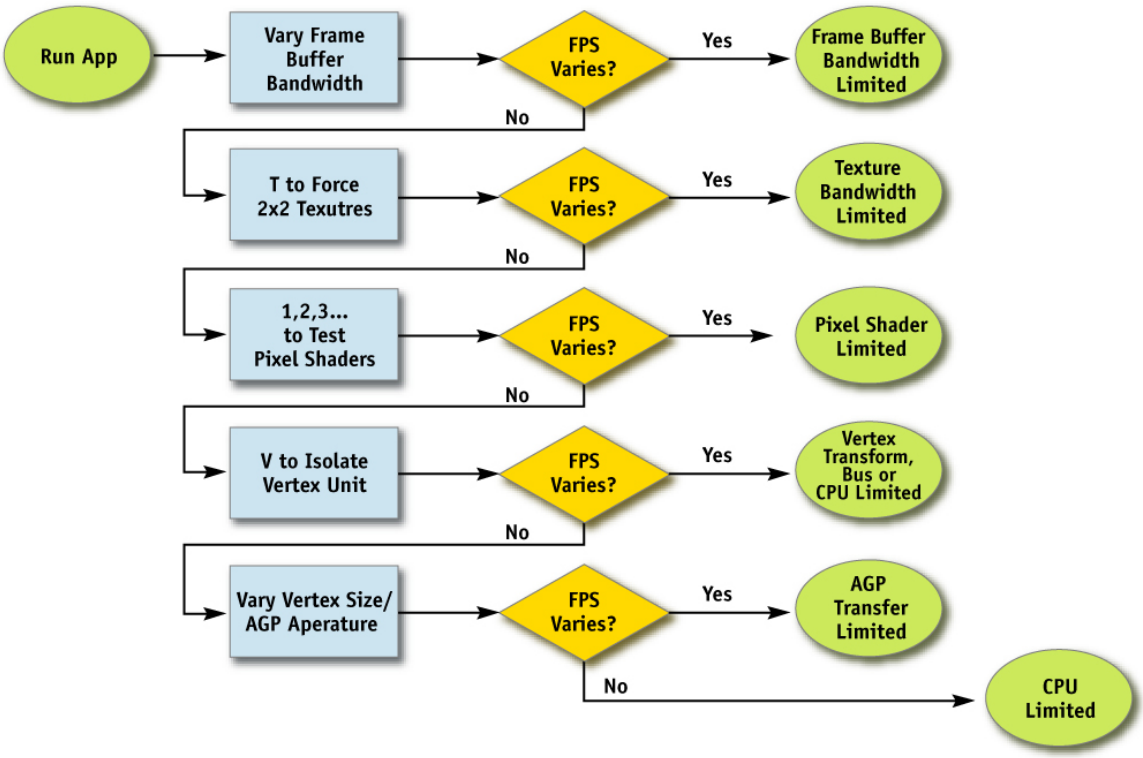


Figure 17. Identifying Bottlenecks

Note: If you suspect that your application is CPU limited, you can press N at any time. If the frame rate of your application does not dramatically increase, your application is CPU limited.

6.2.1. Raster Operation Bottlenecks

The backend of the pipeline, often called the ROP, is responsible for reading and writing depth and stencil, doing the depth and stencil comparisons, reading and writing color, and doing alpha blending and testing. As you can see, much of the ROP workload taxes the available frame buffer bandwidth.

The best way to test if your application is frame buffer bandwidth bound is to vary the bit depths of the color and/or depth buffers. If reducing your bit depth from 32-bit to 16-bit significantly improves your performance, then you are definitely frame buffer bandwidth bound.

Frame buffer bandwidth is a function of GPU memory clock speed.

6.2.2. Texture Bandwidth Bottlenecks

Texture bandwidth is consumed anytime a texture fetch request goes out to memory. Although modern GPUs have texture caches designed to minimize extraneous memory requests, they obviously still occur and consume a fair amount of memory bandwidth.

