PerfHUD 6 User Guide

DU-01231-001_v09
June 2008

DEVELOPMENT

Table of Contents

Chapter 1. New in PerfHUD 6

This chapter describes all the new features of PerfHUD 6, including:

- Support for SLI
- Support for 32-bit apps on 64-bit operating systems
- Support for the latest NVIDIA GPUs
- Support for new features in DirectX10 and Windows Vista, including NVIDIA's Unified Instrumented Driver
- More signals
- Better and deeper display of textures
- Better on-the-fly editing and comparisons of in-game HLSL shaders

And that's just the tip of the iceberg.

NVIDIA's performance engineers have been working hard to make this the best PerfHUD release ever. We hope you enjoy using it and that it proves itself to be as "mission critical" to your development process as earlier versions have been.

We are especially interested in your opinions! As you explore and use PerfHUD, be sure to give us your feedback at perfhud@nvidia.com.

Hardware Support and Driver Changes

SLI Support

Support for dual GPU SLI is now supported in the form of new Performance Dashboard signals, and the Dependencies view in the Frame Debugger. Quad SLI is not currently supported.

GPU Support

Support for GeForce 8800GT, GeForce 9600GT, and GeForce 9800GX2 cards has been added.

Unified Driver on Windows Vista

For Windows Vista driver releases after Rev 173, a special instrumented driver is no longer required. This should be a welcome advance to all Vista developers, especially those using laptop displays. XP Users will still need to use the instrumented driver provided with NVIDIA PerfKit.

32-bit Applications on 64-bit Operating Systems

PerfHUD now support the analysis of 32-bit applications under a 64-bit operating system.

New Performance Dashboard Features

Signals

PerfHUD 6 adds several new signals that you can graph on the Performance Dashboard:

- API Calls per Draw Call/Frame
- State changes per Draw Call/Frame
- Unique Shaders / Frame
- Shader Changes per Draw Call/Frame
- Unique Textures/Vertex Buffers/Index Buffers/Constant Buffers per Draw Call/Frame
- Redundant state changes per Draw Call/Frame
- Locks (total and IB/VB/Texture/Constant Buffer) per Draw Call/Frame
- Texture/VB/IB/Constant Buffer changes per Draw Call/Frame
- Constant updates per Draw Call/Frame
- Clear Count per Frame

SLI Graphs

The Performance Dashboard now supports new SLI graphs, which allow you to view signal values across multiple GPUs. You can view the signals individually, or choose from a variety of methods to combine the values, including average, sum, minimum, and maximum.

- To create an SLI graph, right-click on an empty area in the Performance Dashboard, and then choose “Create New SLI Graph” from the resulting context menu.
- PerfHUD will automatically detect if you are running on an SLI configuration and will only show SLI graphs if multiple GPUs are present and enabled.

Save and Load Graph Layouts

You can now save and load multiple Performance Dashboard graph layouts. See the Performance Dashboard Customization section of this document for more information.

New Frame Debugger Features

Texture and Sampler State Editing

You can now replace textures on a per-texture with a variety of useful debug textures, including 2x2 textures, and mipmap visualization textures. See documentation on the Frame Debugger’s Texture View for more information.

Sampler state can also be overridden in Vertex, Geometry, and Shader Inspectors in the Frame Debugger's Advanced Shader Inspectors .

Texture Visualization

Textures can now be visualized in a full screen mode. See the Frame Debugger's Texture Viewer for more information.

API Call List

PerfHUD 6 adds an interactive list of API Calls to the Frame Debugger. For more information, please see the Frame Debugger's API Call List.

Dependency View

PerfHUD 6 can now graphically show you dependencies between different draw calls. Dependencies have performance impacts on both single GPU and SLI systems. Please see the Frame Debugger's Dependency View for more information.

Improved D3D Perf Event View

Direct3D Performance Events can be used to indicate key points in your frame, as well as locations where you'll want to do debugging or profiling.

PerfHUD can now generate breaks on a user-specified Perf Event, so you can break into your debugger only when a Perf Event is hit, or disable your state overrides for particular stretches of API calls.

Please see the documentation on the Frame Debugger's Perf Event View for more information.

New Frame Profiler Graphs

New in the Frame Profiler is the CPU and GPU Timings Graph. See documentation on the Frame Profiler for more information.

New Hotkeys

PerfHUD 6 sports many new hotkeys, both to support dedicated keyboard users, and to access some of PerfHUD's newest features.

Global Hotkeys

- **F1:** Show help
- **F2:** Hide/show PerfHUD UI
- **F4:** Instant Feedback™
- **F5:** Switch to Performance Dashboard
- **F6:** Switch to Debug Console
- **F7:** Switch to Frame Debugger

- **F8:** Switch to Frame Profiler
- **F9:** Toggle edited shaders
- **F10:** Toggle edited renderstates
- **F11:** Capture screenshot
- **Shift+F11:** Capture screenshot (without HUD)

Performance Dashboard Hotkeys

Several new hotkeys for real-time experiments and for adjusting FPS computation are now available.

Common Frame Debugger/Profiler Hotkeys

These hotkeys work in both the Frame Debugger and Frame Profiler.

- **Ctrl + A:** Toggle Advanced mode.
- **Ctrl + H:** Toggle Highlight current draw call
- **Right/Up-arrow:** Increment draw call by one
- **Left/Down-arrow:** Decrement draw call by one
- **Home:** Jump to draw call 0
- **End:** Jump to last draw call
- **PageDown:** Jump back by 10 draw calls (-10)
- **PageUp:** Jump ahead by 10 draw calls (+10)

Advanced Frame Debugger Hotkeys

All context menus support keyboard controls. Use the **up/down arrow keys** to move the selection up and down, **Enter** to execute the currently selected option, and **Escape** to close the menu. Some menus also have hotkeys.

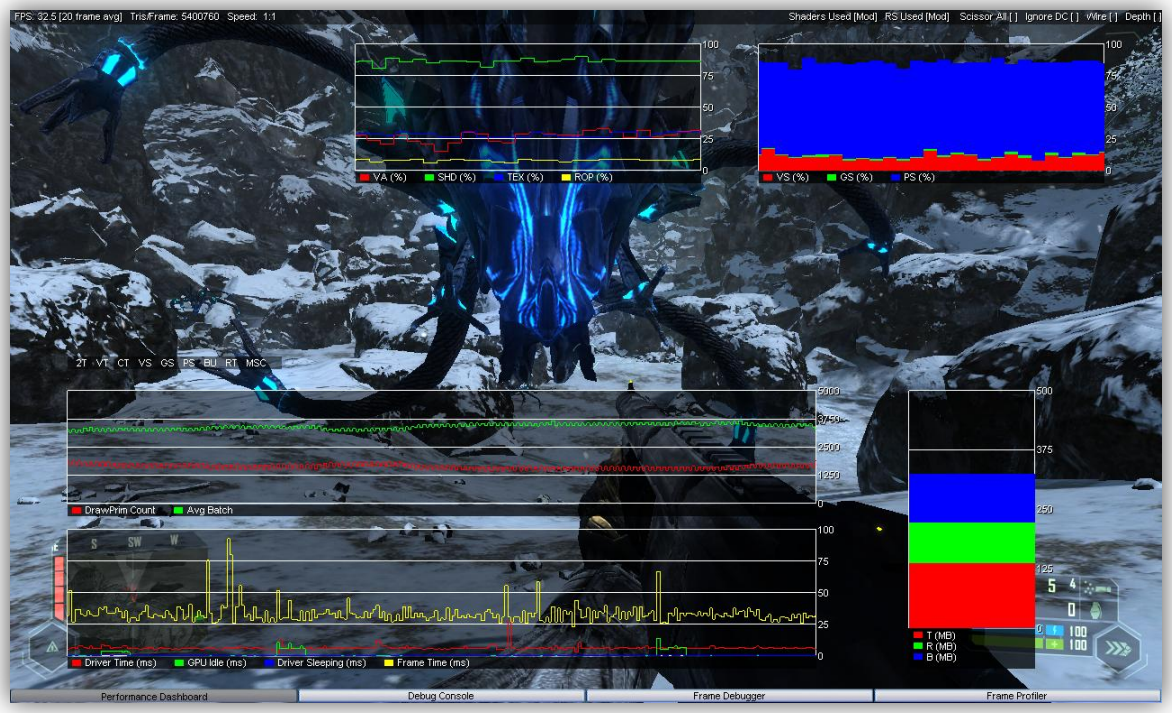
- Texture View
  - **Ctrl + Up/Down:** Select the previous/next texture on this draw call.
  - **Ctrl + Plus/Minus:** Increase/Decrease the size of the selected texture.
  - **Ctrl + Shift + Up/Down:** Select the previous/next render target.
  - **Ctrl + Shift + Plus/Minus:** Increase/Decrease the size of the selected render target.
  - **Alt + T/R:** Bring up the context menu of the currently selected texture or render target respectively.
    - Context Menu Hotkeys:
      - **'O'** for original texture
      - **'2'** for 2x2 texture
      - **'B'** for black texture
      - **'W'** for white texture
      - **'M'** for color mipmap texture.
- Advanced Inspectors

