NVIDIA PerfHUD 5.1 User Guide

DU-01231-001_v09
Nov 2007

DEVELOPMENT

Table of Contents

List of Figures

Chapter 1. Quick Tutorial

Overview

This chapter presents a short tutorial to quickly introduce you to several convenient and powerful new features. Even if you've used previous versions of PerfHUD, we highly recommend that you read through this tutorial because so much is new in PerfHUD 5.

Launching PerfHUD

By default, the PerfHUD installer will place a shortcut to the PerfHUD Launcher on your desktop. To analyze an application, simply drag its icon onto the PerfHUD launcher. Keep in mind that the application needs to opt-in for PerfHUD analysis, to prevent unauthorized parties from analyzing your application.

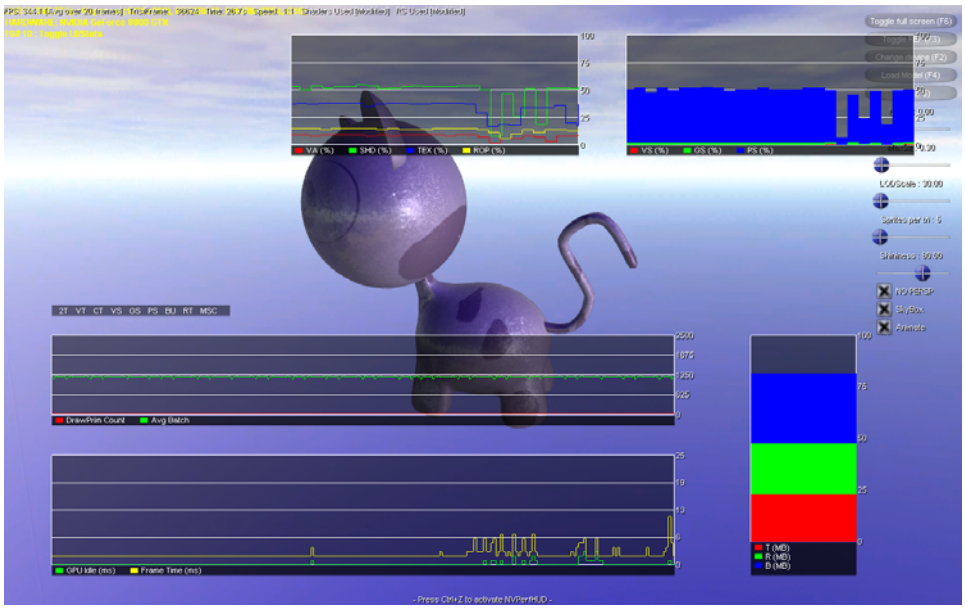
Let's analyze the sample DirectX 10 application that ships with PerfHUD, Sparkles. (This sample is taken from the NVIDIA Direct3D 10 SDK, and includes the opt-in modification.) For this particular application, you can use the "Sparkles Sample" shortcut in the PerfHUD group in the Start menu.

If this is the first time you're running PerfHUD, you'll see a configuration dialog box. The main thing you have to do here is to choose a shortcut key. Pick **Ctrl+Z**.

Once you click **OK**, Sparkles will start, and PerfHUD will be running on it, as shown below.

Note that any keyboard or mouse input will still go the Sparkles application, and not to PerfHUD, until you activate PerfHUD using your hotkey (Ctrl+Z). PerfHUD reminds you of your hotkey with a message at the bottom of the screen: "Press Ctrl+Z to activate PerfHUD".

Before activating PerfHUD, press **F9** and **F10** to hide the user interface of Sparkles, reducing clutter. (Remember, these are hotkeys of Sparkles – once PerfHUD is active, F9 and F10 will perform different functions.)



Activating PerfHUD

Activate PerfHUD by pressing **Ctrl+Z**. You'll see the status line at the bottom of the screen change to four buttons, one for each mode of PerfHUD:

Now, any keyboard or mouse input you make will affect PerfHUD. You can toggle between PerfHUD and your application at any time. For example, you may want to navigate to a different part of the scene to analyze it, and then re-enable PerfHUD when you're done.

Help Screen

At any time while you're running PerfHUD, you can press **F1** to view the Help window. This window also has options for getting System Information as well as setting various PerfHUD options.

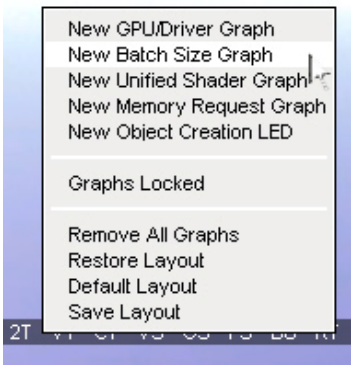
Performance Dashboard

PerfHUD starts in the Performance Dashboard. This mode displays many useful data values, such as per-unit GPU utilization, driver time, memory usage, and more. New in PerfHUD 5 is the ability to completely customize the Performance Dashboard's layout.

Creating a New Batch Size Graph

Let's start by creating a new Batch Size graph. This graph displays batches and sizes, allowing you to easily understand the batching characteristics of your application.

To add a new graph, **right-click on the background** and choose **New Batch Size Graph**.



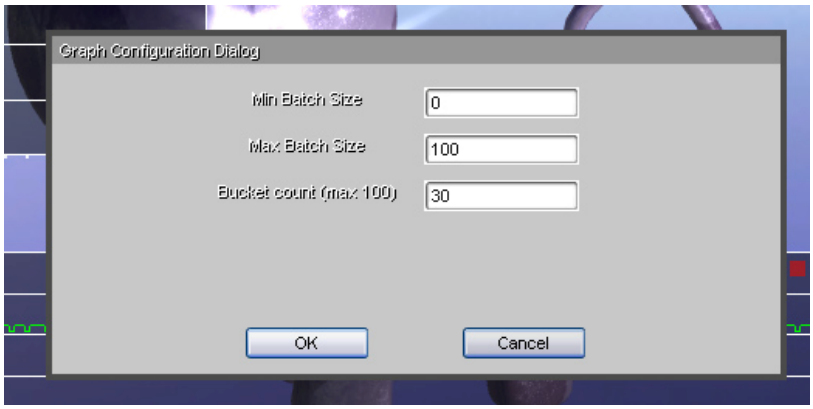
A new Batch Size graph will now appear with its default settings:

Every graph in the Performance Dashboard is customizable. To do this, simply hover your mouse anywhere on the graph. You'll see three boxes appear: blue and red boxes at the upper right of the graph, and a green box on the lower right:

- Clicking on the **blue box** brings up a configuration dialog.
- Clicking on the **red box** closes the current graph.
- Clicking and dragging on the **green box** resizes the current graph.

Let's customize the Batch Size Graph. First, resize it using the **green box**. Then click on the **blue box** and you'll see the Graph Configuration Dialog.

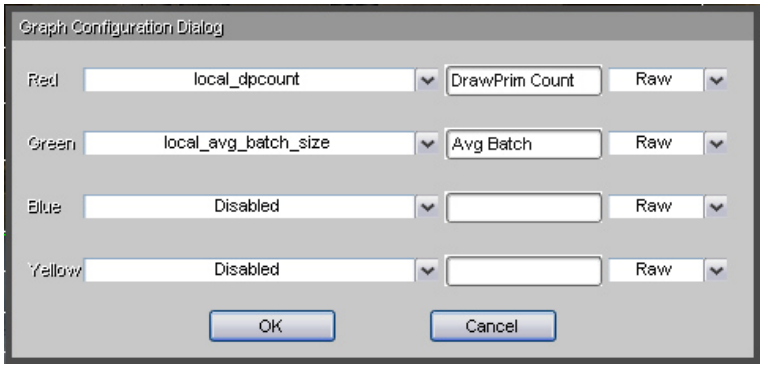
Set the **Maximum Batch Size** to **100**. Then click **OK**. The graph will now show more bars.



Adding Signals

The most common type of graph in PerfHUD is the GPU/Driver Graph. Each GPU/Driver graph can display up to 4 signals simultaneously. PerfHUD 5 allows you to choose from a huge list of both GPU and driver signals, allowing you to monitor virtually any aspect of your application's graphics performance.

Let's add some signals to the GPU/Driver graph that displays the DrawPrimitive Count and Average Batch by default. To do this, hover over the graph and **click on the blue square** at the upper-right of the graph. A Graph Configuration Dialog will pop up:



Here, you can choose any signal you want for each line color, as well as descriptions for each. You can also decide whether you want to graph the raw signal or a percentage.

Choose **D3D FPS** for the blue line, and name it "FPS".

Choose **D3D vidmem MB** for the yellow line, and name it "D3D Vid Mem (MB)"

Speeding Up and Slowing Down Time

By pressing the **+** and **–** keys, you can scale the passing of time from 6x faster than normal down to 1/8 speed. Pressing the **–** key again when at 1/8 speed will freeze

time completely. Controlling time is helpful when you want to find a particularly troublesome set of frames.

Running Experiments

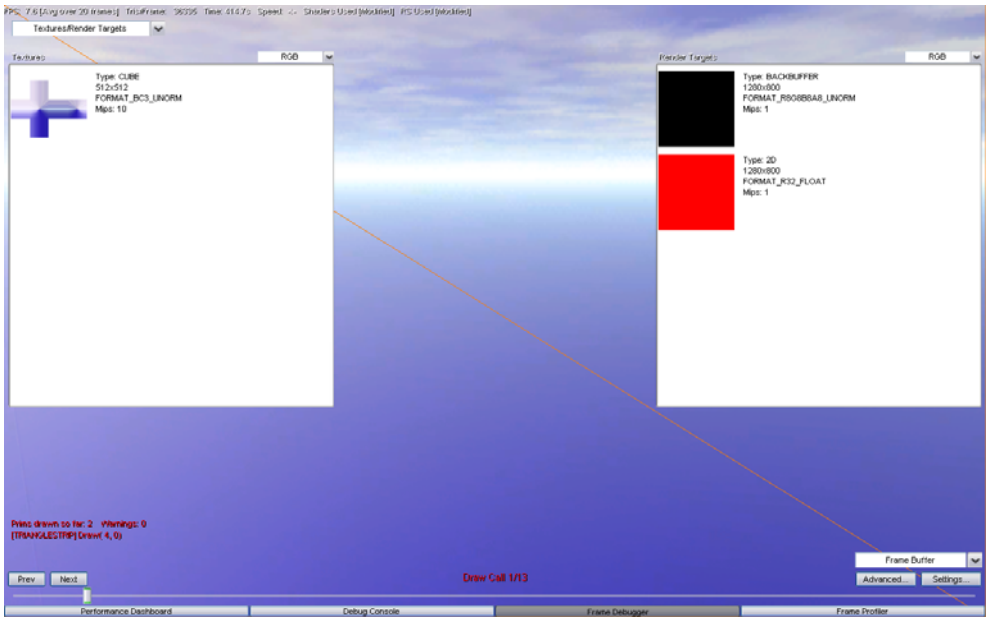
You can also perform various useful experiments from the Performance Dashboard. These are listed below along with their respective keyboard shortcuts.

Use 2x2 Textures	Ctrl+T
Set NULL Viewport	Ctrl+V
Wireframe	Ctrl+W
Ignore Draw Calls	Ctrl+N
Color Fixed Function Shaders Red	Ctrl+1
Color ps_1_1 Shaders Light Green	Ctrl+2
Color ps_1_3 Shaders Green	Ctrl+3
Color ps_1_4 Shaders Yellow	Ctrl+4
Color ps_2_0 Shaders Light blue	Ctrl+5
Color ps_2_a Shaders Blue	Ctrl+6
Color ps_3_0 Shaders Orange	Ctrl+7
Color ps_4_0 Shaders Red	Ctrl+8

Using the Frame Debugger

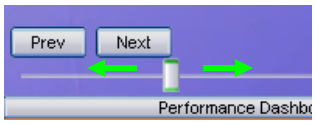
The Performance Dashboard is most useful for finding a troublesome spot in your scene. Once you've found that spot, you will often want to freeze the frame, debug its draw calls, and analyze its performance in detail.

Press F7 to switch to the Frame Debugger. The Frame Debugger will show you just the first draw call in the scene, which in this case is the skybox:

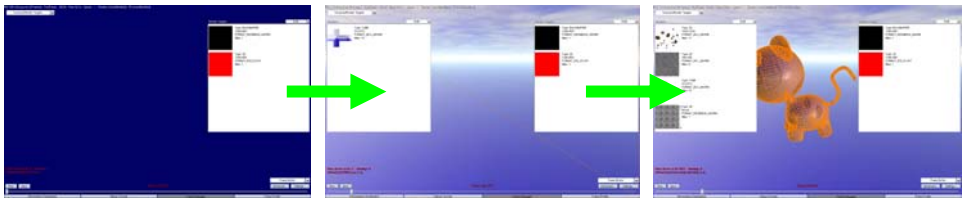


Scrubbing Through the Frame

Click and drag the slider at the bottom of the screen from side to side.



You'll see how the frame builds up with various draw calls.



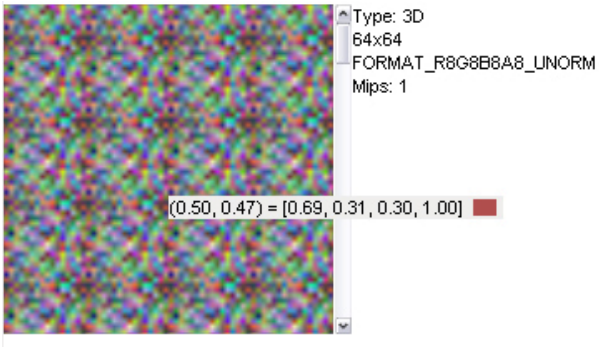
The current draw call is highlighted with an orange wireframe.

You can use the **up** and **down** arrow keys to decrement or increment the current draw call. **Home** jumps to the first draw call, and **End** jumps to the last draw call. **Page Up** and **Page Down** decrement or increment the current draw call by larger amounts.

Drag the slider to draw call 2. You should see the cat highlighted in orange wireframe.

Viewing Textures and Render Targets

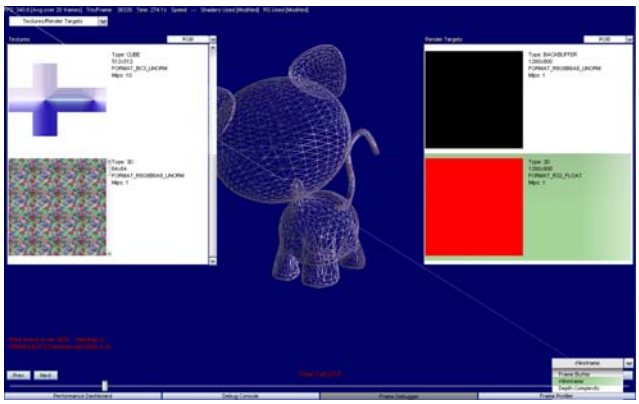
All the textures used by the current draw call are shown in the Textures panel on the left of the screen. **Click on the Textures panel** (to get focus) and **press + twice** to enlarge the textures. (Pressing - will reduce the textures.) Note that if you hover over a texture, a tooltip will appear showing u-v coordinates and RGBA color information.