Pressing **T** while NVPerfHUD is activated replaces all textures in your application with a 2×2 texture. This emulates a much faster texture-fetch with much better texture cache coherence. If this causes performance to improve significantly, you are bound by texture bandwidth.

Texture bandwidth is also a function of GPU memory clock speed.

6.2.3. Pixel Shading Bottlenecks

Pixel shading refers to the actual cost of generating a pixel, with associated color and depth values. This is the cost of running the "pixel shader". Note that pixel shading and frame buffer bandwidth are often lumped together under the heading "fillrate" since they are both a function of screen resolution, but they are two distinct stages in the pipeline, and being able to tell the difference between the two is critical to effective bottleneck identification and optimization.

The first step in determining if pixel shading is the bottleneck is using NVPerfHUD to substitute all the shaders with very simple shaders. To do this, disable the pixel shader profiles one at a time using 1, 2, 3, … in Performance Analysis Mode and watch for changes in frame rate. If this causes performance to improve, the culprit is most likely pixel shading.

The next step is to figure out which shaders are the most expensive using NVShaderPerf or the Shader Perf panel in FX Composer. Remember that pixel shader cost is per-pixel, so an expensive shader that only affects a few pixels may not be as much of a performance problem as an expensive shader that affects many pixels. Basically, shader_cost = cost_per_pixel * number_of_pixels_affected. Focus your performance optimization efforts on the shaders with the highest cost.

FX Composer includes a number of shader optimization tutorials that may be helpful in reducing the performance cost of your shaders.

Note: The latest versions of FX Composer and NVShaderPerf will help you analyze shader performance across the entire family of NVIDIA GPUs. Both are available from http://developer.nvidia.com.

6.2.4. Vertex Processing Bottlenecks

The vertex transformation stage of the rendering pipeline is responsible for taking a set of vertex attributes (e.g. model-space positions, vertex normals, texture coordinates, etc.) and producing a set of attributes suitable for clipping and rasterization (e.g. homogeneous clip-space position, vertex lighting results, texture coordinates, etc.). Naturally, performance in this stage is a function of the work done per-vertex, along with the number of vertices being processed.

Determining if vertex processing is your bottleneck is a simple matter of running your application with NVPerfHUD and pressing the **V** key to isolate the vertex unit. If the resulting frame-rate is roughly equivalent to the original frame-rate, then your application is limited by vertex/index buffer AGP transfers, vertex shader units, or inefficient locks and resulting GPU stalls.

Note: To rule out inefficient locks you should run the app in the Direct3D debug run-time and verify that no errors or warnings are generated.

6.2.5. Vertex and Index Transfer Bottlenecks

Vertices and indices are fetched by the GPU as the first step in the GPU part of the pipeline. The performance of vertex and index fetching can vary depending on where the actual vertices and indices are placed, which is usually either system memory, which means they will be transferred to the GPU over a bus like AGP or PCI-Express, or local frame buffer memory. Often, on PC platforms especially, this decision is left up to the device driver instead of your application, though modern graphics APIs allow applications to provide usage hints to help the driver choose the correct memory type. Refer to the NVIDIA GPU Programming Guide [Link] for advice about using Index Buffers and Vertex Buffers optimally in your application.

Determining if vertex or index fetching is a bottleneck in your application is a matter of modifying the vertex format size.

Vertex and index fetching performance is a function of the AGP/PCI-Express rate if the data is placed in system memory, and a function of the memory clock if placed in local frame buffer memory.

6.2.6. CPU Bottlenecks

There are two ways to know if your application performance is limited by the CPU.

One quick way to tell that your application performance is limited by the CPU is to watch the GPU Idle (green) line in the NVPerfHUD timing graph. If the green line is flat at the bottom of the graph, the GPU is never idle. If the green line jumps up from the bottom of the graph, it means that the CPU is not submitting enough work to the GPU.

You can also tell that your application is limited by the CPU by pressing **N** to isolate the GPU. When you do this, NVPerfHUD forces the Direct3D runtime to ignore all the DP calls. The resulting frame rate approximates the frame rate your application would have with an infinitely fast GPU and display driver.