- **Ctrl + Tab/Ctrl + Shift + Tab:** Cycle to the next/previous advanced inspector.

Chapter 2. Performance Dashboard

The Performance Dashboard allows you to monitor and tweak your application while it runs in real-time. You can graph GPU and driver signals and performance counters, as well as tweaking your application with real-time experiments.

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



Crysis used with permission from Crytek. © Crytek GmbH. All Rights Reserved. Crysis and CryENGINE are trademarks or registered trademarks of Crytek GmbH in the U.S and/or other countries.

The Info Strip

General performance metrics are displayed in a status bar across the top of the screen. Together these numbers provide a measure of how quickly your application is accomplishing its workload.

The number of frames used to compute the frame rate shown is, by default, 20. You can adjust this number using the following keys:

- **[** : reduce number of frames to average for FPS computation
- **]** : increase number of frames to average for FPS computation

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
- **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.

Real-time Experiments

Real-time experiments offer a quick and easy way to quickly gather information about where your application's performance issue lies. With one keystroke, you can replace all textures in your application, or make your application ignore all draw calls. By monitoring the frame rate differences during these experiments, you can get more insight into the bottlenecks.

Note: On GeForce 6 and later GPUs, the Frame Profiler offers more automated and accurate performance information on a per-draw call basis.

The following experiments are available in PerfHUD 6:

- **Ctrl + T: Isolate the texture unit**
  This experiment replaces every texture in the scene with a tiny (2×2) texture. If the frame rate increases dramatically your application performance is limited by texture bandwidth.
- **Ctrl + 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.

- **Ctrl + N: Eliminate the GPU**
  This experiment 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.
- **Ctrl + M: Eliminate Geometry**
  This experiment reduces the geometry in every draw call to a single triangle. This reduces the amount of geometry processing your GPU is doing.
- **Ctrl + D: Show Depth Complexity**
  This experiment shows the amount of overdraw in your scene. The more overdraw you have, the more work your GPU is doing that results in no visual improvement to the scene.
- **Ctrl + W: Toggle Wireframe View**
  This experiment shows your entire scene in wireframe with no shading.
- **Ctrl + Shift + T: Toggle Mipmap Visualization View**
  This experiment replaces every texture in the scene bound to slot zero with a mipmap visualization texture. Each mip level is a different color. Mipmap level 0 (the highest-resolution mip level) is shown in red, while the lowest-resolution mipmap level is shown in blue. Levels in between are shown in various other colors.
  If some of the color textures in your application are not bound to slot zero, please use the texture overrides in the Texture View of the Frame Debugger to override individual textures with a mipmap visualization texture.
- **Ctrl + 1: Color fixed function pixels with Red**
- **Ctrl + 2: Color ps_1_1 in light green**
- **Ctrl + 3: Color ps_1_3 in green**
- **Ctrl + 4: Color ps_1_4 in yellow**
- **Ctrl + 5: Color ps_2_0 in light blue**
- **Ctrl + 6: Color ps_2_a in blue**
- **Ctrl + 7: Color ps_3_0 in orange**
- **Ctrl + 8: Color ps_4_0 in red**

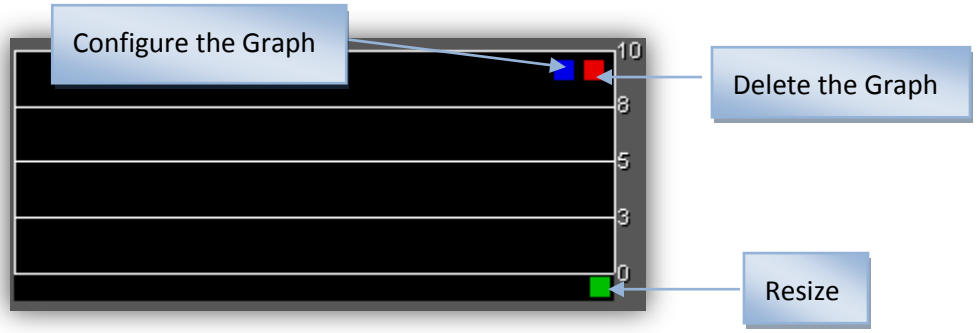
Using the **Ctrl + 1..9** shortcuts both disables the specified shader type as well as coloring those pixels in the specified color. You can use this to both visualize how and where shaders of that version are used, as well as estimating the performance impact of each shader version in your application.

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.

Customizing your Graphs and Layouts

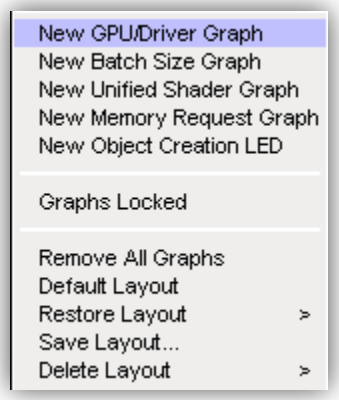
Graphs and graph layouts can be customized in a variety of ways. You can move the graphs, customize their content and display properties, and, new in PerfHUD 6, save your own custom layouts.

Hovering your mouse over a graph will cause several buttons to appear, which you can use to **delete, resize, and configure** the graph.



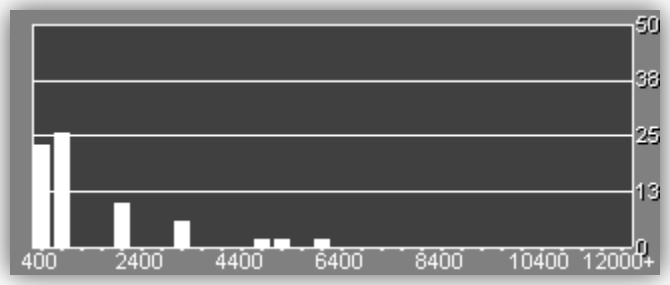
You can also **move** a graph by **left-clicking** anywhere on the graph and then dragging the graph to the desired location.

Right-clicking anywhere in the Performance will bring up a context-menu, which you can use to **add new graphs** or **manage your layouts.**

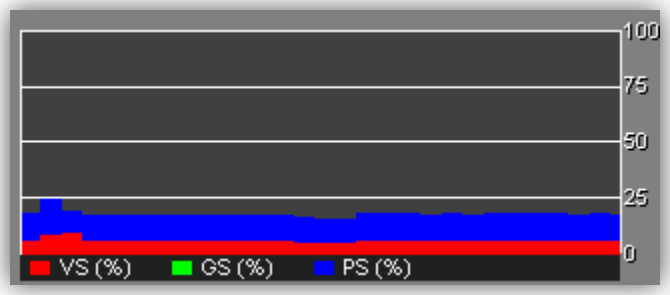


- **New GPU/Driver Graph:** Each graph of this type can be configured to display up to 4 signals from the GPU or driver, by clicking the blue button in the upper left of the graph.