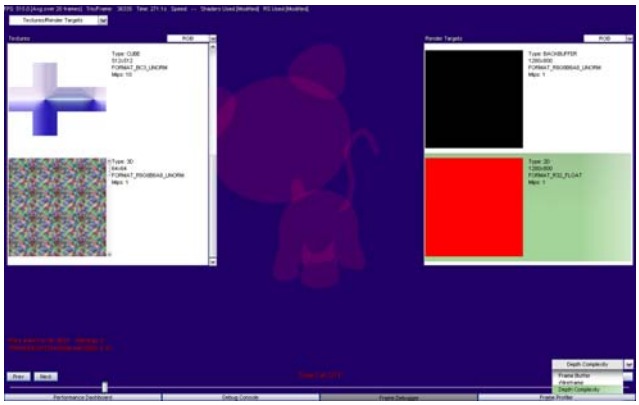
On the right is the list of Render Targets. You can perform the same operations in that panel as in the Textures panel.

Changing the Viewing Mode

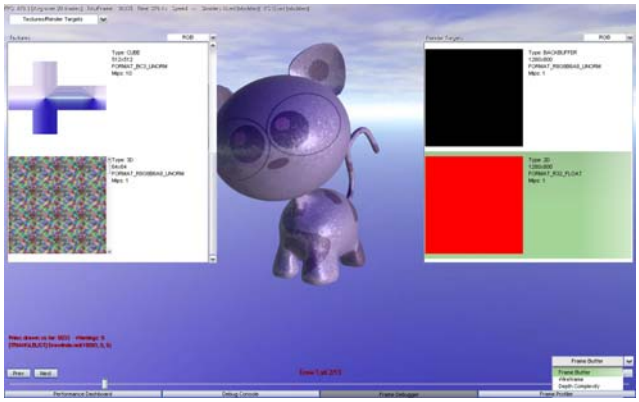
In addition to viewing the **Frame Buffer** as usual, you can also view **Wireframe**, **Depth Complexity**, and **Depth Buffer** renderings for the current frame by choosing options from the drop-down. These views are shown below.



Wireframe



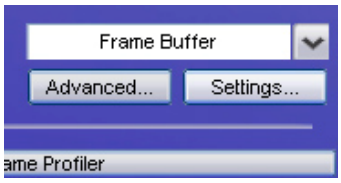
Depth Complexity



Frame Buffer

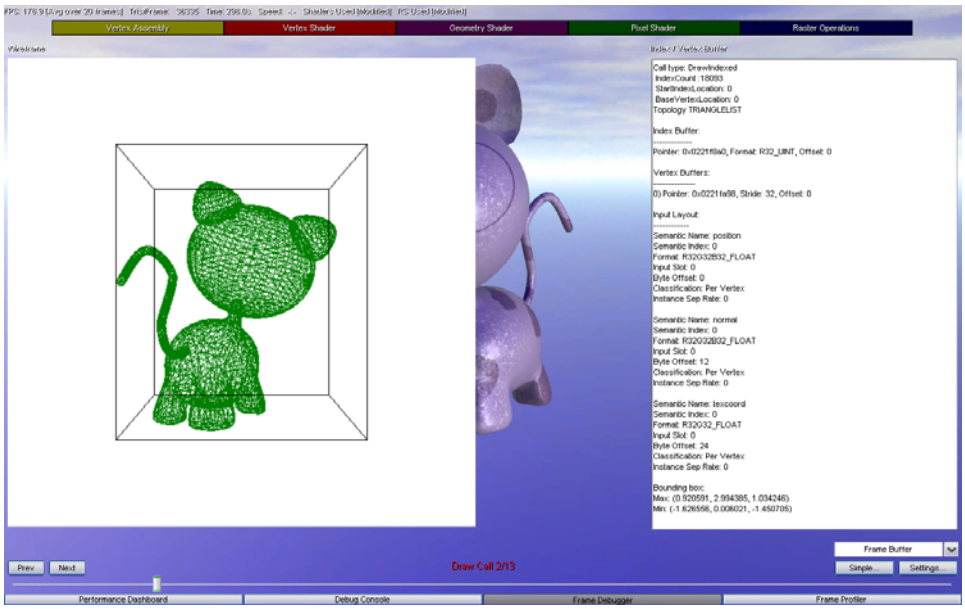
Using the Advanced State Inspectors

To analyze a particular draw call in depth, you can use PerfHUD's Advanced State Inspectors. Access these by clicking on the **Advanced…** button at the lower-right of the screen.



The Vertex Assembly State Inspector

You'll first see the Vertex Assembly State Inspector. Here you can see the geometry used in the current draw call. You can click and drag the mouse on the geometry to rotate it. You can also view details about the geometry in the panel on the right.

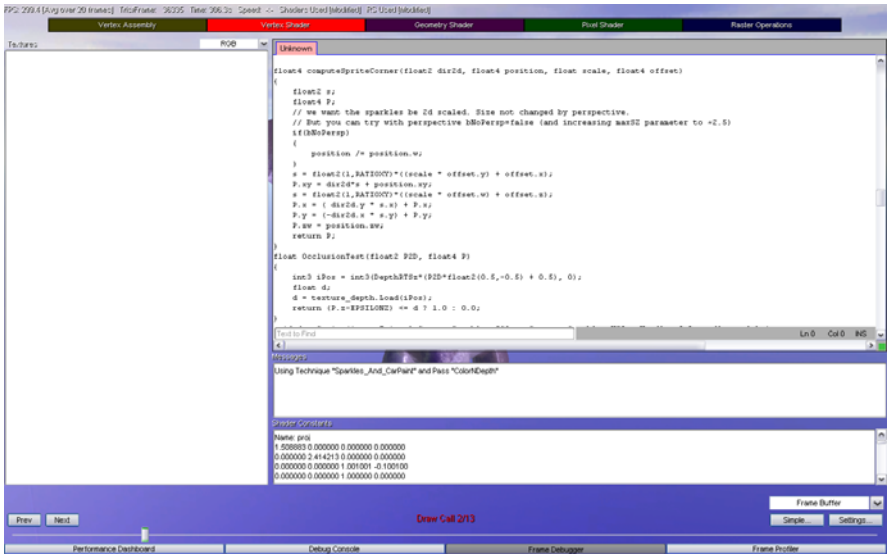


Next, switch to the Vertex Shader state inspector by clicking on the red **Vertex Shader** block at the top of the screen.

Vertex Shader State Inspector

The Vertex Shader State Inspector shows you any vertex shader code from the current draw call, as well as any textures and shader constants that are used. In this case, there are no textures, so the panel at the left of the screen is blank. You can also edit the shader in real-time (we'll cover that when we look at the pixel shader).

Click on the purple **Geometry Shader** block at the top of the screen.



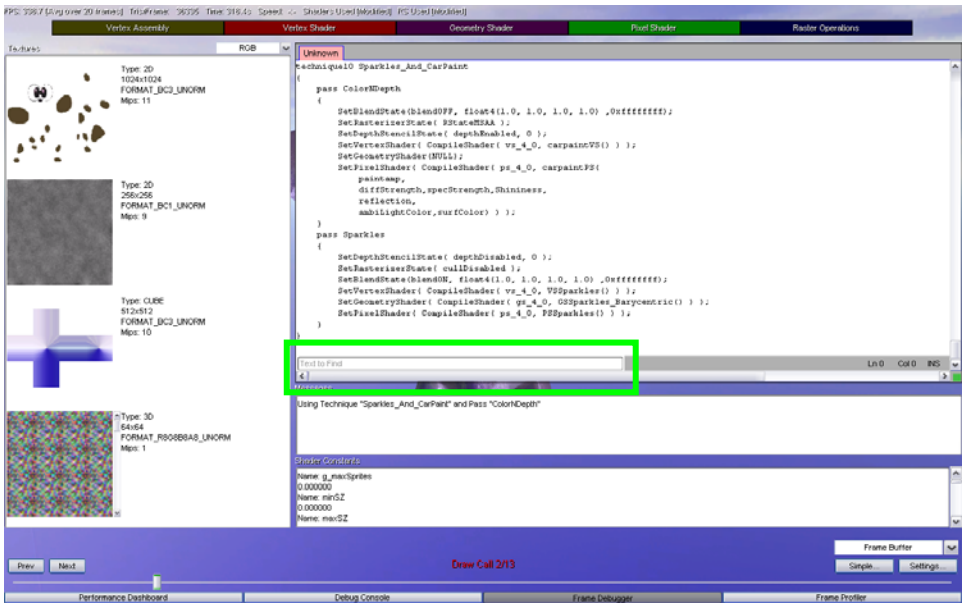
Geometry Shader State Inspector

This state inspector is similar to the Vertex Shader state inspector, showing any geometry shader code, textures, and constants.

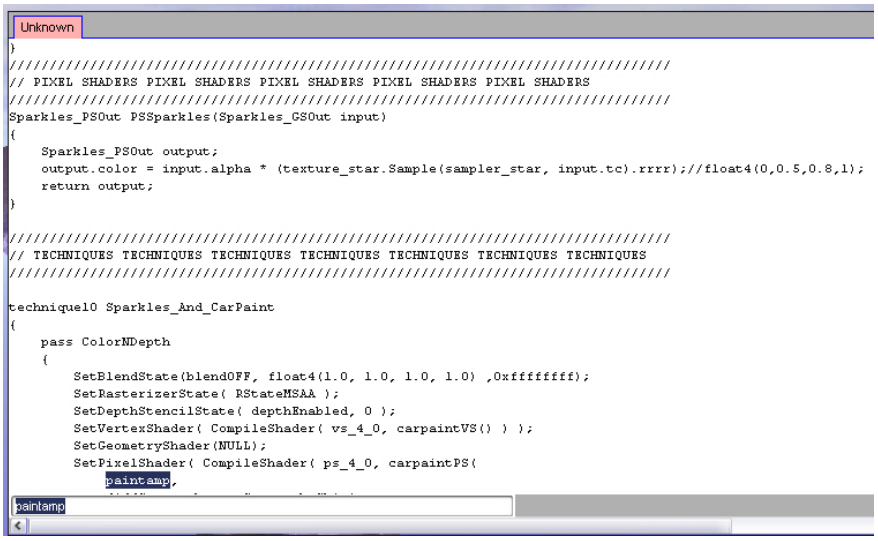
Click on the green **Pixel Shader** block at the top of the screen.

Pixel Shader State Inspector

The Pixel Shader state inspector is similar to the Vertex Shader and Geometry Shader state inspectors, showing any geometry shader code, textures, and constants.



The search field (shown in green above) allows you to quickly find a particular text string. Type **paintamp** into the search field and press **Enter**. The shader editor will jump to the first occurrence of paintamp.



Now, replace **paintamp** with **0.5**. Then **right-click** in the editing area and choose **Compile** from the context menu. (You can also save and load your shaders using the context menu.)

Your modified shader is now used by the application. Press **F2** to hide PerfHUD's user interface, so you can see the modified rendering.

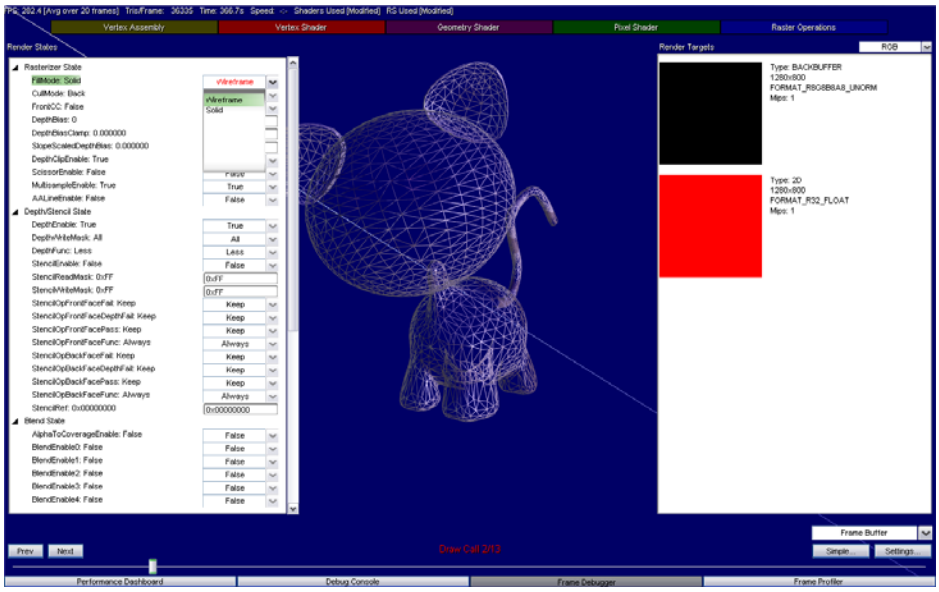
Revert the shader to its original form by **right-clicking** in the editing area and choosing **Revert to Original Shader**.

Next, click on the blue **Raster Operations** block at the top of the screen.

Raster Operations State Inspector

The Raster Operations state inspector allows you to view and manipulate numerous useful render states. Any changes you make here affect all draw calls in the scene. (Future versions of PerfHUD will allow you to affect draw calls grouped by state buckets.)

Select the first dropdown (for the Fillmode) and change it to **Wireframe**. Your screen should now look like this:

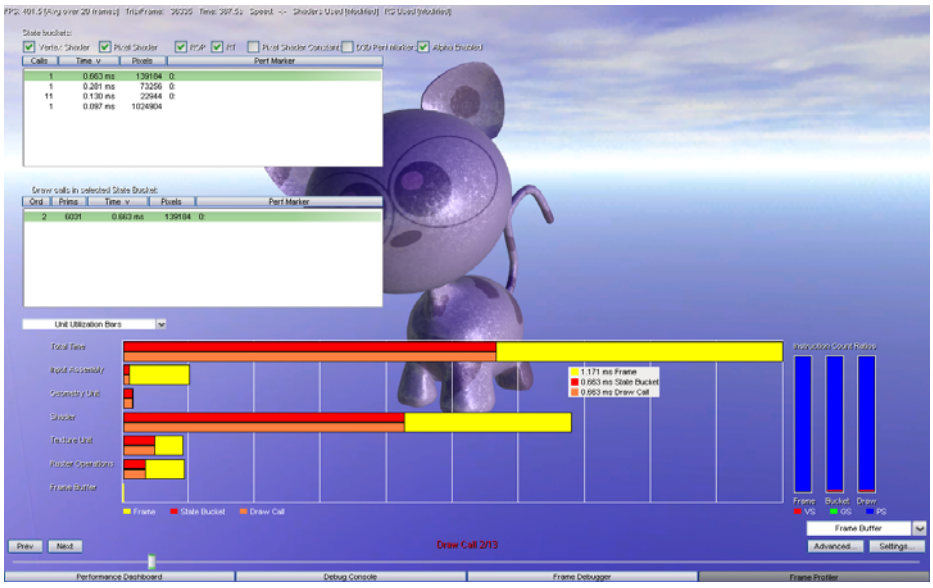


Now right-click on that same drop-down and select **Restore All States** from the resulting context menu. Note that you can restore states by category if you want to.