If your application performance is CPU limited (too busy to feed the GPU), it may be caused by:

- Too many DP calls – there is driver overhead for each call. See Figure 18 and the accompanying description below.
- Demanding application logic, physics, etc. - the gap between the FRAME_TIME (YELLOW) line and TIME_IN_DRIVER (**RED**) line represents the amount of time the CPU was dedicated to your application.
- Loading or allocating resources – for example, the driver must process each texture, so the GPU may finish all pending work while the driver is busy loading many (large) textures. Watch for these events in the Resource Creation Monitor described in Section 3.1.2.

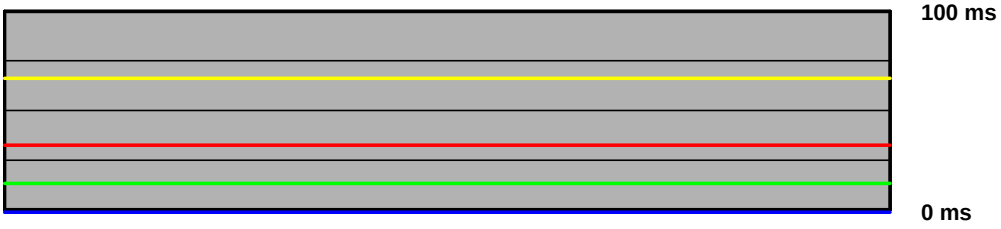


Figure 18. Too Many Calls to the Driver

Figure 18 shows a typical case where your application is doing too many calls to the driver, double check this scenario by also looking at the graph that reports number of batches.

6.3. Optimization

Now that we have identified the bottleneck, we must optimize that particular stage in order to improve application performance. The following optimization suggestions are organized by the stage for which they may improve overall application performance.

6.3.1. CPU Optimizations

Applications performance may CPU-limited due to complex physics or AI. Your performance may also suffer due to poor batch size and resource management. If you've found that your application is CPU-limited, try the following suggestions to reduce CPU work in the rendering pipeline.

6.3.2. Reduce Resource Locking

Resources can be either textures or vertex buffers. Anytime you perform a synchronous operation which demands access to a GPU resource, there is the potential to massively stall the GPU pipeline, which costs both CPU and GPU cycles. CPU cycles are wasted because the CPU must sit and spin in a loop waiting for the GPU pipeline to drain and return the requested resource. GPU cycles are then wasted as the pipeline sits idle and has to refill.

This can occur any time you:

- Lock or read from a surface you were previously rendering to.
- Write to a surface the GPU is reading from, like a texture or a vertex buffer.

Locking a busy resource contributes to raising the DRIVER_WAITS_FOR_GPU (BLUE) line. See Appendix A for more information about things that cause the driver to wait for the GPU.

To rule out inefficient locks you should run the application in the Direct3D debug run-time and verify that no errors or warnings are generated. Read more about effectively managing how you lock resources in this whitepaper:
http://developer.nvidia.com/object/dynamic_vb_ib.html

6.3.3. Minimize Number of Draw Calls

Every API function call to draw geometry has an associated CPU cost, so minimizing the number of API calls and, in particular, minimizing the number of graphics state changes minimizes the amount of CPU work used for a given number of triangles rendered.

We define a *batch* as a group of primitives rendered with a single API rendering calls such as `DrawPrimitive()` and `DrawIndexedPrimitive()` in DirectX 9. The "size" of a batch refers to the number of primitives contained in it.

You can know how well you are batching using NVPerfHUD. Pressing **B** displays a histogram showing the distribution of number of triangles per draw call per frame. Figure 19 shows an application that probably performs poorly because it has too many DP calls with a small number of primitives.