- **New Batch Size Graph:** This graph displays a histogram of the number of primitives per draw call. In general, an application should draw fewer, larger batches. You can configure this graph to change the scale and



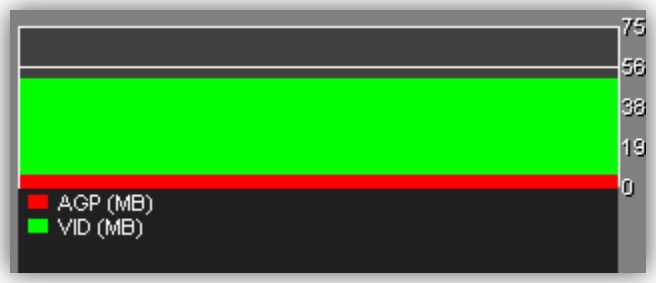
New Unified Shader Graph: Create a graph showing relative usage of the unified shader unit between vertex, geometry, and pixel shaders. (GeForce 8 and later)



- **New Memory Request Graph:** This graph shows the breakdown between AGP and on-board video memory.

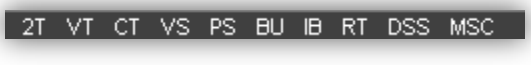
  On Windows Vista, the memory request graph has the following labels:

  - **T** = Textures
  - **R** = Render Targets (Any renderable surface)
  - **B** = Buffers (vertex, index, constant buffers)



- **New Object Creation LED:** The Object Creation LED blinks each time an object is created. Dynamic creation of resources in Direct3D is generally bad for performance and should be avoided whenever possible.

  The types of resource creation events monitored are:



  - **2T** 2D textures created using CreateTexture().

- **VT** Volume textures created using CreateVolumeTexture().
- **CT** Cubemap textures created using CreateCubeTexture().
- **VS** Vertex shader created.
- **GS** Geometry Shader created.
- **PS** Pixel shader created.
- **BU** Buffer created.
- **IB** Index buffers created using CreateIndexBuffer().
- **RT** Render targets created using CreateRenderTarget().
- **DSS** Depth stencil surfaces created using CreateDepthStencilSurface().
- **MSC** Any object creation that doesn't fit into the other categories.

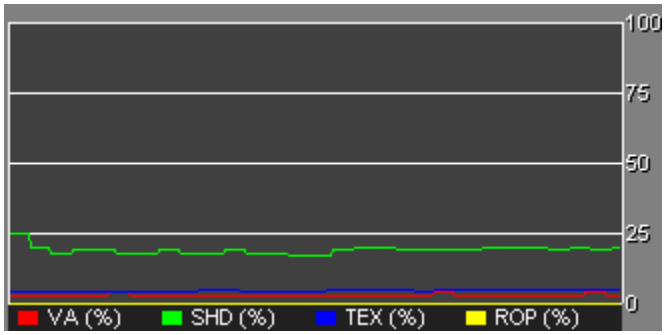
- **Graphs Locked:** Disables the ability to move or resize the graphs.
- **Remove All Graphs:** Removes all graphs from this layout.
- **Default Layout:** Resets the current layout to the PerfHUD shipping default.
- **Restore Layout:** Click this menu item to show the list of layouts available.
- **Save Layout…:** Click this menu button to save and name the current graph layout.
- **Default Layout:** Restore your current layout to the layout that ships standard with NVIDIA PerfHUD 6.

Interpreting the Default Graphs

Note: The default layout of the Performance Dashboard contains several useful graphs. Some of these graphs may not be available on pre-GeForce 6 hardware.

The Unit Utilization Graph

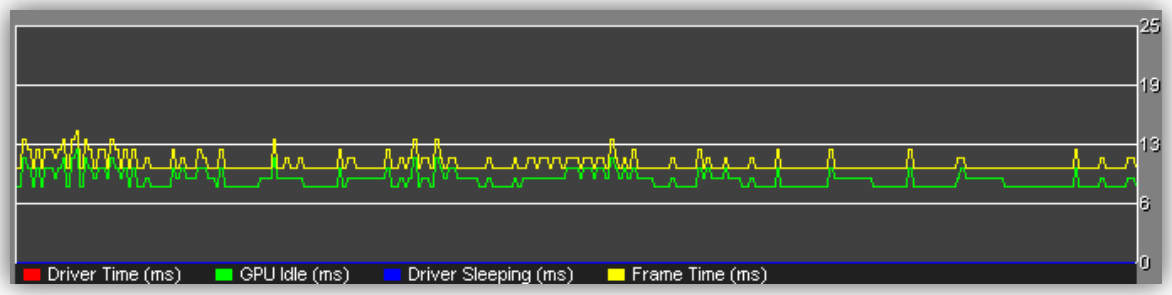
This graph is very useful to keep tabs on the high level performance characteristics of your application. When you see a potential problem, switch to the Frame Profiler to freeze the current frame and analyze unit utilization by draw call. Each line represents the percentage utilization of the specified GPU unit.



- VA: Vertex Assembly Unit

- SHD: Vertex Shader Unit
- TEX: Pixel Shader Unit
- ROP: Raster Operations Unit

The Timing Graph



- **Driver Time (ms)**
  Total amount of time per frame that the CPU is executing driver code, including Driver Sleeping (ms).
- **GPU Idle (ms)**
  Total amount of time per frame that the GPU was idle
- **Driver Sleeping (ms)**
  Accumulated elapsed time when the driver had to wait for the GPU
  (See Appendix A for more information on why this happens)
- **Frame Time (ms)**
  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.

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 (ms) line and the Driver Time (ms) line is the time consumed by application logic and the OS.

You may see occasional spikes in this graph. They are usually caused by an operating system process running in the background performing a hard disk access, texture upload, or operating system context switch.

As long as the spikes are sporadic, the situation is normal. However, if they occur regularly, your application may be performing CPU-intensive operations inefficiently.

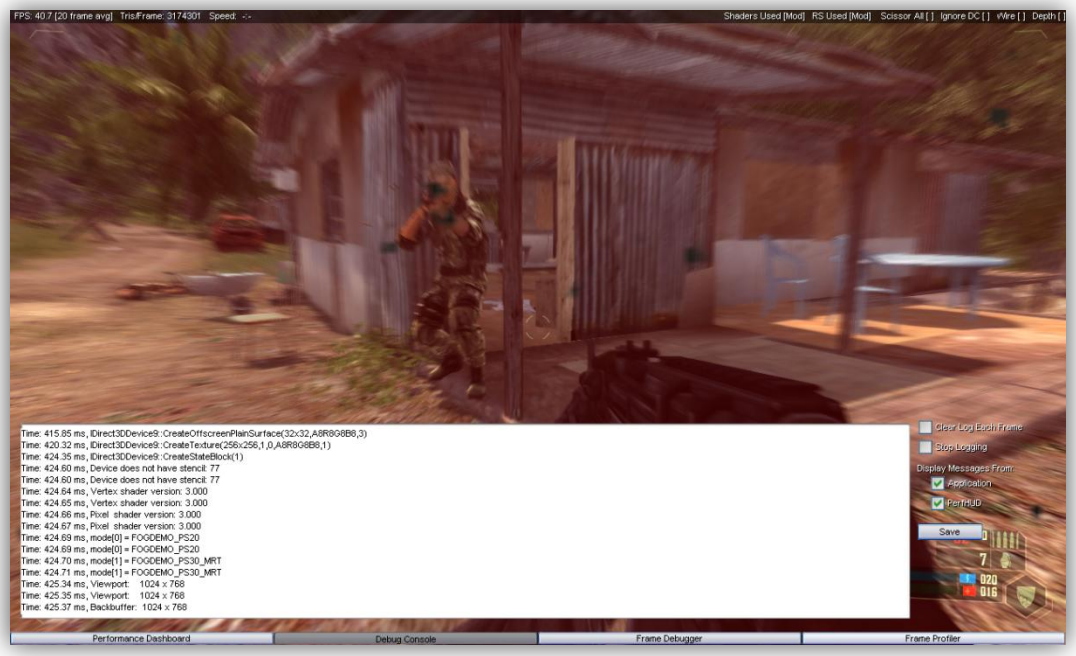
- If the **Driver Time** and **Frame Time** lines spike simultaneously it is likely because the driver is uploading a texture from the CPU to the GPU.
- If the **Frame Time** line spikes and the **Driver Time** line does not, your application is likely performing some CPU-intensive operation (like decoding audio) or accessing

the hard disk. This situation may also be caused by the operating system attending to other processes.