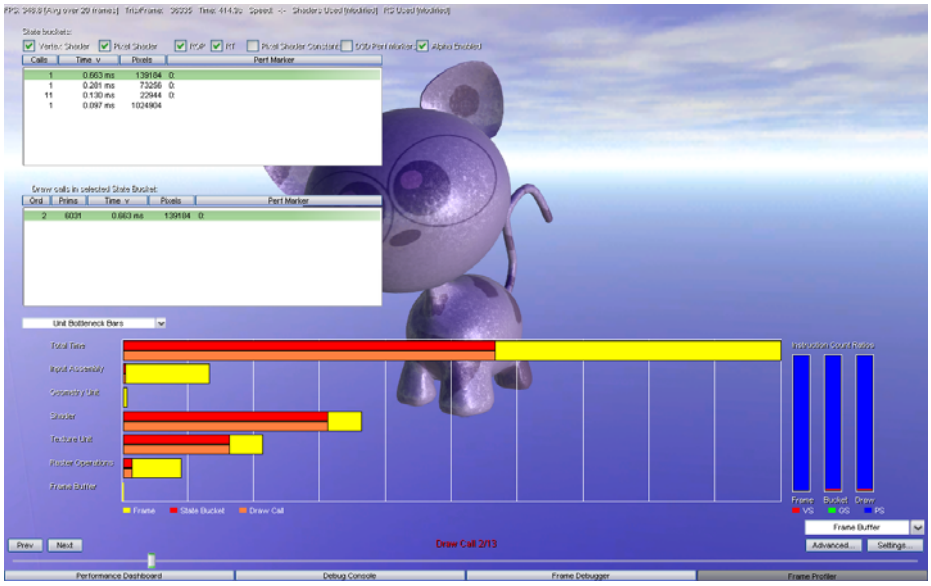
Frame Profiler

Press **F8** to enter the Frame Profiler. You'll see PerfHUD quickly run a series of tests on the current frame, giving you detailed statistics about draw call performance and GPU usage. This is one of the uniquely powerful features of PerfHUD – complete bottleneck analysis with just one key press.

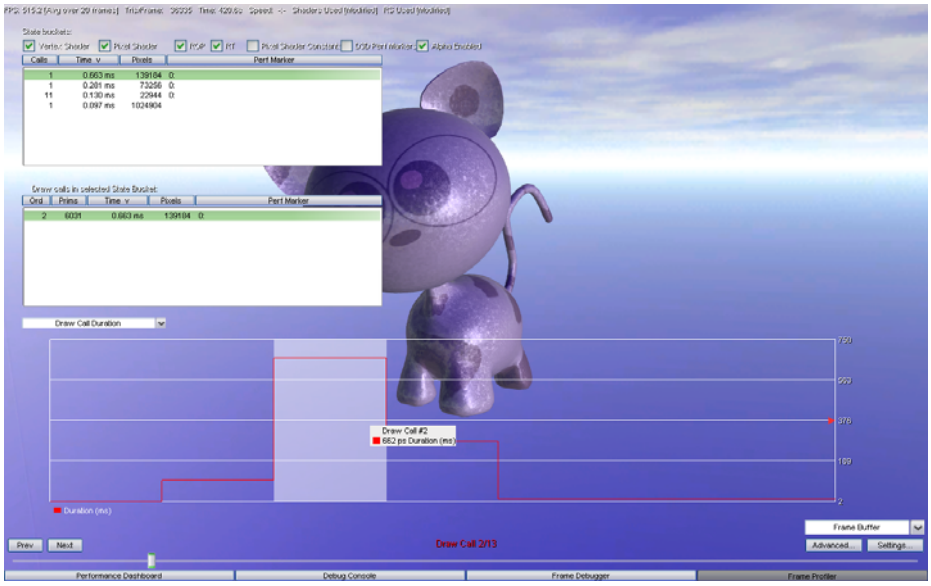
The Frame Profiler offers several different visualizations, which are listed and explained briefly below.



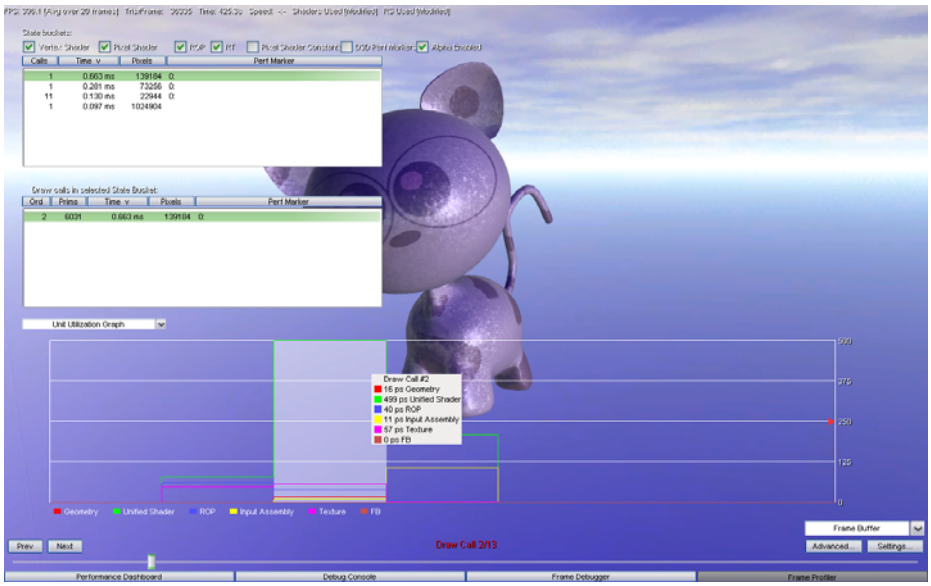
Unit Utilization Bars. Shows how long each GPU unit was used for the selected draw call, state bucket, and frame. You can define state bucket groupings using the checkboxes at the top of the screen.



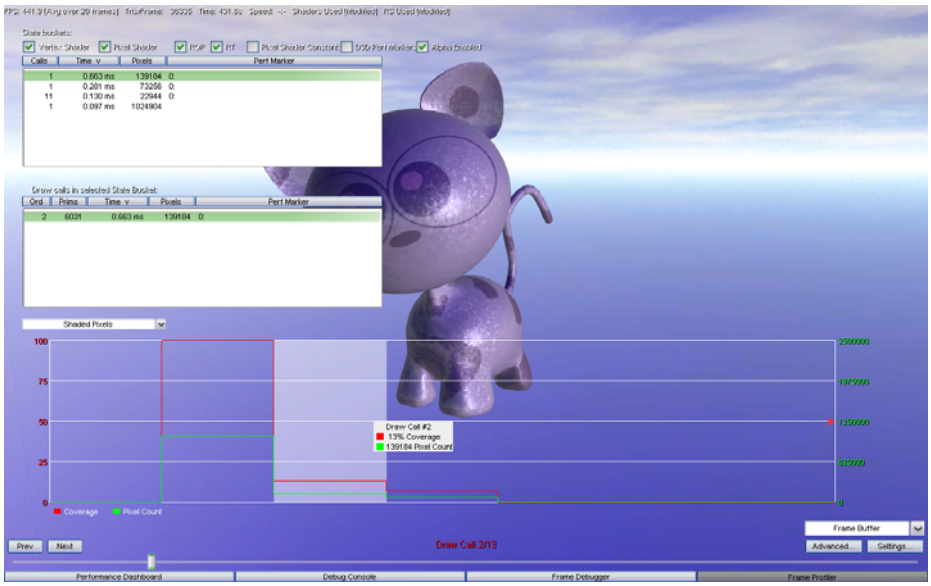
Unit Bottleneck Bars. Shows how long each GPU unit was the bottleneck for the selected draw call, state bucket, and frame. You can define state bucket groupings using the checkboxes at the top of the screen.



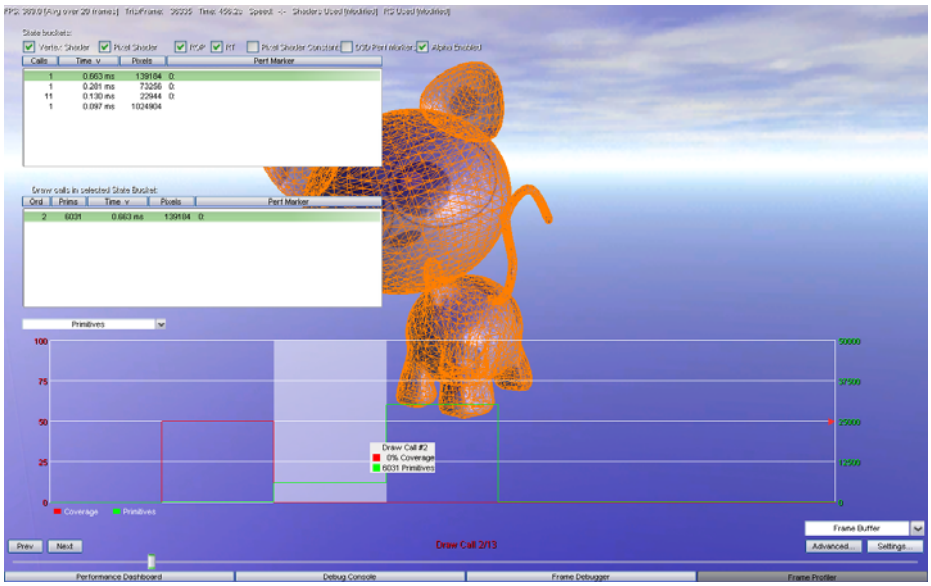
Draw Call Duration. Shows how long each draw call in the frame took. (The horizontal axis is draw call number.) You can click to jump to a draw call, and see tooltips to get exact values.



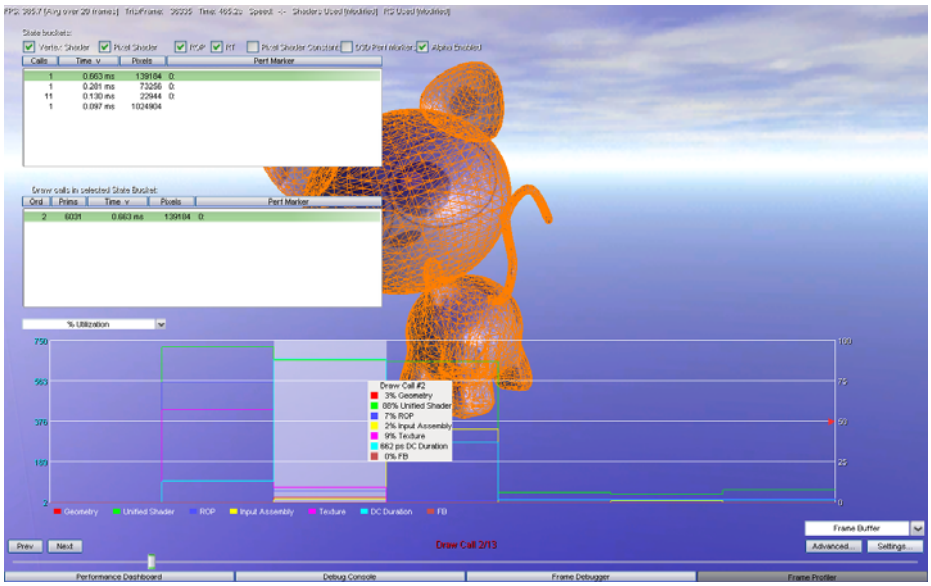
Unit Utilization Graph. Shows how much each GPU unit was utilized for each draw call in the frame. You can click to jump to a draw call, and see tooltips to get exact values.



Shaded Pixels. Shows how many pixels were drawn by each draw call, as well as what percentage of the screen was covered by the draw call. You can click to jump to a draw call, and see tooltips to get exact values.



Primitives. Shows the number of primitives drawn by each draw call, along with the percentage of the screen covered. You can click to jump to a draw call, and see tooltips to get exact values.



% Utilization. Shows how utilized each GPU unit was for each draw call. You can click to jump to a draw call, and see tooltips to get exact values.

Chapter 2. What's New in PerfHUD 5?

New Features

This chapter lists the key NEW features of PerfHUD 5. This is in addition to all the features that were carried over from previous PerfHUD versions.

- DirectX 10 support on Windows Vista
- DirectX 9 support on Windows Vista
- Edit & Continue for HLSL and .fx vertex, geometry, and pixel shaders
- Edit & Continue for Raster Operations state
- Customizable Performance Dashboard
  - User chooses up to 4 counters per graph
  - Full set of 40+ PerfSDK Direct3D and GPU counters available
  - Arrange graphs as you choose
  - Create and delete graphs
  - Save/load custom layouts
  - Layout stored automatically when exiting PerfHUD
  - Double-clicking color swatch in graph legend toggles display of that channel
- Improvements to Frame Debugger
  - Visualization of 2D textures, 3D textures, shadow maps, and cube maps
  - User can arbitrarily rotate wireframe visualization
  - Show selected draw call only (versus selected and all previous calls)
  - Mouseover on textures and render targets shows texture coordinates and texel color
- Improvements to Frame Profiler
  - Instruction Count Ratio graphs
  - Tooltips for graphs with graph values
  - "Alpha Enabled" state bucket criteria
  - Support for Hierarchical Direct3D Performance Markers
- Improved user interface
  - Polished look-and-feel with new fonts and widgets
  - Hardware mouse cursor improves responsiveness when frame rate is low
  - Clicking on graphs in Frame Profiler jumps to corresponding draw call
  - Clear legends for all graphs

- New Help screen with software version, GPU, driver information, and keyboard shortcuts
- Options screen with numerous configuration options:
  - Clear color buffer when viewing Z-only passes
  - Preserve backbuffer
  - Draw call visualization mode
- F2 hides/shows PerfHUD UI

- Compatibility, stability, and reliability improvements
  - Extensive testing on a wide range of applications
  - Minor bug fixes

Chapter 3. What is PerfHUD?

As graphics processing units (GPUs) grow ever more complex, getting optimal performance out of them can be a daunting task. Because GPUs are pipelined processors, it's particularly important that you identify and address the slowest stages of the pipeline. Otherwise, you can spend a great deal of effort without getting any improvement in frame rate.

PerfHUD is a performance profiling and visual debugging tool that helps you to solve this complex problem. It does this by presenting a variety of graphs, experiments and state inspectors about the graphics pipeline superimposed on your Direct3D application as a heads up display (HUD). (An activation hotkey allows you to switch between interacting with your application and interacting with PerfHUD.) In addition, PerfHUD offers automated performance analysis to quickly identify the most expensive draw calls.

A short overview video of PerfHUD is on the PerfHUD home page: http://developer.nvidia.com/PerfHUD.

PerfHUD is used by leading game developers around the world on top titles. Some examples are listed below:

Unreal Tournament 3 (Epic Games)	*Company of Heroes* (Relic Entertainment)	*Crysis* (Crytek)
EVE Online (CCP Games)	*World of Warcraft* (Blizzard Entertainment)	*Battlefield 2142* (DICE)
Hellgate: London (Flagship Studios)	*Gamebryo* (Emergent Technologies)	*Guild Wars* (ArenaNet)

For screenshots, testimonials, and reviews, please see the PerfHUD page on developer.nvidia.com.

How PerfHUD Works

When PerfHUD is enabled, it effectively wraps itself around your application, using a variety of sophisticated techniques to gather the information it needs to generate the HUD. PerfHUD uses special performance monitoring routines in the display driver that collect metrics directly from the GPU and within the driver itself. PerfHUD also uses API interception to collect various metrics and interact with your application. These instrumentation techniques are required for PerfHUD to function properly, and introduce some small additional overhead.