Figure 19. Many Small DP Calls

To reduce the number of DP calls, try the following:

- **If using triangle strips, use degenerate triangles to stitch together disjoint strips**. This enables you to send multiple strips, provided they share material, in a single draw call. The `NVTristrip` library is available from http://developer.nvidia.com provides source code for this.
- **Use texture pages**. Batches are frequently broken when different objects use different textures. By arranging many textures into a single 2D texture and setting your texture coordinates appropriately, you can send geometry that uses multiple textures in a single draw call. Note that this technique can have issues with mipmapping and anti-aliasing. One technique that sidesteps many of these issues is to pack individual 2D textures into each face of a cubemap. The latest version of the NVIDIA SDK includes a collection of texture atlas tools that help you create and preview texture pages.
- **Use the vertex shader constant memory as a lookup table of matrices.** Often batches get broken when many small objects share all material properties but differ only in matrix state (for example, a forest of similar trees). In these cases, you can load multiple matrices into the vertex shader constant memory and store indices into the constant memory in the vertex format for each object. Then you use this index to lookup into the constant memory in the vertex shader and use the correct transformation matrix, thus rendering N objects at once.
- **Use geometry instancing if you have multiple copies of the same mesh in your scene.** This technique allows you to draw multiple copies of the same mesh object with a single draw call and two vertex streams. Each copy or "instance" of the mesh can be drawn in a different locations, and (optionally) with different visualizations. One stream contains a single copy of the mesh to be instanced, and the other contains per-instance data (world transforms, colors, etc). Then you can issue a single draw call and tell it how many instances you'd like to draw. In general, geometry instancing is most useful when you have many low (sub 1000) poly objects because this reduces the CPU overhead of many draw calls. An example of geometry instancing (with full source code) is also included in the latest version of the NVIDIA SDK.

- **Use GPU shader branching to increase batch size.** Modern GPUs have flexible vertex and pixel processing pipelines that allow for branching inside the shader. For example, if two batches are separate because one requires a 4 bone skinning vertex shader, while the other requires a 2 bone skinning vertex shader, you could instead write a vertex shader that looped over the number of bones required, accumulating blending weights, and broke out of the loop when the weights summed to one. This way, the two batches could be combined into one. On architectures that don't support shader branching, similar functionality can be implemented, at the cost of shader cycles, by using a 4 bone vertex shader on everything, and simply zeroing out the bone weights on vertices that have fewer than 4 bone influences.
- **Defer decisions as far down in the pipeline as possible.** It's faster to use the alpha channel of your texture as a gloss factor, rather than breaking the batch to set a pixel shader constant for glossiness. Similarly, putting shading data in your textures and vertices can allow for larger batch submissions.

6.3.4. Reduce the Cost of Vertex Transfer

Vertex transfer is rarely the bottleneck in modern applications, but it's certainly not impossible for this to be a problem. If the transfer of vertices or, even less likely, indices, is the bottleneck in your application, try the following:

- **Use the fewest number of bytes possible in your vertex format**. Don't use floats for everything if bytes would suffice (for colors, for example).
- Generate potentially derivable vertex attributes inside the vertex program instead of storing them inside of the input vertex format. For example, there's often no need to store a tangent, binormal, and normal, since given any two, the third can be derived using a simple cross-product in the vertex program. This technique trades vertex processing speed for vertex transfer rate.
- **Use 16-bit indices instead of 32-bit indices.** 16-bit indices are cheaper to fetch, cheaper to move around, and take less memory.
- **Access vertex data in a relatively sequential manner.** Modern GPUs cache memory accesses when fetching vertices. As in any memory hierarchy, spatial locality of reference helps maximize hits in the cache, thus reducing bandwidth requirements.

6.3.5. Optimize Vertex Processing

Vertex processing is rarely the bottleneck on modern GPUs, but it is possible this might be a problem, depending on your usage patterns and target hardware. Try these suggestions if you find that vertex processing is the bottleneck in your application:

- **Pull out per-object computations onto the CPU.** Often, a calculation that changes once per-object or per-frame is done in the vertex shader for convenience. For example, transforming a directional light vector to eye space is sometimes done in the vertex shader, although the result of the computation only changes per-frame.
- **Optimize for the post-TnL vertex cache**. Modern GPUs have a small FIFO cache that stores the result of the most recently transformed vertices; a hit in

this cache saves all transform and lighting work, along with all work earlier in the pipeline. To take advantage of this cache, you must use indexed primitives, and you must order your vertices to maximize locality of reference over the mesh. There are freely available tools that help you with this task, including D3DX and NVTriStrip - http://developer.nvidia.com/object/nvtristrip_library.html.