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

Chapter 3. 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.



Crysis used with permission from Crytek. © Crytek GmbH. All Rights Reserved. Crysis and CryENGINE are trademarks or registered trademarks of Crytek GmbH in the U.S and/or other countries.

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:

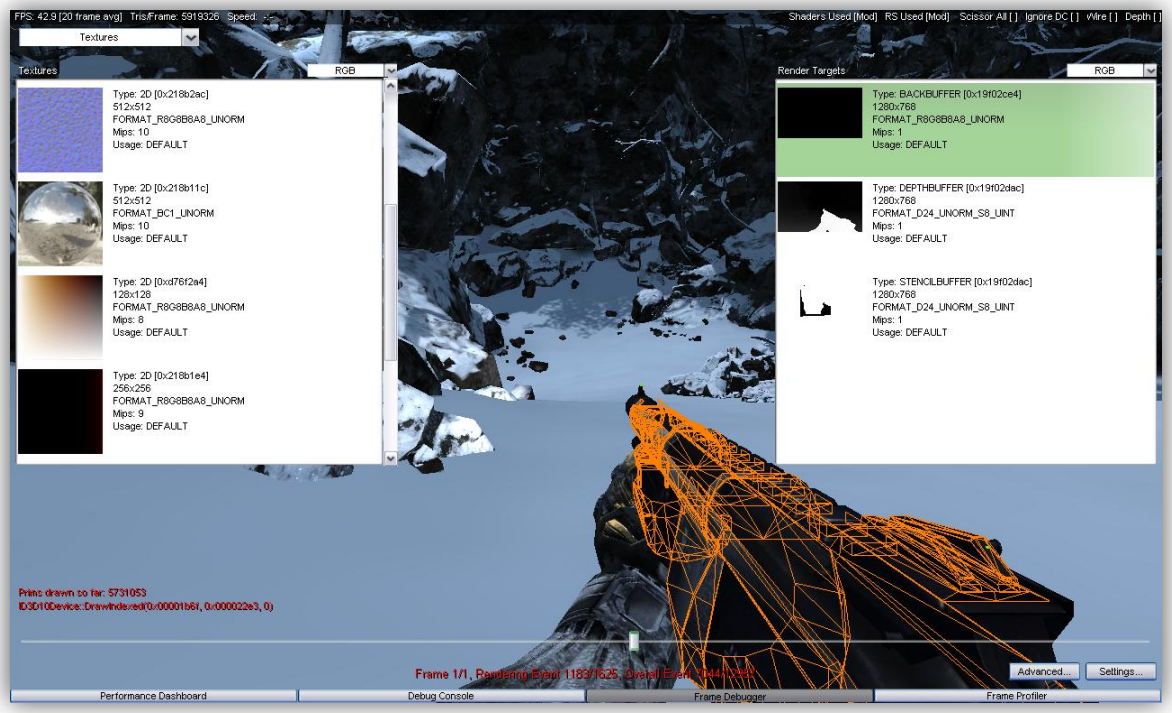
- **Clear Logging**: This option causes the contents of the console window to be cleared at the beginning of the frame so you only see the warnings generated by the current frame. This is useful when your application generates more warnings per frame than fit in the console window.
- **Stop Logging:** This option causes the console to stop displaying new messages.

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

Chapter 4. Frame Debugger

This chapter contains reference information on the PerfHUD 6 Frame Debugger and its advanced graphics pipeline Inspectors.

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.



Crysis used with permission from Crytek. © Crytek GmbH. All Rights Reserved. Crysis and CryENGINE are trademarks or registered trademarks of Crytek GmbH in the U.S and/or other countries.

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.

The Frame Debugger has two views: **Simple** and **Advanced**, which you can toggle between by clicking the **Advanced…** button in the lower right of the screen, or by typing **Ctrl+A**. Simple mode lets you look at high-level information for the current draw call, such as Textures and Render Targets, a complete API Call List for the frame, dependency information for the draw call, and Direct3D markers for the frame.

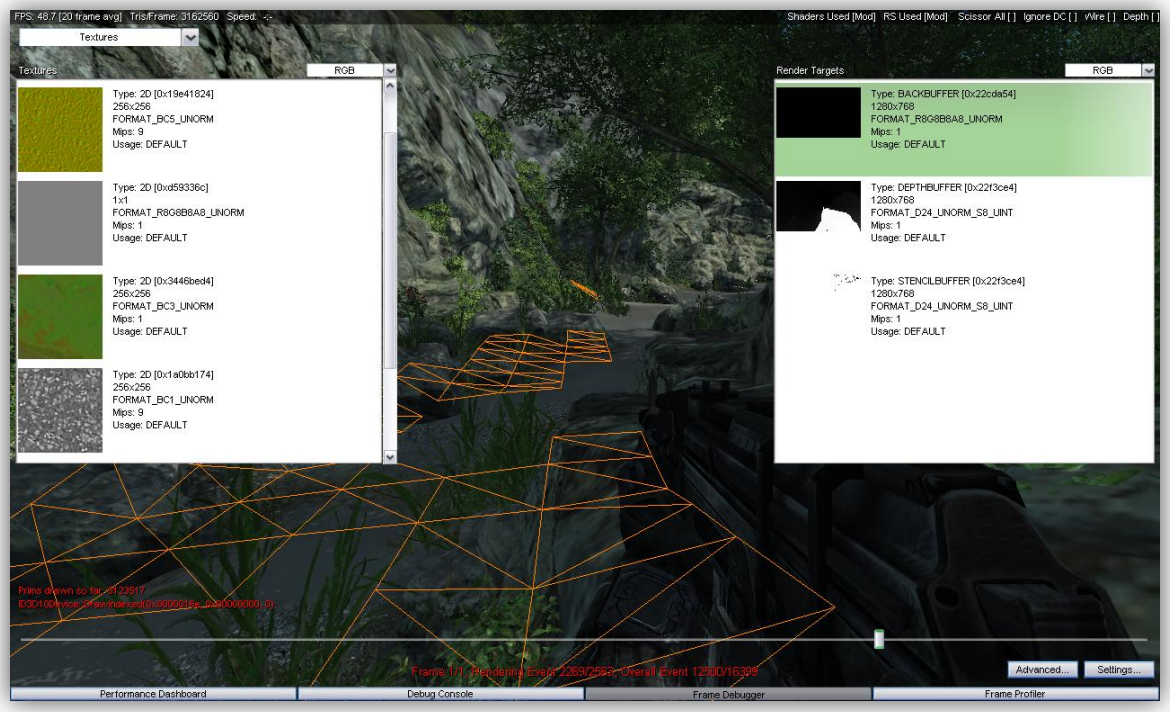
Advanced mode lets you view and edit data and shaders in each part of the rendering pipeline for the currently selected draw call.

Simple Mode

You can change the view in Simple Mode using the pull-down menu in the top-left corner of the screen, and switch between different views of your frame.