Figure 1 illustrates how PerfHUD interacts with various system components to gather data. The green boxes and arrows in the diagram represent components and interactions related to PerfHUD.

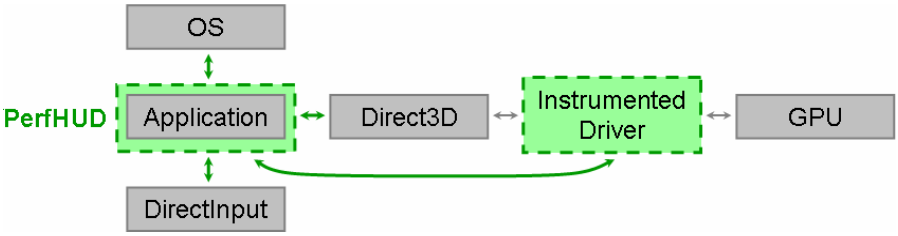


Figure 1. How PerfHUD Interacts with Various System Components

PerfHUD is a component of a larger performance toolkit called NVIDIA PerfKit. The diagram below shows the various components of PerfKit.

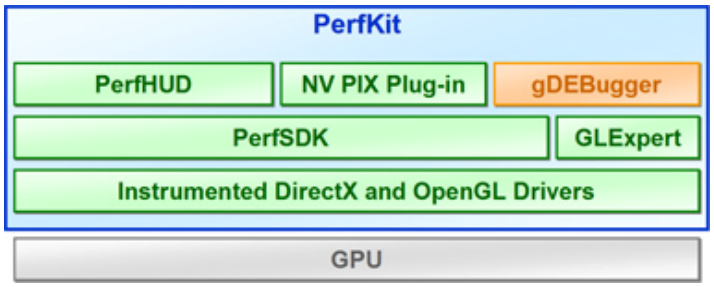


Figure 2. The Components of PerfKit

For more information about PerfKit, please see the PerfKit page on developer.nvidia.com.

PerfHUD Modes

PerfHUD provides four different ways to look at your application's performance. By switching between them, you can identify large-scale performance problems, per-frame issues, and drill down all the way to a detailed analysis of the draw calls in a particular frame. The four modes are:

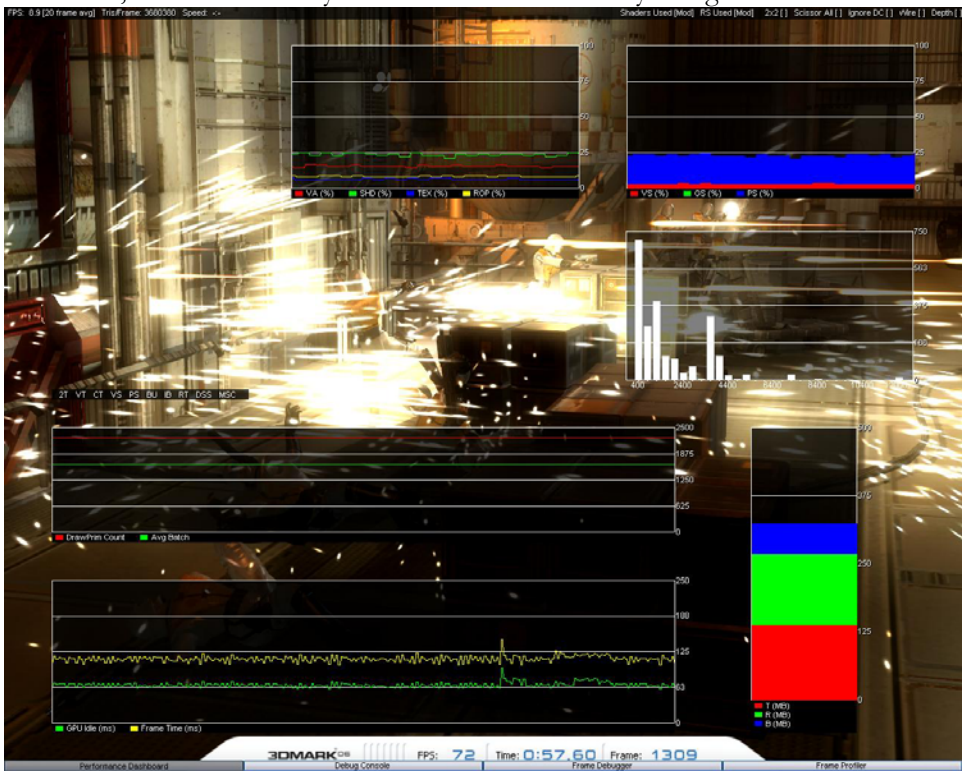
F5 **Performance Dashboard**
Watch the timing graphs and utilization graph detect problem areas in your application. Control how fast your application is running to zero in on specific problem frames and then switch to Frame Debugger Mode or Frame Profiler Mode for more details. On older GPUs, use directed experiments to identify bottlenecks.

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

F7 **Frame Debugger**
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 advanced State Inspectors that show details for each stage in the graphics pipeline.

F8 **Frame Profiler**
Freeze the current frame and profile how your application is using the GPU. This is the most powerful mode that PerfHUD offers, allowing you to sort all draw calls in the current frame by cost. In addition, several performance graphs and analysis tools are available.

When you run your application with PerfHUD, it starts in the Performance Dashboard mode, a graphical overlay is displayed on top of your Direct3D application, as shown in Figure 3 below. The four modes are listed at the bottom of the screen, with the currently selected mode indicated by two green boxes.



3DMark06 used with permission from Futuremark Corporation.

Figure 3. PerfHUD Running on a Direct3D Application

Read the chapters dedicated to each mode to ensure you get the most out of PerfHUD.

System Requirements

- Any recent NVIDIA GPU (GeForce 8 Series, GeForce 7 Series, GeForce 6 Series, G80-based, G70-based, or NV4X-based Quadro FX, or better

recommended.)
Older GPUs are supported with reduced functionality.
NVIDIA display driver with instrumentation enabled (PerfKit installs an instrumented driver and automatically enables instrumentation. You can disable instrumentation through the NVIDIA Control Panel.)

- Microsoft DirectX 9.0c
- Windows XP

Recommended Links

- [Link] PerfHUD Introductory Video
- NVIDIA Developer Web Site
  http://developer.nvidia.com
  - [Link] Optimize your GPU with the Latest NVIDIA Performance Tools
  - [Link] *NVIDIA GPU Programming Guide* – all the latest tips and tricks
  - [Link] *Balancing the Graphics Pipeline for Optimal Performance*
  - [Link] NVShaderPerf – shader performance analysis utility
  - [Link] NVIDIA SDK – hundreds of code samples & effects
- [Link] PerfKit User Guide
- [Link] *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*
  Several of the performance-related chapters are particularly helpful.
- [Link] *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*
- [Link] *GPU Gems 3*
- [Link] Microsoft DirectX web site
- [Link] Microsoft Developer Network (MSDN) web site
  Search for "performance" and "optimization"
- Microsoft DirectX SDK documentation [in the Start menu after installation]

Chapter 4. Getting Started

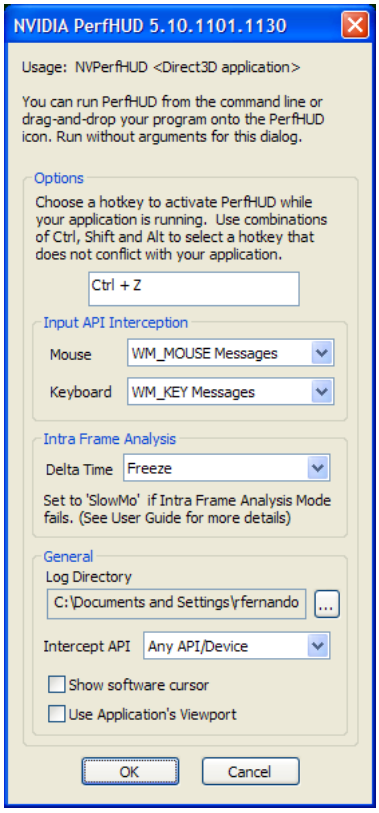
This chapter explains the basics of starting and using PerfHUD, as well as the simple steps needed to enable PerfHUD to work with your application.

If this is your first experience with PerfHUD, we highly recommend that you check out the introductory video that is available online.

The subsequent chapters explain how to use PerfHUD's various capabilities to identify and address rendering and performance issues in your application.

Quick Start

When you run your application with PerfHUD, the Performance Dashboard Mode graphs are displayed on top of your application. The advanced features of PerfHUD are available via the activation hotkey you provide during the setup process described below.



1. **Install PerfKit**
   The installer will update your driver, install PerfKit, and installer PerfHUD on your system. A new icon will be placed on your desktop for PerfHUD.
2. **Run PerfHUD**
   The first time you run the PerfHUD 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. For example, you might choose F12, Ctrl+Z, or Shift+Z.

Note: Once PerfHUD is activated using the activation hotkey, all subsequent keyboard events are intercepted by PerfHUD. When you are done analyzing your application and want to close

it, you'll have to press the activation hotkey again to disable PerfHUD so that you can use the keyboard and mouse to quit your application.

4. **Configure API Interception**
   Tell PerfHUD 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: By default, PerfHUD forces NON-PURE device creation because this is required for several features.

5. **Drag-and-drop your application onto the PerfHUD desktop icon.**
   Click **OK** to confirm your configuration options and then drag-and-drop your `.EXE`, `.BAT` or `.LNK` (shortcut) file onto the PerfHUD 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.
6. **Optional: Change the "Delta Time" setting to SlowMo if your application crashes or has problems in Frame Debugger Mode or Frame Profiler Mode.** This will help determine whether the problem is in PerfHUD or your application. See the Troubleshooting section for more details.
7. **Optional: Change the "Log Directory".** This specifies where screenshots and exported files will appear. You can take a screenshot at any time from within PerfHUD by pressing **F11**. The screenshot will be named according to the current date and time.

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

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

Note: PerfHUD 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.

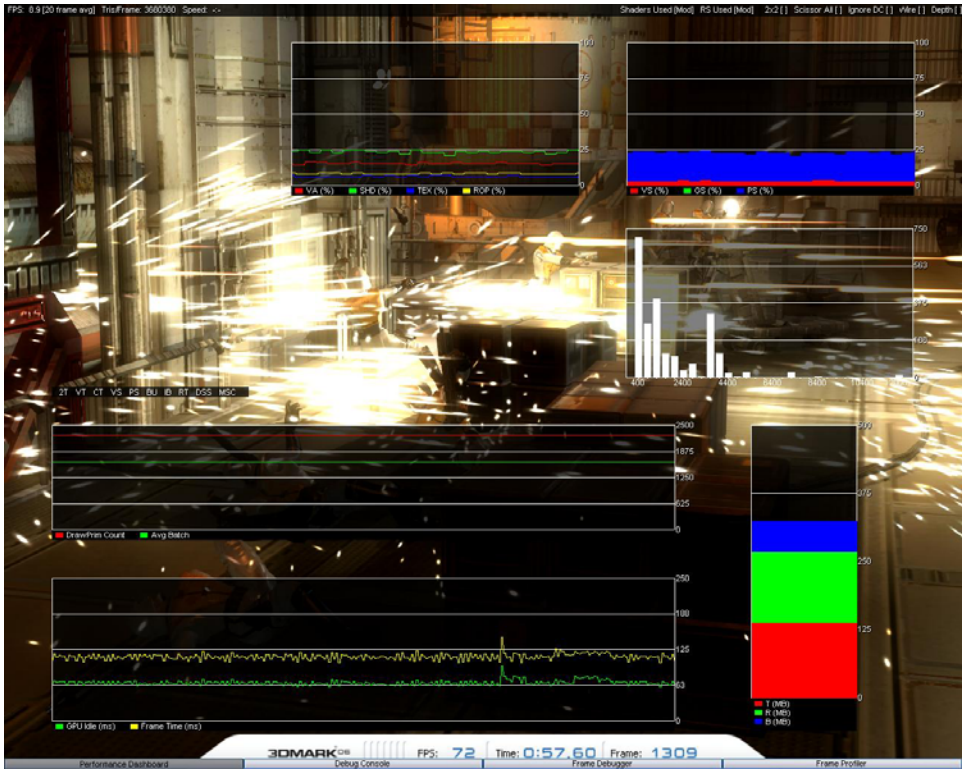
Profiling Effectively with PerfHUD

With all performance tuning, it's very important to identify the largest bottlenecks first. By using PerfHUD's various modes effectively, you can do just that.

When your application first starts with PerfHUD enabled, you'll be in Performance Dashboard Mode (as shown in Figure 4). The Performance Dashboard is a great

place to start because it gives you a broad look at the graphics pipeline, including time spent by the CPU. If your application is CPU-limited, you'll see a big gap between the yellow line ("Total Frame Time") and the red line ("Driver Time"). Another easy way to check for CPU-boundedness is to press "N" to ignore draw calls. If your frame rate doesn't increase, then even an infinitely fast GPU wouldn't help your application run faster – therefore, it is definitely CPU-limited.

For CPU-limited situations, you should use a CPU performance analyzer such as Intel's VTune or AMD's CodeAnalyst to make your CPU code more efficient. For more information on Performance Dashboard Mode, please read Chapter 5.



3DMark06 used with permission from Futuremark Corporation.

Figure 4. PerfHUD Starts in the Performance Dashboard

Once you've verified that your application is not CPU-bound, navigate through your application to the area you want to analyze. If you notice any rendering issues along the way, switch to the Frame Debugger and solve those problems first. The Frame Debugger allows you to step through your scene, one draw call at a time. For each draw call, you can see what geometry, textures, shaders, raster operations are used. Learn more about the Frame Debugger in Chapter 7.

When you notice a performance issue, switch to the Frame Profiler (if you are using a GeForce 6 Series or later GPU) and use the advanced profiling features to identify the bottleneck. The Frame Profiler provides automated performance analysis, giving you very detailed information about your draw calls and time spent in the various GPU stages, as well as other useful GPU statistics. It also allows you to group draw calls into buckets to identify specific types of bottlenecks. If you don't have a

GeForce 6 Series or later GPU, you must manually use the pipeline experiments in the Performance Dashboard to identify the bottleneck.

Enable Your Application

PerfHUD 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 PerfHUD analysis. Additional information about making your application work well with PerfHUD is detailed in the Troubleshooting section at the end of this document.

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

One of the first functions called when setting up your graphics pipeline is the Direct3D CreateDevice() function that creates 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 PerfHUD, a special **NVIDIA PerfHUD** adapter is created. Your application can give PerfHUD permission to analyze it by selecting this adapter. In addition, since some applications might select the **NVIDIA PerfHUD** adapter ID unintentionally and expose themselves to unauthorized analysis, you must select the reference rasterizer as the device type. Your application will not actually use the reference rasterizer as long as you have selected the PerfHUD adapter.