- **Reduce the number of vertices processed.** This is rarely the fundamental issue, but using a simple level-of-detail scheme, like a set of static LODs, certainly helps reduce vertex processing load.
- **Use vertex processing LOD.** Along with using LODs for the number of vertices processed, try LODing the actual vertex computations themselves. For example, it is likely not necessary to do full 4-bone skinning on distant characters, and you can probably get away with cheaper approximations for the lighting. If your material is multi-passed, reducing the number of passes for lower LODs in the distance will also reduce vertex processing cost.
- **Use the correct coordinate space**. Frequently, your choice of coordinate space impacts the number of instructions required to compute a value in the vertex program. For example, when doing vertex lighting, if your vertex normals are stored in object space, and the light vector is stored in eye space, then you have to transform one of the two vectors in the vertex shader. If the light vector was instead transformed into object space once per-object on the CPU, no per-vertex transformation would be necessary, saving GPU vertex instructions.
- **Use vertex branching to "early-out" of computations.** If looping over a number of lights in the vertex shader, and doing normal, low dynamic range [0..1] lighting, you can check for saturation to one, or if you're facing away from the light, and break out of further computations. A similar optimization can occur with skinning, where you can break when your weights sum to 1 (and therefore all subsequent weights would be zero). Note that this depends on the way that the GPU implements vertex branching, and isn't guaranteed to improve performance on all architectures.

6.3.6. Speed Up Pixel Shading

If you're using long and complex pixel shaders, it is often likely that you're pixel shading bound. If you find that to be the case, try these suggestions:

- **Render depth first.** Rendering a depth-only (no color) pass before rendering your primary shading passes can dramatically boost performance, especially in scenes with high depth complexity, by reducing the amount of pixel shading and frame buffer memory access that needs to be performed. To get the full benefits of a depth-only pass, it's not sufficient to just disable color writes to the frame buffer, you should also disable all shading on pixels, including shading that affects depth as well as color (e.g. alpha test).
- **Help early-Z optimizations throw away pixel processing.** Modern GPUs have silicon devoted to not shading pixels you can't see, but these rely on knowledge of the scene up to the current point, and can be dramatically helped out by rendering in a roughly front-to-back order. Also, laying depth down first (see above) in a separate pass can help dramatically speed up subsequent passes

(where all the expensive shading is done) by effectively reducing their shaded depth complexity to one

- **Store complex functions in textures.** Textures can be enormously useful as lookup tables, with the additional benefit that their results are filtered for free. The canonical example here is a normalization cubemap, which allows you to normalize an arbitrary vector at high precision for the cost of a single texture lookup.
- **Move per-pixel work to the vertex shader.** Just as per-object work in the vertex shader should be moved to the CPU instead, per-vertex computations (along with computations that can be correctly linearly interpolated in screen-space) should be moved to the vertex shader. Common examples include computing vectors and transforming vectors between coordinate systems.
- **Use the lowest precision necessary.** APIs like DirectX 9 allow you to specify precision hints in pixel shader code for quantities or calculations that can work with reduced precision. Many GPUs can take advantage of these hints to reduce internal precision and improve performance.
- **Avoid unnecessary normalization.** A common mistake is to get overly normalization-happy and normalize every single vector every step of the way when performing a calculation. Recognize which transformations preserve length (like a transformation by an orthonormal basis) and which computations do not depend on vector length (such as a cubemap lookup).
- **Use half precision normalizes when possible.** Normalizing at half-precision is essentially a free operation on the NV4x class of GPUs. Use the 'half' type in HLSL for the vector that is to be normalized. If using ps_2_0 or higher version assembly shaders in DirectX 9, use **nrm_pp** (or use the '_pp' modifier on *all* operations for the equivalent math). While testing your HLSL shaders it is a good idea to check the generated assembly to make sure the _pp modifier is being used on the operations corresponding to normalize to ensure that you have used the 'half' data type appropriately. You can verify the assembly generated from an HLSL shader by running fxc.exe or the FX Composer Shader Perf panel. You can also use the NVShaderPerf command line utility.
- **Consider using pixel shader level-of-detail.** While not as high a bang for the buck as vertex LOD (simply because objects in the distance naturally LOD themselves with respect to pixel processing due to perspective), reducing the complexity of the shaders in the distance, along with reducing the number of passes over a surface, can reduce the pixel processing workload.
- **Make sure you are not limited by texture bandwidth.** Refer to the Reduce Texture Bandwidth section below for more information.