Texture View

The textures view shows every texture and render target associated with the current draw call. Textures are listed in the leftmost window, and render targets (if any) are listed in the rightmost window.



Crysis used with permission from Crytek. © Crytek GmbH. All Rights Reserved. Crysis and CryENGINE are trademarks or registered trademarks of Crytek GmbH in the U.S and/or other countries.

In PerfHUD 6, you can also replace a texture with an assortment of debug textures from this view.

Right-click a texture to bring up a context menu with all of the available options:

- **2x2 Texture**: Reduces texture bandwidth usage by using the smallest texture possible.
- **Black**, **25% Gray**, **50% Gray**, **75% Gray**, **White**, **Horizontal gradient**, **Vertical Gradient**: Each of these can be useful as debug input to your shaders.
- **Color Mipmap Texture**: Each mip level is given a different color so you can easily tell how well the texture is mipped at a glance. Mipmap level 0 (the highest-resolution mip level)

is shown in red, while the lowest-resolution mipmap level is shown in blue. Levels in between are shown in various other colors.

Mipmap visualization can help identify when a texture has insufficient resolution, which leads to on-screen blurriness, and can also identify scenarios where a texture has too much resolution – for example, when a 2048 x 2048 texture is assigned to a rock that is small on-screen.

- **Remove All Texture Overrides**: Restores the texture to its unmodified state.
- **Tonemap 0-1:** Useful for visualizing floating point textures. Maps values in the texture from their natural range down to between 0 and 1.
- **Save To File:** Save the current texture to anywhere in the file system.
- **Visualize:** Allows you to visualize the texture full-screen. Use the **left/right** arrow to switch between mip levels,the **up/down arrow** to visualize different faces/slices of a cubemap or 3d texture, the **numpad +/-** to zoom in and out, and **left-click and drag** to pan around the texture when it is larger than screen size. Hit **Escape** to return to the Frame Debugger.

API Call List View

The API Call List shows you every Direct3D API call made in the current frame. The list is colorized as follows:

- **Draw:** Blue
- **State/Effect:** Purple
- **Lock/Unlock:** Red
- **Perf Marker:** Green
- **Other:** Black

You can also **double click** on any call in the list, and the scrubber will jump to that part of the frame. This is an easy way to see which piece of geometry is associated with the call of interest.

Dependency View

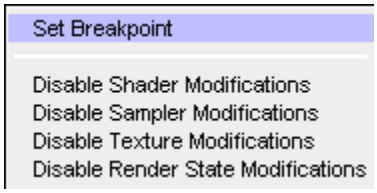
Dependencies between draw calls can have performance implications for both single GPU and SLI systems. The Dependencies view shows you producer/consumer relationships between draw calls both graphically on the frame scrubber, and in a listbox in the upper-right portion of the screen.

Left-clicking on a producer or consumer in the list view will move the frame scrub bar and the display to that draw call, so you can quickly understand the dependencies between objects in your scene.

D3D Perf Events View

The D3D Perf Events option allows you to see D3D Perf Events (as set by D3DPERF_BeginEvent and D3DPerf_EndEvent) that you have set in your application, displayed in a hierarchical tree view. **Left-clicking** on an event jumps to the associated draw call.

Right-clicking on a Perf Event brings up a context menu:



- **Set Breakpoint:** This option will set a CPU breakpoint at the selected Perf Event. After you set a breakpoint on a Perf Event, switch back to Performance Dashboard Mode (**F5**). Your application will break on the first draw call it encounters after the Perf Event is reached.
- **Disable Shader Modifications:** Disables any shader modifications you have made to the frame within this range of API calls.
- **Disable Sampler Modifications:** Disables any sampler modifications you have made to the frame within this range of API calls.
- **Disable Texture Modifications:** Disables any texture modifications you have made to the frame within this range of API calls.
- **Disable Render State Modifications:** Disables any render state modifications you have made to the frame within this range of API calls.

See the Direct3D API documentation for more information on using D3D Perf Events to instrument your application.

Advanced Mode

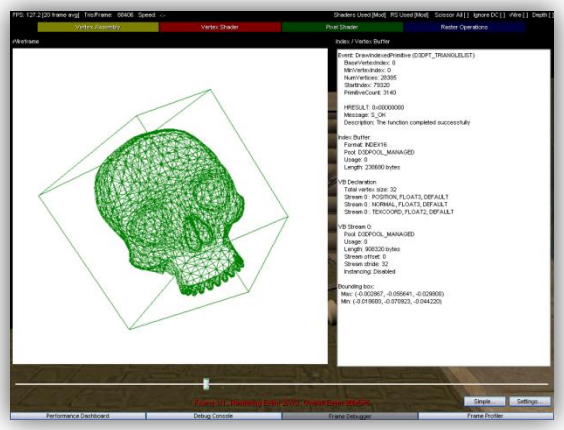
The Advanced view of the Frame Debugger allows you to inspect each pipeline stage in detail, including:

- Vertex Assembly — Fetches vertex data
- Vertex Shader — Executes vertex shaders
- Geometry Shader — Executes geometry shaders
- Pixel Shader — Executes pixel shaders
- Raster Operations — Post-shading operations in the frame buffer

You can find more detailed information about Direct3D structs shown in the Advanced State Inspectors in the documentation installed with your 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 Inspector



The Vertex Assembly Inspector shows the geometry that was submitted through the API for the selected draw call. It contains two windows: one for geometry and one for vertex metadata, which includes information such as:

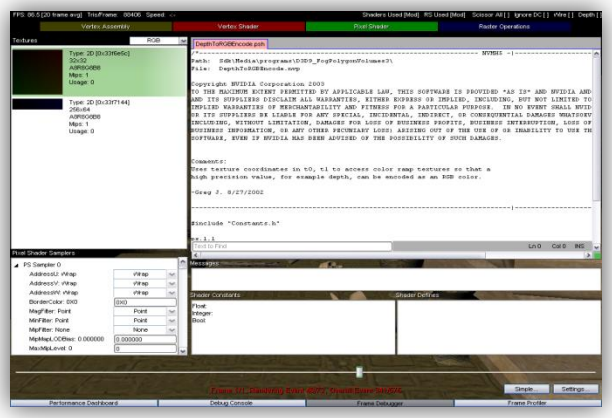
- Draw call parameters and return flags
- Index and Vertex buffer formats, sizes, etc.
- FVFs

You can rotate the geometry in the geometry window by **left-clicking and dragging.**

You can use 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.

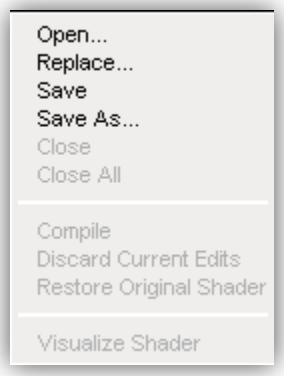
Vertex, Geometry and, Pixel Shader Inspectors



The shader inspectors each consist of 4 main areas.

- Source code editor
- Shader Constants viewer
- Texture viewer
- Sampler editor

The source code editor shows the source code for the vertex program (if any) of the current draw call. You can edit the source code for the current shader directly in the editor, search in the shader for text using the search box, and perform several actions by **right-clicking**, and selecting an action from the context menu**.**

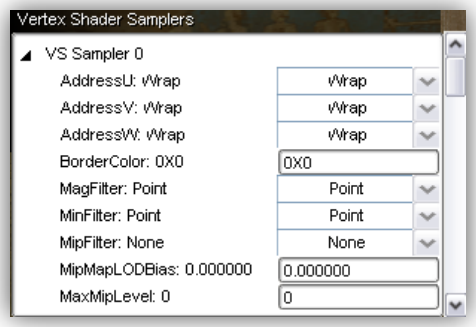


- **Open:** Opens a text file for display in a new tab.
- **Replace:** Replaces the current shader with the contents of the selected file.
- **Save:** Save the contents of the current tab to disk.
- **Save As:** Save the contents of the current tab to a new text file on disk.
- **Close:** Close the currently opened tab.
- **Close All:** Close all opened text files. (The shader source still stays open)
- **Compile:** Compile the current shader source. If there are compilation errors, they will be reported in the ***Messages*** pane just below the source code editor.
- **Discard Current Edits:** Go back to the last shader
- **Restore Original Shader**: Go back to the original shader that the application.
- **Visualize Shader:**

Note: Real-time shader editing currently has a limitation in that you cannot change a shader's language (e.g. from assembly to HLSL or vice versa). In addition, any HLSL edits you make must not change the technique, pass or function names.