A minimal code change that will enable PerfHUD analysis for your DirectX9 application would be something like this:

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

#if SHIPPING_VERSION
// When building a shipping version, disable PerfHUD (opt-out)
#else
// Look for 'NVIDIA PerfHUD' 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 (strstr(Identifier.Description,"PerfHUD") != 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;
}
```

For DirectX10, use the following code example:

```
#if SHIPPING_VERSION
// When building a shipping version, disable PerfHUD (opt-out)
#else
// Look for 'NVIDIA PerfHUD' adapter
// If it is present, override default settings
IDXGIFactory *pDXGIFactory;
ID3D10Device *pDevice;
HRESULT hRes;
```

```
hRes = CreateDXGIFactory(__uuidof(IDXGIFactory),
(void**)&pDXGIFactory);

// Search for a PerfHUD adapter.
UINT nAdapter = 0;
IDXGIAdapter* adapter = NULL;
IDXGIAdapter* selectedAdapter = NULL;
D3D10_DRIVER_TYPE driverType = D3D10_DRIVER_TYPE_HARDWARE;

while (pDXGIFactory->EnumAdapters(nAdapter, &adapter) !=
DXGI_ERROR_NOT_FOUND)
{
if (adapter)
{
DXGI_ADAPTER_DESC adaptDesc;
if (SUCCEEDED(adapter->GetDesc(&adaptDesc)))
{
const bool isPerfHUD = wcscmp(adaptDesc.Description, L"NVIDIA
PerfHUD") == 0;

// Select the first adapter in normal circumstances or the
PerfHUD one if it exists.

if(nAdapter == 0 || isPerfHUD)
selectedAdapter = adapter;

if(isPerfHUD)
driverType = D3D10_DRIVER_TYPE_REFERENCE;

}
}
++nAdapter;