6.3.7. Reduce Texture Bandwidth

If you've found that your application is memory bandwidth bound, but mostly when fetching from textures, consider these optimizations:

- **Reduce the size of your textures.** Consider your target resolution and texture coordinates. Do your users ever get to see your highest miplevel? If not, consider scaling back the size of your textures. This can be especially helpful if overloaded frame buffer memory has forced texturing to occur from non-local memory (like system memory, over the AGP or PCI-Express bus). The NVPerfHUD memory graph can help diagnose this problem, as it shows the amount of memory allocated by the driver in various heaps.
- **Always use MIP mapping on any surface that may be minified.** MIP mapping delivers better image quality by reducing texture aliasing. A variety of filters can be used to create high-quality MIP maps that do not look blury. NVIDIA provides a suite of texture tools to aid you in optimal MIP map creation, including a Photoshop plug-in, a command line utility and a library – all available at http://developer.nvidia.com/object/nv_texture_tools.html. Without MIP mapping, you are limited to point sampling from a texture, which may cause an undesirable shimmering effect.

> **Note:** If you find that mipmapping on certain surfaces makes them look blurry, avoid the temptation to disable mipmapping or add a large negative LOD bias. Use anisotropic filtering instead.

- **Compress all color textures.** All textures that are used just as decals or detail textures should be compressed, using one of DXT1, DXT3, or DXT5, depending on the specific texture's alpha needs. This will reduce memory usage, reduce texture bandwidth requirements, and improve texture cache efficiency.
- **Avoid expensive texture formats if not necessary.** Large texture formats, like 64-bit or 128-bit floating point formats, obviously cost much more bandwidth to fetch from. Only use these as necessary.
- **Use appropriate anisotropic texture filtering levels**. When using low frequency textures and high aniso filtering levels, the GPU is doing extra work that does not improve the visual quality. If you are texture bandwidth limited, use the lowest aniso level that gives you good enough image quality. In an ideal application should have texture-specific aniso setting.
- **Disable trilinear filtering where unnecessary.** Trilinear filtering, even when not consuming extra texture bandwidth, costs extra cycles to compute in the pixel shader on most modern GPU architectures. On textures where miplevel transitions are not readily discernable, turn trilinear filtering off to save fillrate.

6.3.8. Optimize Frame buffer Bandwidth

The final stage in the pipeline, the ROP, interfaces directly with the frame buffer memory and is the single largest consumer of frame buffer bandwidth. For this

reason, if bandwidth is an issue in your application, it can often be traced to the ROP. Here's how to optimize for frame buffer bandwidth:

- **Render depth first.** Not only does this reduce pixel shading cost (see above), it also reduces frame buffer bandwidth cost.
- **Reduce alpha blending.** Note that alpha blending requires both a read and a write to the frame buffer, thus potentially consuming double the bandwidth. Reduce alpha blending to only those situations that require it, and be wary of high levels of alpha blended depth complexity
- **Turn off depth writes when possible.** Writing depth is an additional consumer of bandwidth, and should be disabled in multi-pass rendering (where the final depth is already in the depth buffer), when rendering alpha blended effects, such as particles, and when rendering objects into shadow maps (in fact, for rendering into color-based shadow maps, you can turn off depth reads as well).
- **Avoid extraneous color buffer clears. I**f every pixel is guaranteed to be overwritten in the frame buffer by your application, then clearing color should be avoided as it costs precious bandwidth. Note, however, that you should clear the depth and stencil buffers whenever you can, as many early-Z optimizations rely on the deterministic contents of a cleared depth buffer.
- **Render front-to-back.** In addition to the pixel shading advantages to rendering front-to-back mentioned above, there are also similar benefits in the area of frame buffer bandwidth, as early-Z hardware optimizations can discard extraneous frame buffer reads and writes. In fact, even older hardware without these optimizations will benefit from this, as more pixels will fail the depth-test, resulting in fewer color and depth writes to the frame buffer.
- **Optimize skybox rendering.** Skyboxes are often frame buffer bandwidth bound, but there is a decision to be made in how to optimize them. You can either render them last, reading (but not writing) depth, and allow the early-Z optimizations along with regular depth buffering to save bandwidth, or render the skybox first, and disable all depth reads and writes. Which of these techniques saves more bandwidth is a function of the target hardware and how much of the skybox is visible in the final frame. If a large portion of the skybox is obscured, the former technique will likely be better, otherwise the latter may save more bandwidth.
- **Only use floating point frame buffers when necessary.** These obviously consume much more bandwidth than smaller integer formats. The same applies for multiple render targets.
- **Use a 16-bit depth buffer when possible.** Depth transactions are a huge consumer of bandwidth, so using 16-bit instead of 32-bit can be a huge win and is often enough for small-scale indoor scenes that don't require stencil. It is also often enough for render-to-texture effects that require depth, such as dynamic cubemaps.
- **Use 16-bit color when possible.** This is especially applicable to render-to-texture effects, as many of these, such as dynamic cubemaps and projected color shadow maps, work just fine in 16-bit color.

Chapter 7. Troubleshooting

Your questions and comments are always welcome at on our developer forums and confidentially via email to NVPerfHUD@nvidia.com.

7.1. Known Issues

- NVPerfHUD does not handle multiple devices. It only supports the first device created with **`Direct3DCreate9()`**.
- NVPerfHUD may crash when using software vertex processing
- An application that uses **`rdtsc`** natively won't be able to function properly with the Frame Analysis Mode of NVPerfHUD – see Troubleshooting below for possible solutions.
- Frame Analysis Mode requires that your application use and rely on the **`QueryPerformanceCounter()`** or **`timeGetTime()`** win32 functions. Your application must be robust in handling elapsed time (dt) calculations, especially the case where **`dt`** is zero. In other words, you program should not divide by **`dt`**. If you suspect this is a problem, try setting the Delta Time option to "SlowMo" in the NVPerfHUD Configuration dialog. If this works, please consider fixing your application to handle the case where **`dt`** is zero.
- The State Inspectors do not display detailed information when your application is using fixed T&L.
- When you are in Frame Analysis Mode, if you deactivate NVPerfHUD (using your activation hotkey) and then exit your application you may see a warning dialog that says "Number of references when exiting was not 0."
- Applications running in windowed mode may not exit properly when NVPerfHUD is active and you click on the close button.