The shader constant viewer is located under the source code editor and shows all constants and defines associated with this shader.

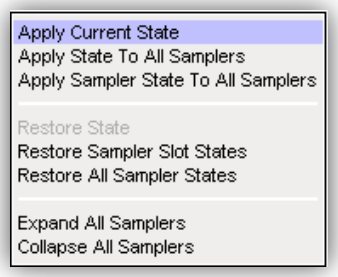
The texture viewer functions exactly like the texture viewer in the Frame Debugger's Simple mode. See that documentation for more information.



The sampler editor, new in PerfHUD 6, is located in the lower left of the screen underneat the texture viewer. You can choose to override any individual sampler input for any valid sampler. For more information on each sampler state, please refer to your DirectX SDK Documentation.

Also, note that by **right clicking** on the sampler editor, you can perform several sampler related actions:

- **Apply Current State:** Applies the current sampler state to the scene.
- **Apply State To All Samplers:** Replicates the selected sampler field to all other samplers.
- **Apply Sampler State to All Samplers:** Replicates the entire current sampler state to all other samplers.
- **Restore State:** Restores the current sampler property to its original state.
- **Restore Sampler Slot States:** Restores all sampler properties for the current sampler slot.
- **Restore All Sampler States:** Removes all sampler edits you have made since application startup.



- **Expand All Samplers**: Expands all samplers, and shows each sampler field in the sampler editor.
- **Collapse All Samplers:** Hides all sampler fields in the sampler editor.

Remember, sampler state edits override the sampler state for ALL draw calls in the frame.

In all shader inspectors, you should:

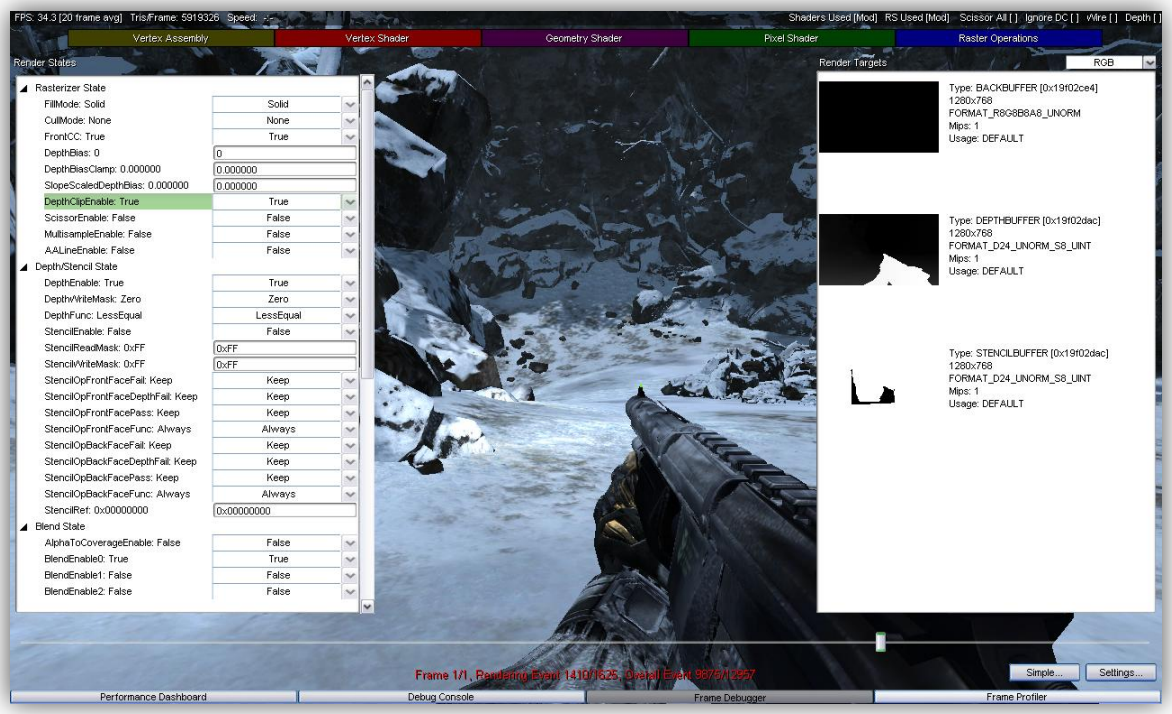
- **Verify** that the expected shader is applied for the current draw call**.**
- **Verify** that the constants are not passing `#NAN` or `#INF`**.**
- **Verify** that texture states are set correctly**.**

In the Pixel Shader Inspector, you should:

- **Verify** that the textures and render-to-texture textures are used correctly
- **Verify** that texture filtering states are set correctly

Raster Operations Inspector

The Raster Operations Inspector 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. The Raster Operations Inspector has two windows: one, on the left, for viewing and editing render state, and another, on the right, for viewing your render targets.



Crysis used with permission from Crytek. © Crytek GmbH. All Rights Reserved. Crysis and CryENGINE are trademarks or registered trademarks of Crytek GmbH in the U.S and/or other countries.

Every raster operations state for the current draw call is displayed for inspection and editing. There are several categories of raster properties which you can view and edit:

- **Depth State**
- **Stencil State**
- **Alpha Test State**
- **Rasterizer State**
- **Color Mask State**
- **Blend State**
- **Vertex and Primitive State**
- **Lighting and Material State**
- **Fog State**
- **Point and Sprite State**
- **Texture State**
- **Tessellation State**

- **Other State**

To learn more about the specific properties in each category, please consult the DirectX documentation installed with the DirectX SDK. Also, remember that editing raster operations state overrides that state for the entire frame, not just the current draw call. To perform an action on a particular state, **right-click** on the state, and select from one of the following options:

- **Apply Current State Globally:** Forces the clicked-upon state to the current value for the entire frame.
- **Restore State:** Restore the clicked-upon state back to its original value before any edits.
- **Restore Category States:** Restore the entire group
- **Restore All States:** Restore the entire ROP state to the original.
- **Expand All Categories:** Expand all categories so you can see all the properties per categories. Click on an expanded category to collapse it.
- **Collapse All Categories:** Collapse all entries so only the categories names are visible. Click on a category name to see its properties.