}

#endif

            if(FAILED(D3D10CreateDevice( selectedAdapter,
driverType, NULL, 0, D3D10_SDK_VERSION, &pDevice)))
                  return E_FAIL;
```

This will enable PerfHUD analysis when you want to use it, and ensure that your application does not use the software reference rasterizer when run normally.

Note: Remember to use the **NVIDIA PerfHUD** device whenever your application enumerates devices, checks for capabilities, and so on. Otherwise you may see rendering errors.

Taking Screenshots

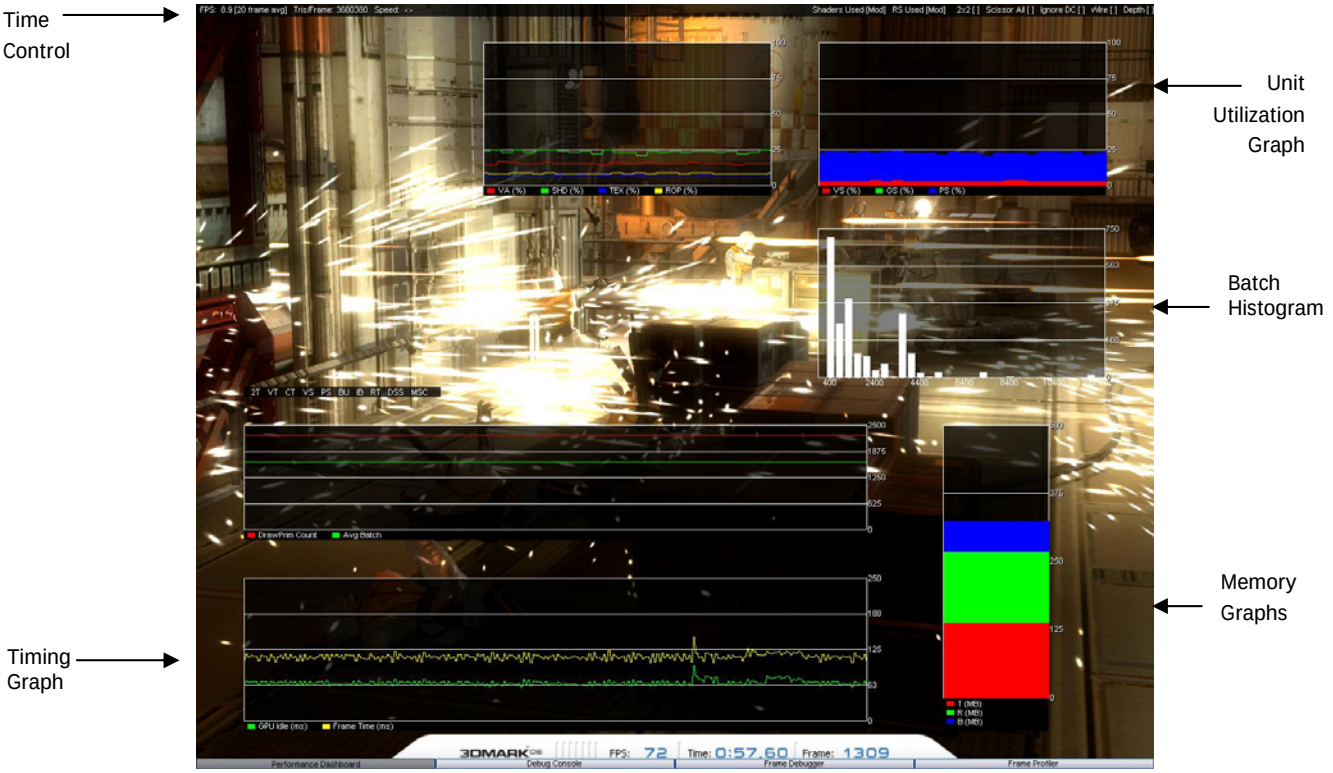
PerfHUD allows you to take a screenshot at any time by pressing the F11 key. The screenshot will be placed in the log directory specified in the PerfHUD launcher, and will be named according to the current day and time.

Chapter 5. Performance Dashboard

This chapter teaches you how to interpret the information displayed by PerfHUD in Performance Dashboard Mode.

Note: Use the Unit Utilization Graph instead of the manual experiments if you are using a GeForce 6 Series or later GPU.

Figure 5 shows the graphs and overlays available in Performance Dashboard Mode.



3DMark06 used with permission from Futuremark Corporation.

Figure 5. PerfHUD Performance Dashboard Mode

The Info Strip

Basic performance metrics are displayed on the top left corner of the screen (see Figure 5). Together these numbers provide a measure of how quickly your application is accomplishing its workload.

Figure 6. PerfHUD Info Strip (top)

Time Control

Notice the speed control icon in the upper left corner of the screen, just below the Info Strip. This control allows you to determine the playback speed of your application. Controlling the time for your application can be very useful when you are zeroing in on a specific frame. Use the following keyboard shortcuts to slow down or speed up your application:

- **NumPad +** Increase speed
- **NumPad -** Decrease speed
- **NumPad Enter** Pause / Continue

Note: PerfHUD "freezes" your application by returning the same value every time your application asks for the current time. This simulates an infinitely fast rendering loop, so the same workload is submitted for each frame.

Note: If your application has implemented a frame rate limiter, you may need to disable this functionality to use the time control, debugging and profiling features of PerfHUD. Please see the FAQ section for more information.

Performance Graphs

PerfHUD 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** Display Help
- **F2** Hide User Interface
- **B** Toggle display of batch size histogram
- **F** Fade the background to improve graph readability
- **H** Hide graphs
- **Y** Hide PerfHUD log output only
- **W** Wireframe
- **D** Depth Complexity

Note: The additional overhead and performance characteristics of the DirectX Debug Runtime make it an inappropriate environment for performance analysis. PerfHUD displays a warning message when it detects that your application is running with the DirectX Debug Runtime.

5.1.1. Reading the Unit Utilization Graph

The scrolling graph in the upper right corner shows utilization of the various units inside the GPU. If you are using a GeForce 6 Series GPU or later you can use this graph to conveniently monitor how your application is using the GPU. If you are using an older GPU you will need to learn how to interpret the timing graphs, described below.

The graph shows the number of milliseconds each unit was busy for each frame. Use this graph to keep tabs on the high level performance characteristics of your application. When you see a potential problem, switch to Frame Profiler Mode to freeze the current frame and analyze unit utilization by draw call. (Learn more about the Frame Profiler in Chapter 8.)

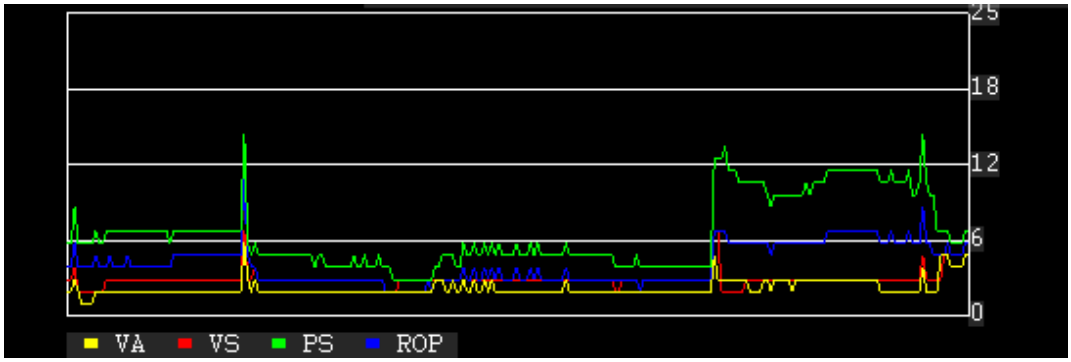


Figure 7. Unit Utilization Graph in Performance Dashboard Mode

- YELLOW = Vertex Assembly Unit
- RED = Vertex Shader Unit
- GREEN = Pixel Shader Unit
- BLUE = Raster Operations Unit

5.1.2. Reading the Timing Graphs

GPUs released before the GeForce 6 Series do not have the internal performance counters required by the Unit Utilization Graph. If you are using one of these older GPUs, you will need to use the manual pipeline experiments and learn to interpret the timing graphs described below.

Scrolling Graphs

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

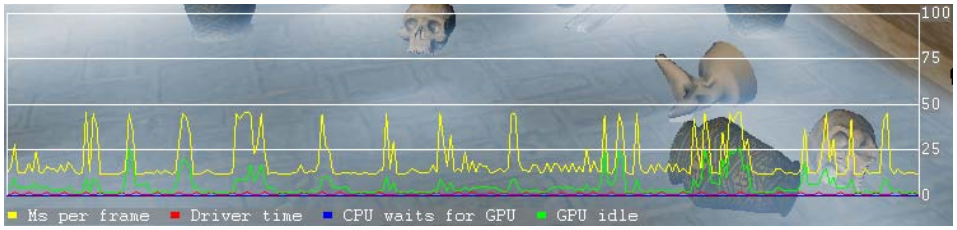


Figure 8. Performance Graphs

- **GPU_IDLE**
  Total amount of time per frame that the GPU was idle
- **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)
- **TIME_IN_DRIVER**
  Total amount of time per frame that the CPU is executing driver code, including **DRIVER_WAITS_FOR_GPU**
- **FRAME_TIME**
  Total elapsed time from the end of one frame to the next - you want to keep the **FRAME_TIME** line as low as possible. For your convenience, the table below lists some common frame times and corresponding frame rates (1 / **FRAME_TIME**).

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

Note: The time gap between the **FRAME_TIME** line and the **TIME_IN_DRIVER** 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.

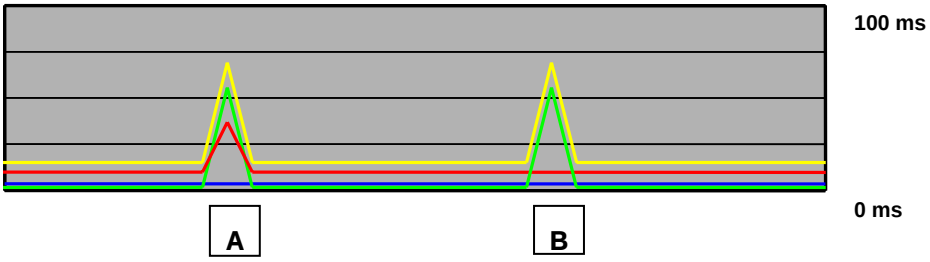


Figure 9. Occasional Spikes

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** and **FRAME_TIME** 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** line spikes and the **TIME_IN_DRIVER** 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 draw calls per frame. This includes calls to `DrawPrimitive, DrawPrimitiveUP` and `DrawIndexedPrimitives.` Using this information to identify performance bottlenecks is discussed in Pipeline Experiments on the next page and in Chapter 9.

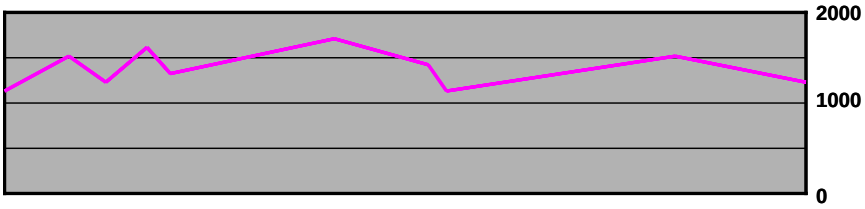


Figure 10. Draw Calls Graph

Batch Size Histogram

The batch size graph only shows up when you turn it on by activating PerfHUD (using your activation hotkey) and pressing the **B** key. 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.

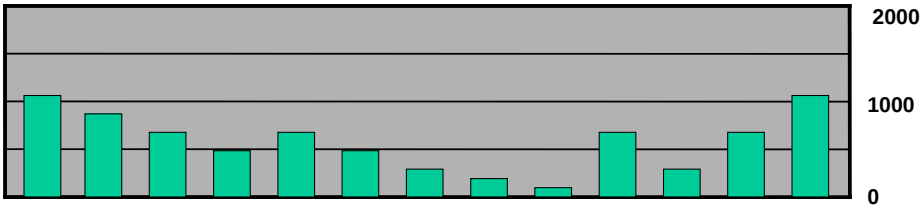
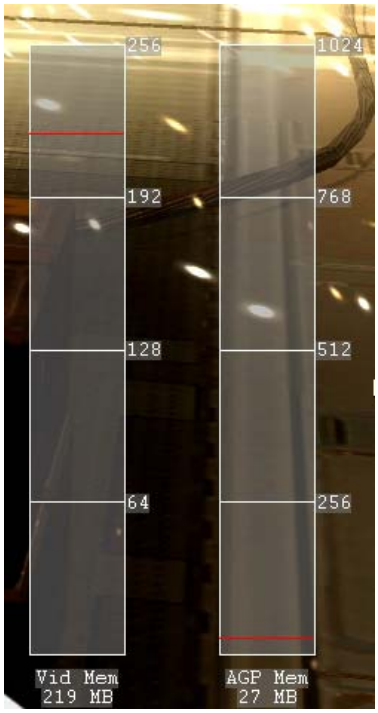


Figure 11. Histogram of Draw Calls

Memory Graphs

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



3DMark06 used with permission from Futuremark Corporation.

Figure 12. Memory Graphs

5.1.3. 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 13. 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.

Pipeline Experiments

Identifying performance bottlenecks requires focusing on certain stages of the graphics pipeline one stage at a time. On GeForce 6 series and later GPUs, you should use Performance Dashboard to identify bottlenecks in the graphics pipeline. These more recent GPUs have special counters designed in that can be used to determine exactly where the bottleneck is. On older GPUs PerfHUD allows you to perform the following experiments to manually identify performance bottlenecks:

- **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.

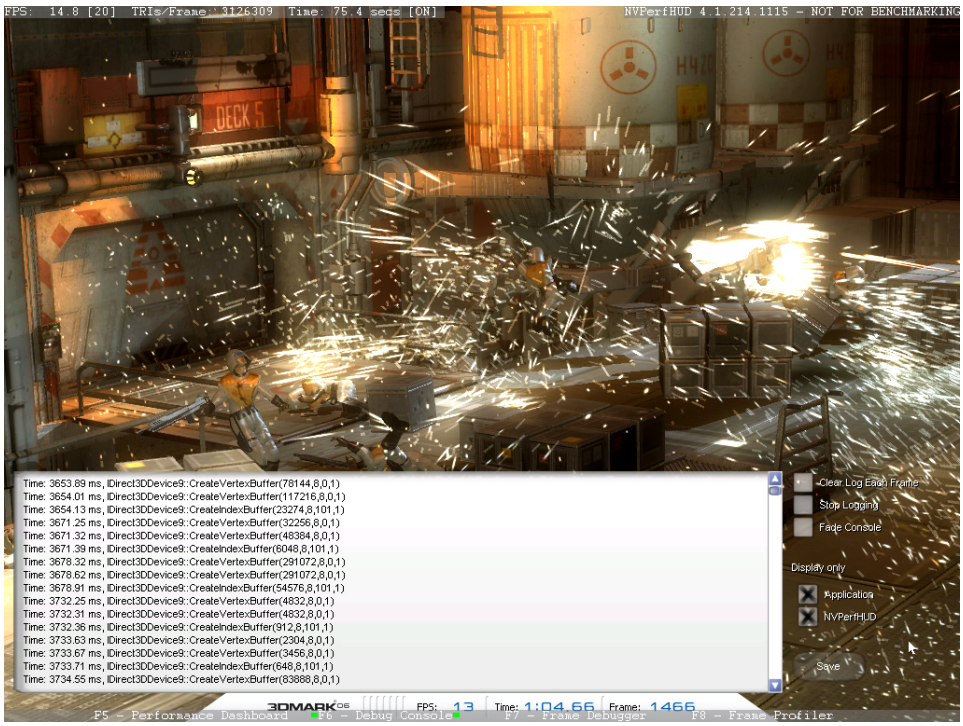
You can also selectively disable pixel shaders by version and visualize how they are used in your application. When a particular shader version is disabled, all shaders in that group are represented by the same color. The shader visualization options provided by PerfHUD are listed below:

1 — Fixed function
2 — 1.1 Pixel shaders
3 — 1.3 Pixel shaders
4 — 1.4 Pixel shaders
5 — 2.0 Pixel shaders
6 — 2.a Pixel shaders
7 — 3.0 Pixel shaders

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 PerfHUD configuration settings.

Chapter 6. Debug Console

This chapter describes the information available to you in the Debug Console mode. The Debug Console screen is shown in Figure 14.



3DMark06 used with permission from Futuremark Corporation.

Figure 14. The Debug Console

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 PerfHUD. Please note that the maximum supported size for debug output strings is 4 KB, and only the first 80 characters of a string will be visible in the Debug Console if the string doesn't contain newline characters. Resource creation events and warnings detected by PerfHUD 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.

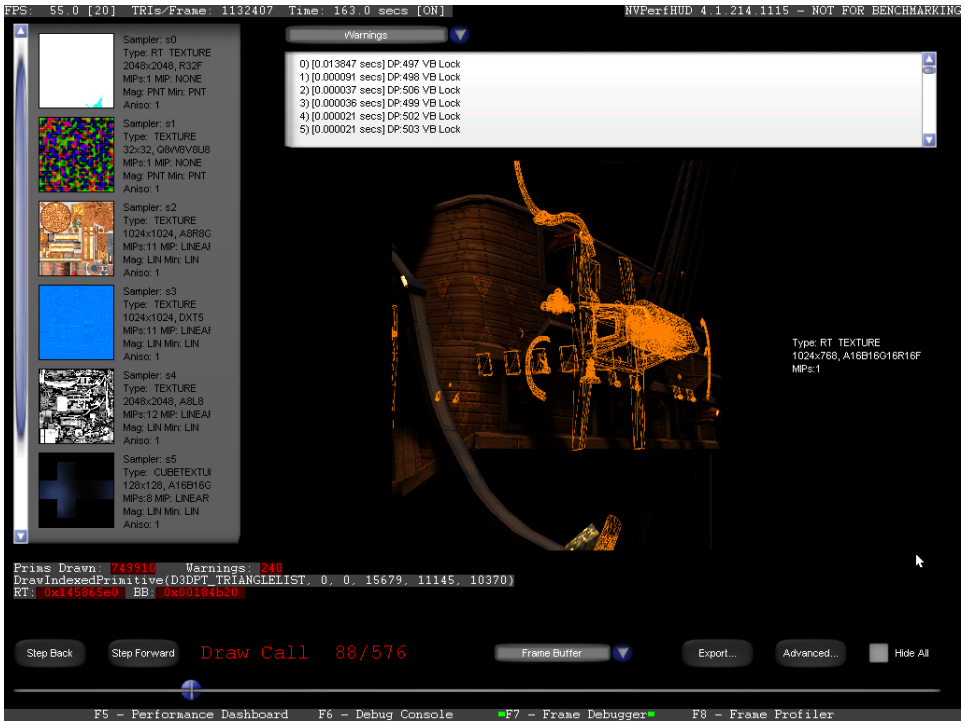
Enabling the "Stop Logging" checkbox causes the console to stop displaying new messages.

You can also choose to display on your application or only PerfHUD in this mode.

Chapter 7. Frame Debugger

This chapter explains how to get the most out of the Frame Debugger and its advanced graphics pipeline State Inspectors. Figure 15 shows the Frame Debugger.

When you first enter Frame Debugger Mode, the results of the first draw call are shown. Use the slider at the bottom of the screen to scrub forward in time and see the results of each successive draw call.



3DMark06 used with permission from Futuremark Corporation.

Figure 15. The Frame Debugger

The geometry associated with the current draw call is highlighted in orange wireframe.

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

- **A** Toggle Advanced / Simple display
- **S** Show Warnings
- **M** D3D Markers

The pull-down menu at the bottom of the screen or a shortcut keys below can be used to select one of the supported rendering modes.

- **W** Wireframe
- **D** Depth Complexity
- **Z** Z-Buffer