7.2. Frequently Asked Questions

NVPerfHUD says my application is not enabled for NVPerfHUD Analysis.
To ensure that unauthorized third parties do not analyze your application without your permission, you must make a minor modification to enable NVPerfHUD analysis. Refer to the Getting Started section of this User Guide for instruction
My application does not respond while NVPerfHUD is active.
When NVPerfHUD is enabled using the hotkey feature, it consumes all keyboard input and does not pass any key stroke events to the application. You can toggle this mode on/off using the activation hotkey you selected.
I can see the NVPerfHUD header across the top of my screen, but it doesn't respond to my activation hotkey.
NVPerfHUD uses several methods of intercepting key stroke events. If you are using a method that is not yet supported, please let us know so we can update NVPerfHUD. Note that in Win2K NVPerfHUD uses DirectInput to listen for your activation hotkey and to intercept keyboard commands while activated. DirectInput supplies two types of data: buffered and immediate. Buffered data is a record of events that are stored until an application retrieves them. Immediate data is a snapshot of the current state of a device. What this means is that your application needs to use the IDirectInputDevice8::GetDeviceData interface instead of the IDirectInputDevice8::GetDeviceState interface if you want to access advanced features of NVPerfHUD such as bottleneck identification experiments, shader visualization, etc.
NVPerfHUD messes up alpha and some rendering states.
NVPerfHUD renders the HUD at the end of the frame. It also changes the rendering states to draw itself, but does not restore them to their original state. In other words, it does not push and pop rendering states for performance reasons—therefore, it is assumed that your application resets the rendering states at the beginning of each frame.
The GPU_IDLE (GREEN) line is not reporting any data.
This information is not available for GeForce4 (NV25) and older GPUs. You may also see this on newer GPUs if NVPerfHUD is unable to communicate properly with the display driver. Verify that you are using the latest version of NVPerfHUD and running the latest NVIDIA display drivers.

The colored lines in the performance graphs don't show up or are all zero, but the NVPerfHUD header and the batching histogram are working.

These features are not enabled when using the DirectX 9 DEBUG runtime. Use the DirectX control panel to switch from using the DEBUG runtime back to the RETAIL runtime for accurate performance analysis. It is OK to use the DEBUG runtime if you are only using the NVPerfHUD Debug Console Mode to look for functional problems in your application.

I see some extra lines in the middle of my NVPerfHUD graphs.

If you are running older drivers (especially on Win2K) you may see portions of the old NVPerfHUD 1.0 graphs super-imposed on top of your NVPerfHUD graphs. Upgrading your drivers to 71.8x or later should fix the problem.

Some objects in my scene continue to animate in Frame Analysis Mode

When you switch to Frame Analysis Mode, NVPerfHUD stops the clock for your application and you can perform in-depth analysis of the current frame while it is frozen. Your application must use and rely on the QueryPerformanceCounter() or timeGetTime() win32 functions. If your application uses the **rdtsc** instruction it will not function properly with Frame Analysis Mode.

If your application uses frame-based animation, freezing time will have no effect on animated objects.

What else do I need to know about NVPerfHUD?

- Multi sampled render targets at not displayed in Frame Analysis mode.
- NVPerfHUD may crash if you turn on "Break on D3D errors" in the DirectX Control Panel.
- Pixel shader visualization does not work for primitives that use VS 3.0.

I have discovered a problem that is not listed above

We want to make sure NVPerfHUD continues to be 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 NVPerfHUD.

NVPerfHUD@nvidia.com

Appendix A.
Why the Driver Waits for the GPU

The GPU fully utilized scenario is the typical situation that happens when you have two processors connected by a FIFO, and one chip is feeding the other with more data than it can process.

In this case shown below, the CPU is feeding the GPU with more commands than it can process. When this happens all the commands start to build up in the FIFO queue, also called the "push buffer". To prevent this FIFO from overflowing the driver is forced to wait until there is some room in the FIFO to place new commands.

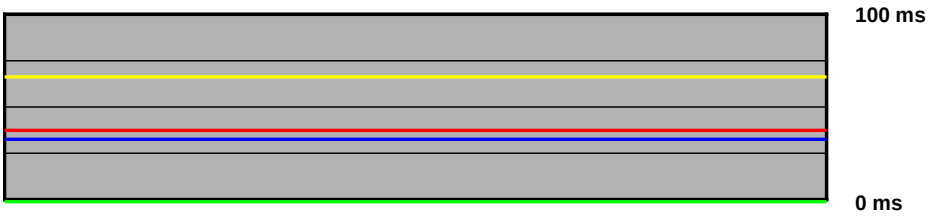


Figure 20. Driver Waiting for the GPU

If you find that the frame rate is:

- **High**, then you can do more work on the CPU and this should not affect the frame rate (object culling, physics, game logic, AI, etc...).
- **Not adequate**, you should reduce the scene complexity to lighten the GPU load.

Notice

Trademarks

Copyright

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