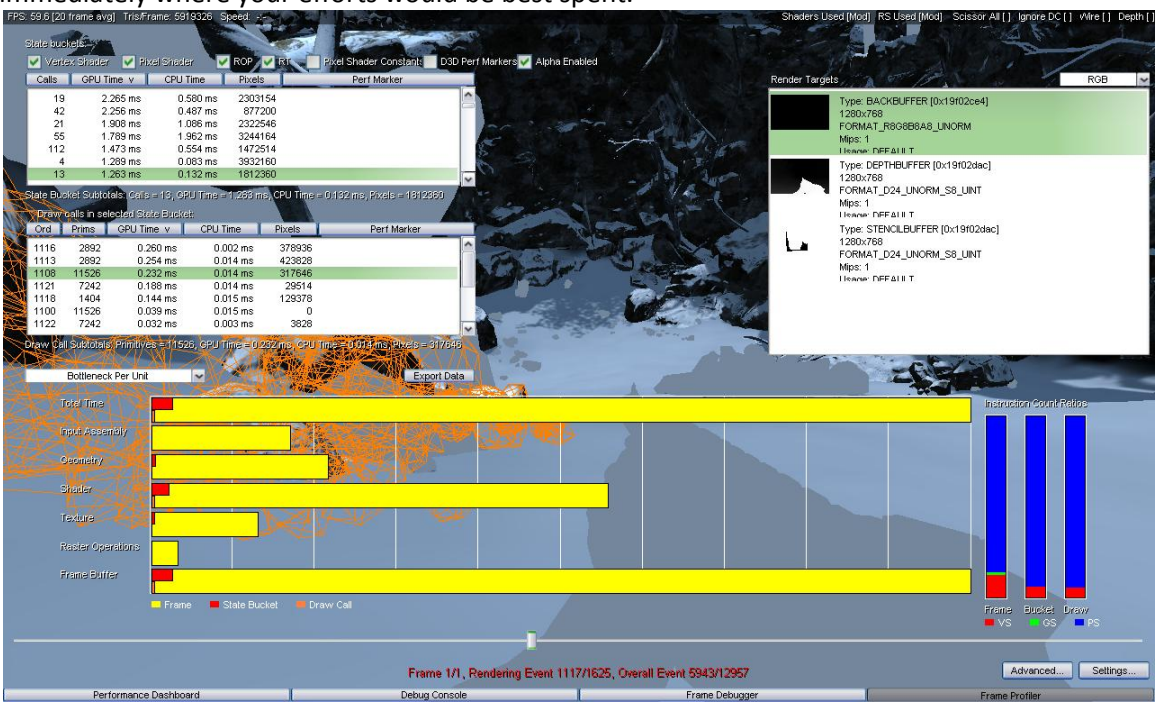
Also, certain render states can be very expensive:

- **Zenable**
- **Fillmode**
- **ZWriteEnable**
- **AlphaTestEnable**
- **SRCBLEND** and **DSTBLEND**
- **AlphablendEnable**
- **Fogenable**
- **Stencil enable**
- **StencilTest**

Chapter 5. Frame Profiler

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

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 immediately where your efforts would be best spent.



Crysis used with permission from Crytek. © Crytek GmbH. All Rights Reserved. Crysis and CryENGINE are trademarks or registered trademarks of Crytek GmbH in the U.S and/or other countries.

When you first switch to the Frame Profiler Mode, either by **left-clicking** the Frame Profiler button on the bottom of the screen or by typing **F8**, PerfHUD re-renders the current frame several times in a row, all the while collecting performance data from special hardware inside the GPU and instrumentation inside the drivers.

The Frame Profiler, just like the Frame Debugger, has two views: **Simple** and **Advanced**, which you can toggle between by clicking the **Advanced…** button (or the **Ctrl+A** keyboard shortcut).

In Simple Mode, you can access per-draw call graphs that pinpoint exactly where your GPU horsepower is being spent. In Advanced Mode, you can do perform all the operations available

in the Frame Debugger's Advanced Mode, all while viewing unit utilization bar charts for the current draw call.

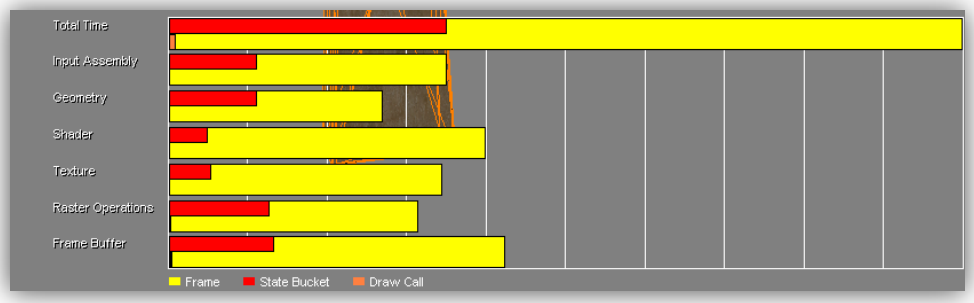
Simple Mode

The most important feature of the Frame Profiler's Simple mode is variety of graphs that you can select, each of which show you performance information on a **per draw call basis.**

Each frame profiler graph exposes a different aspect of graphical performance, and each graph is linked to the frame selection bar (scrubber) at the bottom of the screen. Using the scrubber you can quickly locate and identify expensive draw calls, and even understand which specific parts of the draw call are troublesome.

In timing and utilization graphs, where the x-axis corresponds to draw call number, you can **left-click** anywhere on a graph to immediately jump the scrubber and the display to that draw call. Both bar type graphs and timing graphs will modify

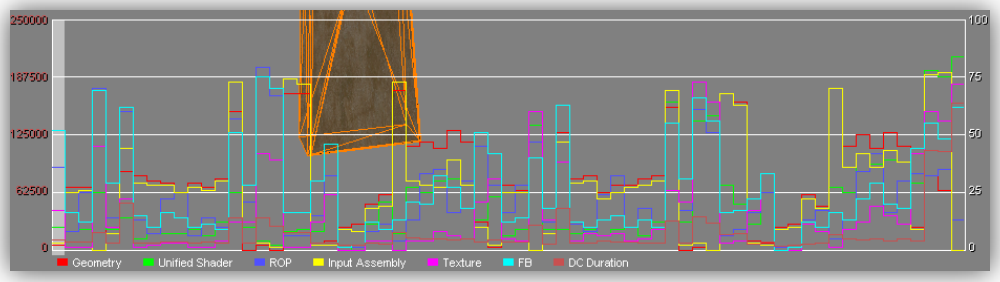
1. Bottleneck Per Unit



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.

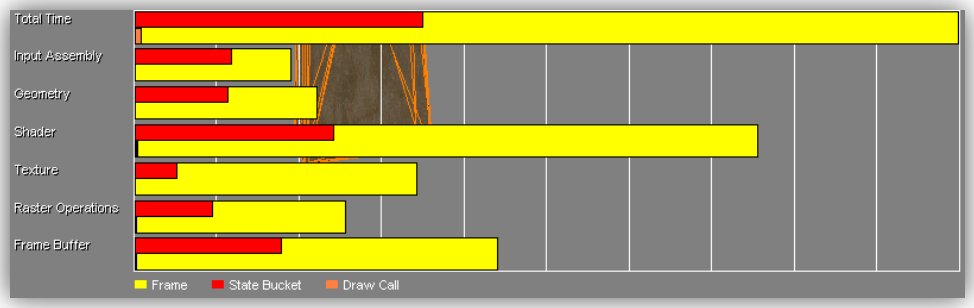
2. Bottleneck Percentage

This graph shows % utilization vs. the draw call. You can quickly see which draw calls are being bottlenecked, and which unit is the issue.



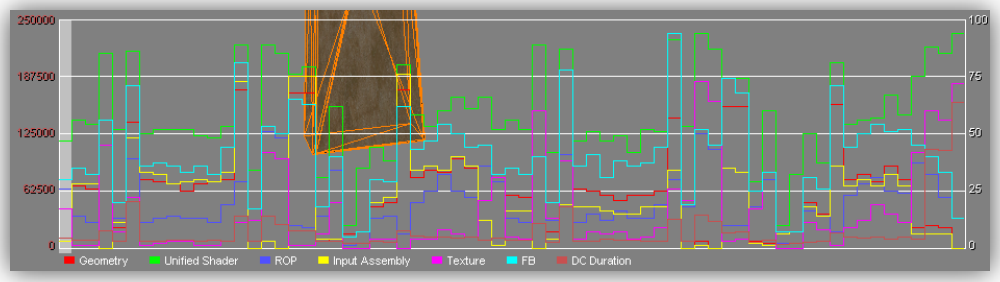
3. Utilization Per Unit

This graph shows GPU unit utilization in milliseconds for each draw call (and its associated state bucket) in the current frame.