- **F** Dest Alpha

Rendering Decomposition

You can use Frame Debugger Mode to navigate to any Draw Call within a frame. When you have identified a frame that has rendering artifacts, you can use Frame Debugger Mode to verify the order in which your scene gets drawn or learn more about what is causing any warnings that arise. When you switch to Frame Debugger Mode, PerfHUD 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 Debugger Mode effectively, your application must behave in a way that PerfHUD can control it. Several requirements are described below. See the Troubleshooting section at the end of this document for additional issues.

Frame Debugger 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. The geometry drawn by the current draw call is highlighted on the screen.

If mouse or keyboard event interception isn't working properly, exit your application and select an alternate API interception option in the PerfHUD 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.

7.1.1. Show Warnings

When **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.

7.1.2. Show D3D Markers

The D3D Markers option allows you to see markers you have set in your application, displayed in a hierarchical tree view (see Figure 16). Clicking on a marker to jumps to the associated draw call. See the Direct3D API documentation for more information on using D3D Markers to instrument your application: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Accurately_Profiling_Direct3D_API_Calls.asp

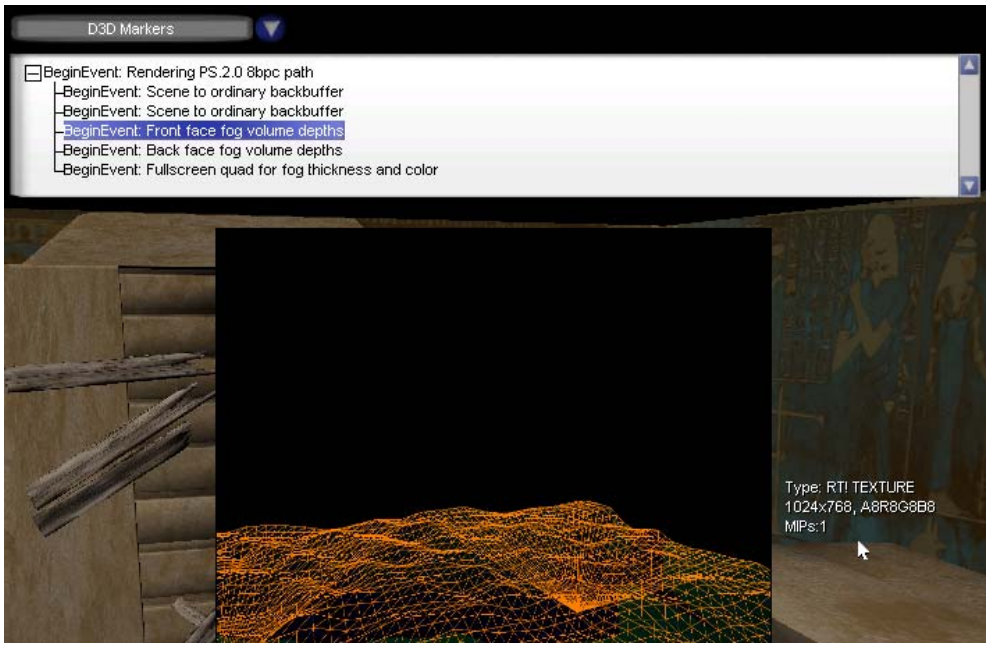


Figure 16. Viewing D3D Event Markers

7.1.3. 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.

7.1.4. Visualization Options

Frame Debugger 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 frame buffer 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, possibly leading to fill-limited performance bottlenecks.
- **W Wireframe**: this option forces wireframe rendering so you can examine the geometric complexity of the scene.
- **F Dest Alpha**: this option allows you to visualize how your application is using destination alpha.

You can also export the current set of data to an XML file for later analysis by clicking on the Export button.

7.1.5. 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 for various GPU pipeline units:

- **1 Vertex Assembly** – 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 PerfHUD 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.

Vertex Assembly State Inspector

When this State Inspector is selected, PerfHUD displays information about the Vertex Assembly Unit during the current draw call.

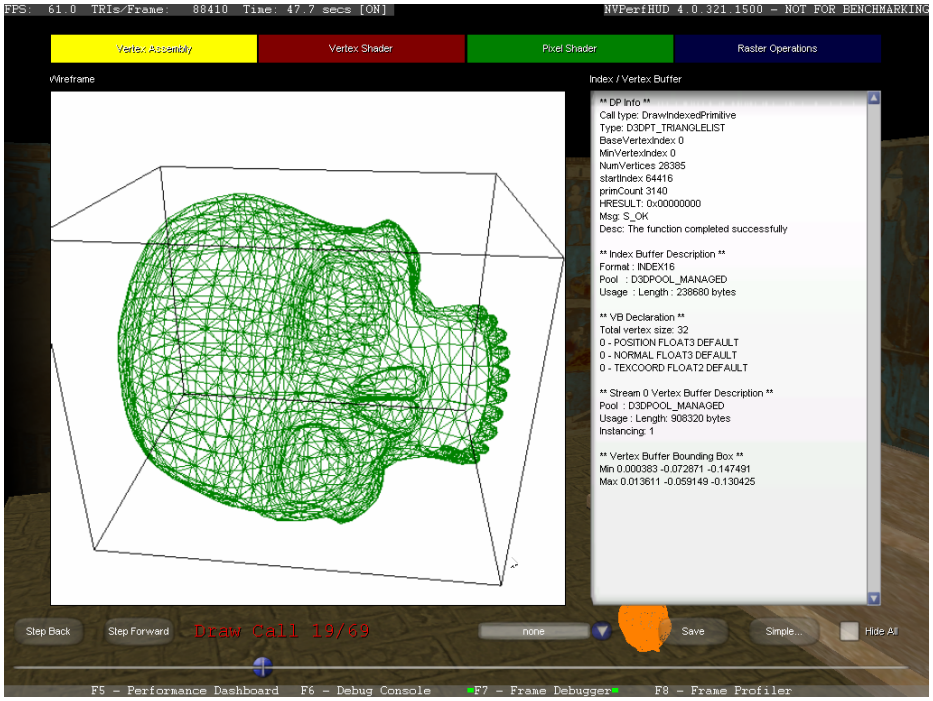


Figure 17. Vertex Assembly Unit State Inspector

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 Vertex Assembly 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.

Vertex Shader State Inspector

When this State Inspector is selected, PerfHUD displays information about the vertex shader used for the current draw call.

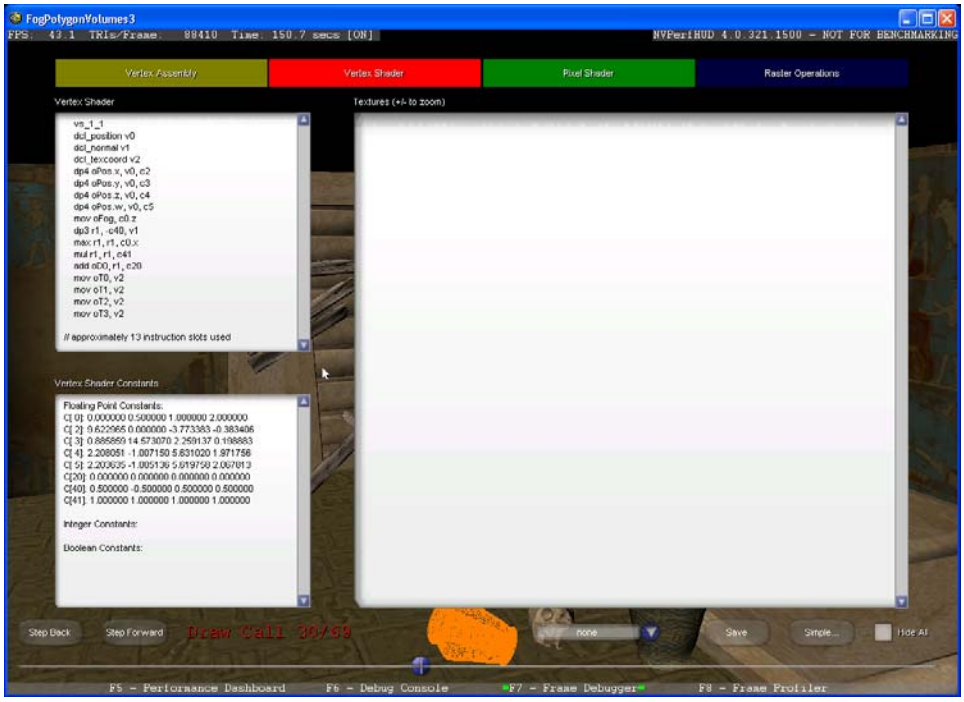


Figure 18. Vertex Shader State Inspector

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**
- Verify that texture states are set correctly

Pixel Shader State Inspector

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

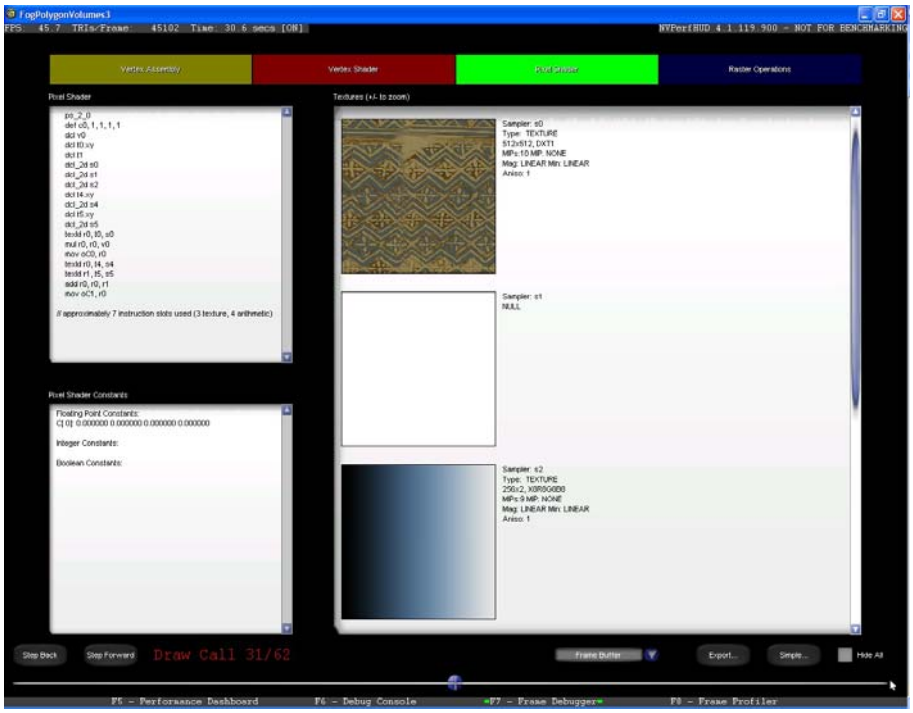


Figure 19. Pixel Shader State Inspector

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
- Verify that texture filtering states are set correctly

Raster Operations State Inspector

When this State Inspector is selected, PerfHUD displays information about the raster operations (ROP) unit for the current draw call. This information is displayed in a collapsible tree to help manage the large amount of data.

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.

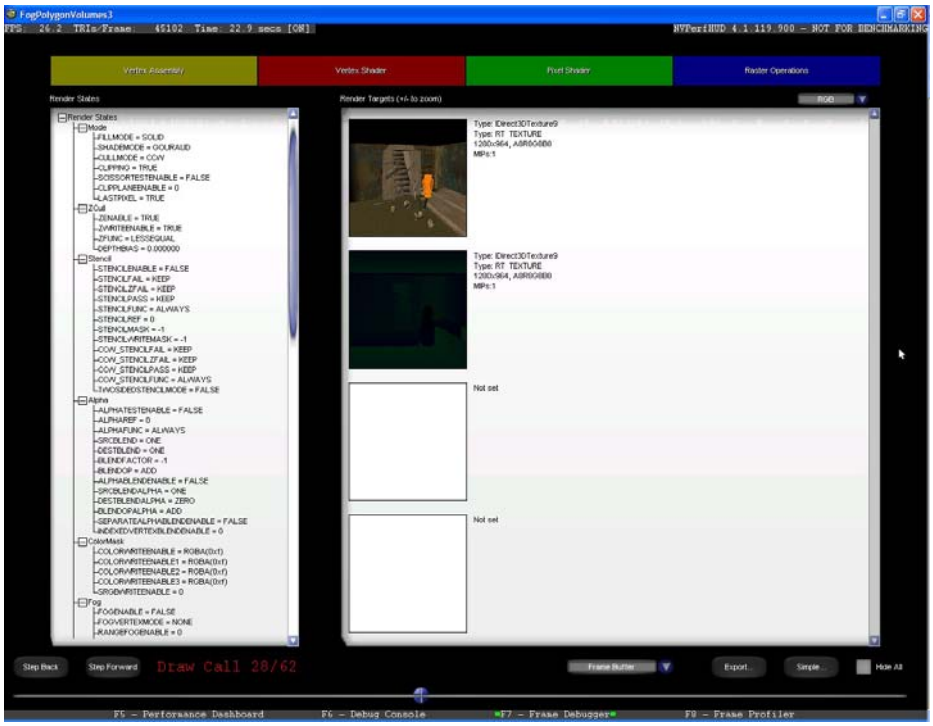


Figure 20. Raster Operations State Inspector

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
  - **`StencilTest`**

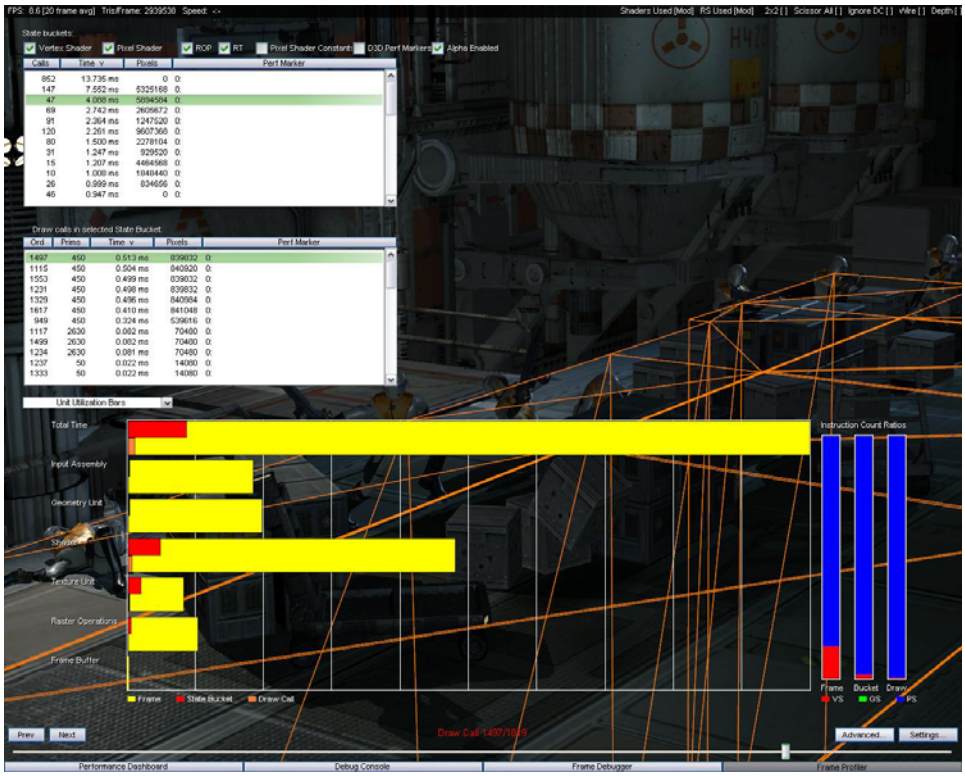
When you are using the Raster Operations State Inspector, you should:

- 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 8. Frame Profiler

The PerfHUD Frame Profiler uses special hardware inside the GPU and instrumentation inside the drivers to measure exactly how your application is using the GPU and report information you can use to identify and remove bottlenecks from your graphics pipeline.

The Frame Profiler is the most powerful and effective way to find bottlenecks in a particular frame because it automatically identifies that frame's most expensive draw calls. In addition, it allows you to access a wealth of detailed information for any particular draw call so you can see how to address suboptimal performance.



3DMark06 used with permission from Futuremark Corporation.

Figure 21. The Frame Profiler

Note: The Frame Profiler is available only on GeForce 6 Series and newer GPUs.

Using the Frame Profiler

When you switch to Frame Profiler Mode (**F8**), PerfHUD forces your application to render the same frame several times and monitors how the driver and GPU are used by each draw call in your application. This information is then used to group draw calls with similar state attributes into "state buckets". Think of state buckets as a "group by bottleneck" operation.

Since all the draw calls in a state bucket share common characteristics, optimizing the bottleneck of the most expensive (that is, time consuming) draw call in state bucket is likely to benefit all draw calls in that state bucket.

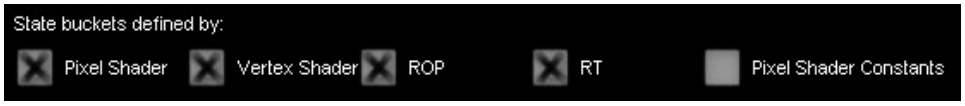
For example, suppose you have a state bucket with two draw calls in it:

- one that draws an object close to the camera (A), and
- another that draws a second, similar object far away from the camera (B).

The object close to the camera will probably take more time to draw. Optimizing for the bottleneck of the most expensive draw call (A) in this state bucket will also benefit the other draw call (B) at times when the second object is close to the camera.

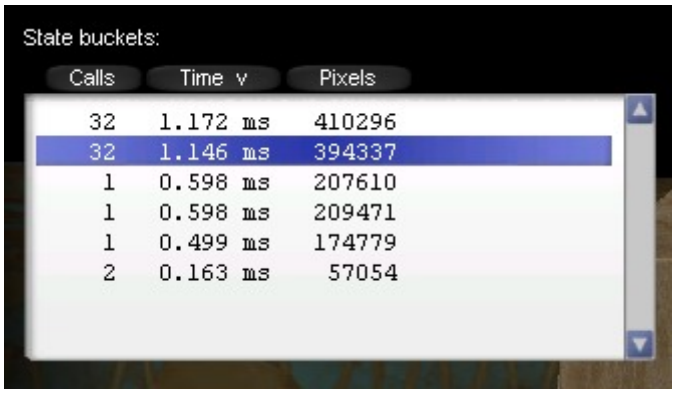
Note: If the number of draw calls in the current frame changes, PerfHUD will prompt you to press the **SPACE BAR** and then reanalyze the current frame. This is an indication that your application is not able to send the same workload repeatedly, and will therefore be very difficult to analyze. See the Troubleshooting section

Initially, you should let PerfHUD use the default state bucket configuration, shown below:



After the initial analysis, you can configure which attributes are used sort draw calls into state buckets manually for a different perspective.

The top list box shows you all the state buckets into which your draw calls have been grouped, sorted by default from most expensive to least expensive. You can click on any column header to sort by that column (in ascending or descending order).



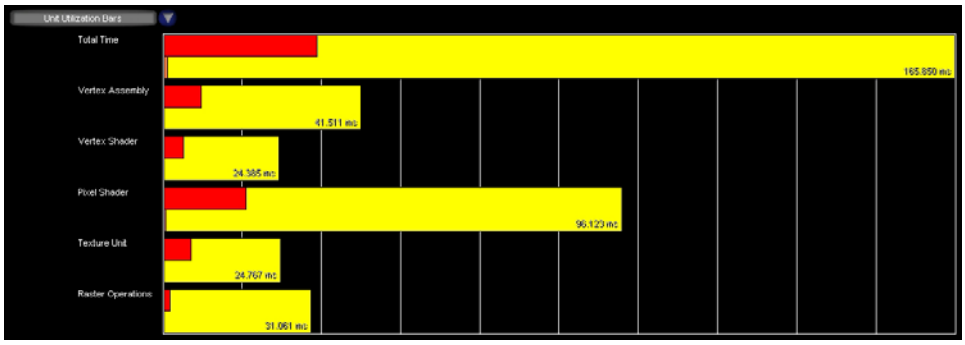
The next list box shows you all the draw calls in the currently selected state bucket. By default the draw calls are sorted from most expensive to least expensive, but you can click on any column header to sort by that column instead (in ascending or descending order).

Clicking on any draw call will cause the slider at the bottom of the screen to jump to that draw call, and the results of that draw call to be highlighted on the screen. You can also drag the slider at the bottom of the screen to a specific draw call to see which state bucket it is in.

Use the pull-down menu below the draw calls list box to select one of several graphs that allow you to better analyze the performance characteristics of the current scene.

8.1.1. Unit Utilization Bars

In this graph, the bar on top represents the entire frame. The bars below that show how busy each unit was during the current draw call and all calls in the same state bucket.



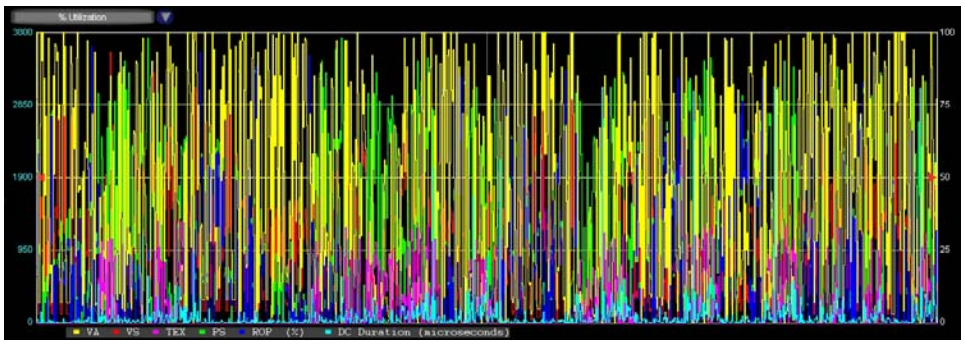
The **yellow section** of each bar represents the total time in ms for the frame.

The **red section** represents time used by all calls in the current state bucket.

The **orange section** represents time used by the current draw call.

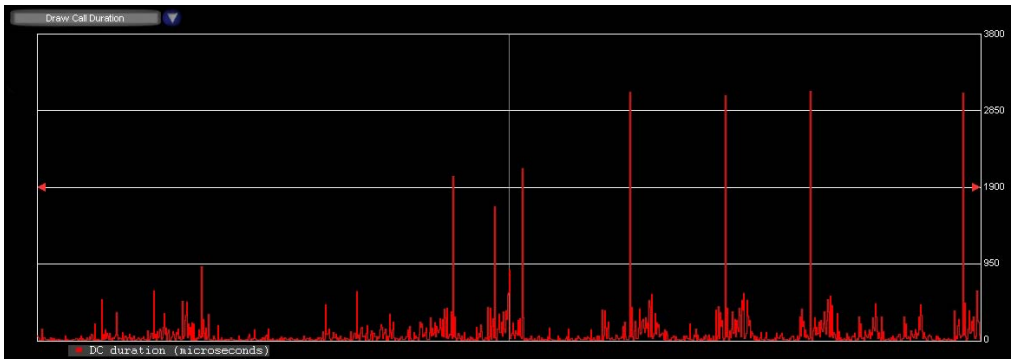
8.1.2. Unit Utilization Graph

This graph is similar to the Utilization Graph in the Performance Dashboard that helps you monitor GPU unit utilization on a frame-by-frame basis. The Frame Profiler is focused on analyzing a single frame, so this graph shows GPU unit utilization in milliseconds for each draw call in the current frame.



8.1.3. Draw Call Duration Graph

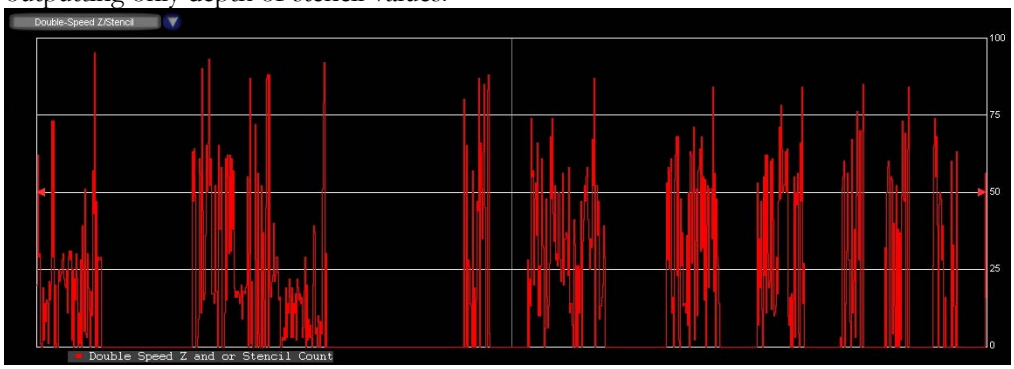
This graph shows the number of ms it took to complete each draw call in the frame. You can use the slider to quickly navigate through the scene looking for potential problems.



8.1.4. Double-Speed Z/Stencil Graph

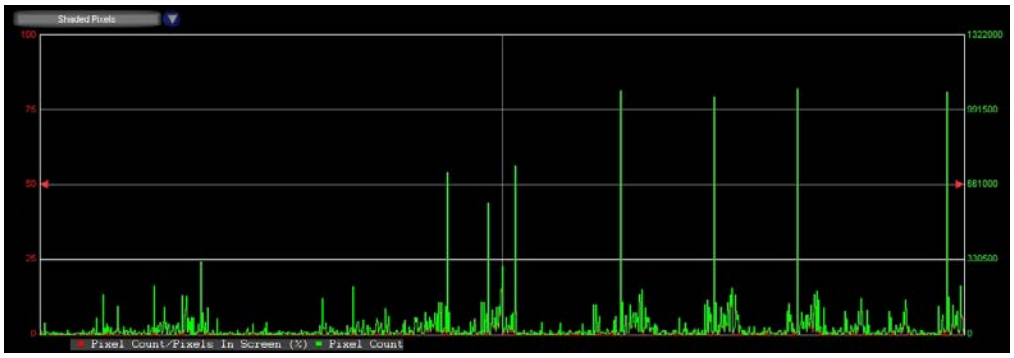
The graph shows you the number of milliseconds during each draw call for which the double-speed depth/stencil "fast path" was active. This refers to the ability of NVIDIA GPUs (from GeForce FX onwards) to render at double speed when

outputting only depth or stencil values.



8.1.5. Pixel Count Graph

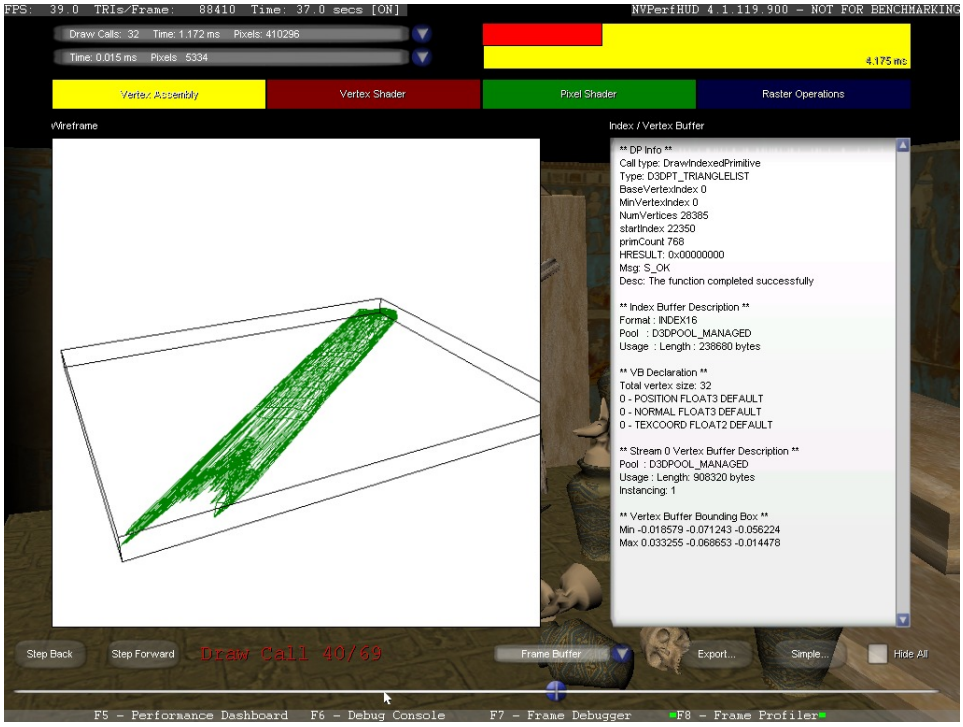
This graph shows the number of pixels actually rendered to the screen for each draw call, both as a percentage of the screen and as the number of pixels drawn per GPU clock cycle.



Frame Profiler Advanced View

You can use the Advanced button at the bottom of the screen to access advanced State Inspectors. When you do this from Frame Profiler Mode, you can select draw calls by state bucket and view how expensive each draw call was in the context of the current frame. The colored bars at the top are the same style as the unit utilization bars described above.

The top list box in this view allows you to choose between state buckets, and the bottom list box allows you to select different draw calls within a state bucket.



Chapter 9. Analyzing Performance Bottlenecks

Graphics Pipeline Performance

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.

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 PerfHUD should be used to identify performance bottlenecks and save you from wasting time.

9.1.1. Pipeline Overview

At the highest level, the pipeline is broken into two parts: the CPU and GPU. Figure 22 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).

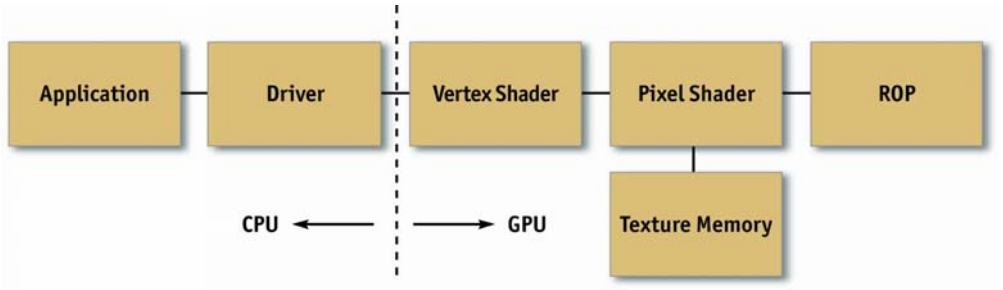


Figure 22. Pipeline Overview

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 PerfHUD 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

9.1.2. 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 23 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.

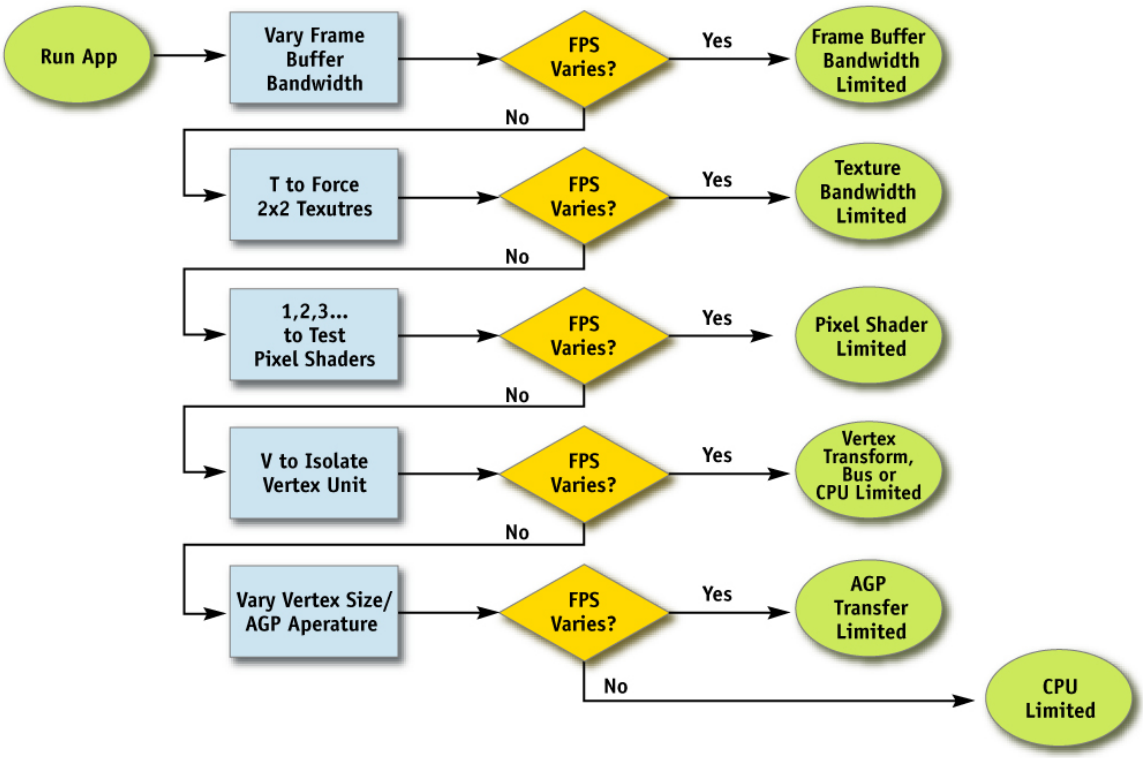


Figure 23. Identifying Bottlenecks

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.

9.1.3. 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.

9.1.4. 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 PerfHUD 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.

9.1.5. 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 PerfHUD 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 Dashboard 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.

9.1.6. 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 PerfHUD 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.

9.1.7. 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.

9.1.8. 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 PerfHUD 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, PerfHUD 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 24 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 5.1.3.

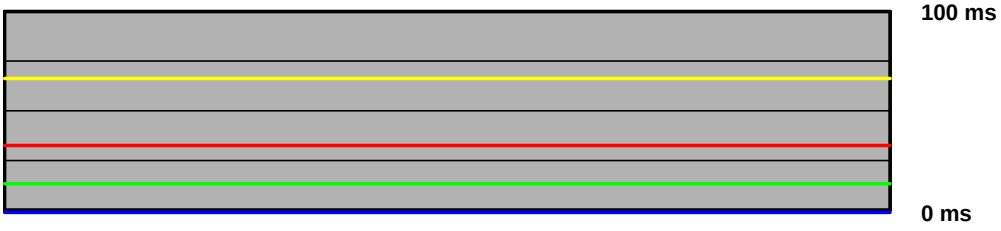


Figure 24. Too Many Calls to the Driver

Figure 24 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.

Chapter 10. Bottleneck Optimizations

Once you have identified the bottleneck, you 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.

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.

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

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 PerfHUD. Pressing **B** displays a histogram showing the distribution of number of triangles per draw call per frame. Figure 25 shows an application that probably performs poorly because it has too many draw calls with a small number of primitives.

Figure 25. Many Small DrawPrimitive Calls

To reduce the number of DrawPrimitive 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) polygon 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 thathave 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.

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.

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 applying LOD to 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.

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.

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 mipmap level? 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 PerfHUD memory graph can help diagnose this problem, as it shows the amount of memory allocated by the driver in various heaps.
- **Always use mipmapping on any surface that may be minified.** Mipmapping delivers better image quality by reducing texture aliasing. A variety of filters can be used to create high-quality mipmaps that do not look blurry. NVIDIA provides a suite of texture tools to aid you in optimal mipmap 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 mipmapping, 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 anisotropic filtering levels, the GPU is doing extra work that does not improve the visual quality. If you are texture bandwidth limited, use the lowest anisotropic filtering level that gives you good enough image quality. In an ideal application should have texture-specific anisotropic filtering settings.
- **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 mipmap level transitions are not readily discernable, turn trilinear filtering off to save fillrate.

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 cube maps.

- **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 11. Troubleshooting

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

Known Issues

- PerfHUD does not handle multiple devices. It only supports the first device created with **`Direct3DCreate9()`**.
- PerfHUD may crash when using software vertex processing.
- An application that uses **`rdtsc`** natively won't be able to function properly with the Frame Debugger Mode of PerfHUD – see Troubleshooting below for possible solutions.
- Frame Debugger Mode and Frame Profiler Mode require 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 PerfHUD 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 Debugger Mode or Frame Profiler Mode, if you deactivate PerfHUD (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 PerfHUD is active and you click on the close button.

Frequently Asked Questions

PerfHUD says my application is not enabled for PerfHUD analysis.
To ensure that unauthorized third parties do not analyze your application without your permission, you must make a minor modification to enable PerfHUD analysis. Refer to the Getting Started section of this User Guide for instructions.
No data is reported in the Unit Utilization Graph in Performance Dashboard mode and/or Frame Profiler mode doesn't seem to work.
Both the Unit Utilization Graph and Frame Profiler require performance signals from PerfKit. Make sure PerfKit is installed and you are using a GPU that is supported by PerfKit.
My application does not respond while PerfHUD is active.
When PerfHUD 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.
My application does not respond while PerfHUD is active.
When PerfHUD 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 PerfHUD header across the top of my screen, but it doesn't respond to my activation hotkey.
PerfHUD 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 PerfHUD. Note that in Win2K PerfHUD 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 PerfHUD such as bottleneck identification experiments, shader visualization, etc.
PerfHUD messes up alpha and some rendering states.
PerfHUD 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 PerfHUD is unable to communicate properly with the display driver. Verify that you are using the latest version of PerfHUD and running the latest NVIDIA display drivers.

In the Frame Profiler's Advanced mode, why can't I use the RGB dropdown to visualize the individual channels of render targets in Raster Operations?

Currently the RGB dropdown only works for textures. It will be grayed out unless at least one texture is present.

Can I use PerfHUD without an instrumented driver?

Yes, PerfHUD will work with a normal driver, but you will not get access to performance counters. Therefore, the graphs in the Performance Dashboard will not work, and automated performance analysis in the Frame Profiler will not work either.

However, you can still use the pipeline experiments in the Performance Dashboard, as well as the Debug Console and Frame Debugger.

Please note that you must still have an NVIDIA GPU in order to use PerfHUD, whether you are using an instrumented driver or not.

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

If you are using very old drivers you may see portions of the old PerfHUD 1.0 graphs super-imposed on top of your PerfHUD graphs. Upgrading your drivers to 71.8x or later should fix the problem.

Some objects in my scene continue to animate when my application is frozen.

The PerfHUD time control feature stops the clock for your application, allowing you to 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 Debugger Mode, Frame Profiler Mode, or the time control features in Performance Dashboard.

If your application has implemented a frame rate limiter, you may need to disable this functionality to use the time control, debugging and profiling features of PerfHUD.

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

What else do I need to know about PerfHUD?

- Multi sampled render targets at not displayed in Frame Debugger Mode.
- PerfHUD 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 PerfHUD 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 PerfHUD.

PerfHUD@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.

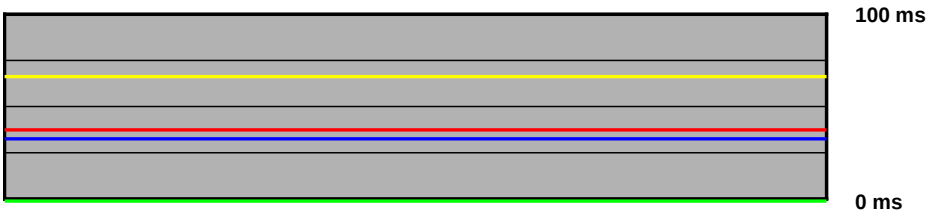


Figure 26. Driver Waiting for the GPU

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.

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