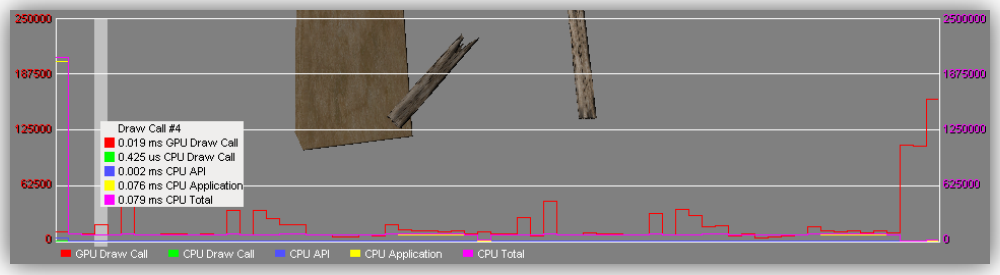
4. Utilization Percentage

This graph shows the percentage utilization per GPU unit. You can use this graph to spot bottlenecks by seeing which units are utilized at 100%, and at which draw call.



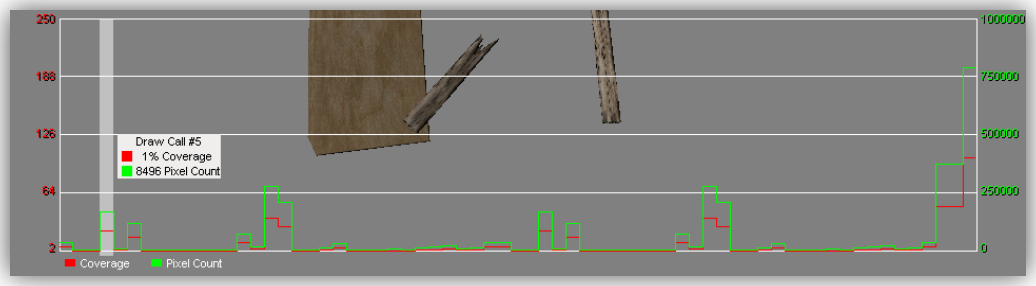
5. GPU and CPU Timings

See GPU draw call, CPU draw call, CPU API, CPU Application, and CPU Total times, all graphed per draw call.



6. Shaded Pixels

The Shaded Pixels graph shows the total number of pixels shaded by the hardware per draw call.

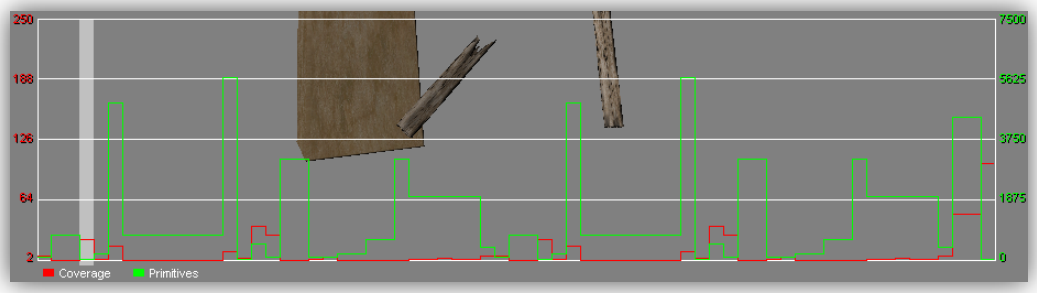


7. Texture LOD

The Texture LOD graph shows you the percentage of texture samples that come from the base level of your mipped texture, and your average LOD level across all textures.

8. Primitives

The primitives graph plots screen coverage and primitive numbers per draw call.



State Buckets

When you switch to Frame Profiler Mode, 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:

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

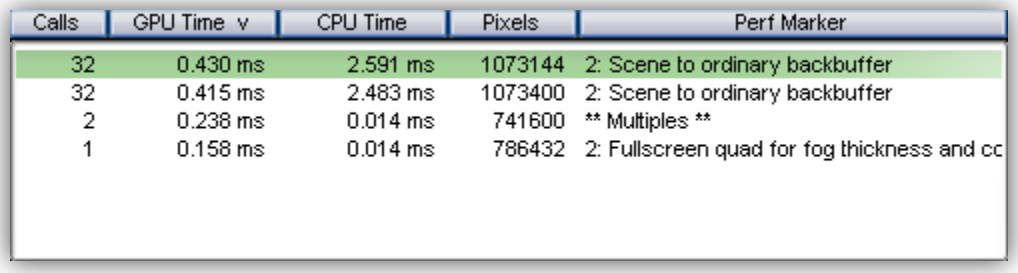
The object close to the camera will probably take more time to draw. Optimizing for the bottleneck of the most expensive draw call (A) in this state bucket will also benefit the other draw call (B) at times when the second object is close to the camera.

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 for more information.

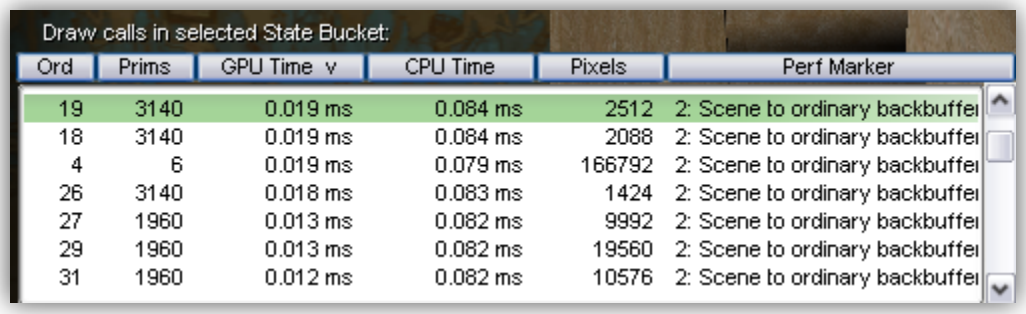
Initially, you should let PerfHUD use the default state bucket configuration, shown below:

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.

Advanced Mode

The Frame Profiler's advanced mode is very similar to the Frame Debugger's Advanced Mode.

However, in the Frame Profiler's Advanced Mode, two additional bars are shown at the top of your screen, which allow you to 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 graph 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 6. 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.
- PerfHUD may crash when using software vertex processing.
- The State Inspectors do not display detailed information when your application is using fixed T&L.
- Applications running in windowed mode may not exit properly when PerfHUD is active and you click on the close button.
- When you enter the Frame Debugger or Frame Profiler, PerfHUD only freezes the graphics threads, so time may still be incrementing in your game threads. If this is a problem, press **Enter** to freeze the game while in the Performance Dashboard, and then switch to the Frame Debugger/Profiler.
- If you see an error message stating “Can’t allocate # objects”, then PerfHUD has failed on a memory allocation. Running your application on a computer with more memory may help. If you are already up against the 32-bit memory limit, you can also try running the command **editbin /LARGEADDRESSAWARE myGame.exe** in order to allow your application to access 4 GB instead of 2 GB of memory.

Chapter 7. 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.

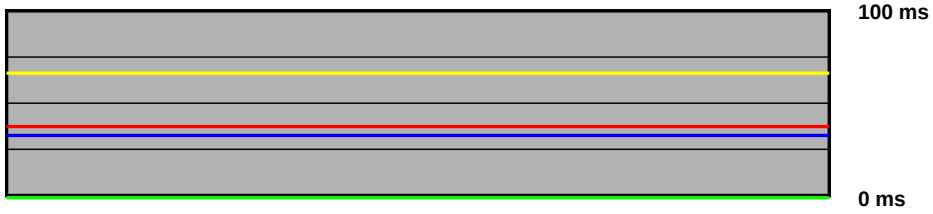


Figure 1. Driver Waiting for the GPU

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.

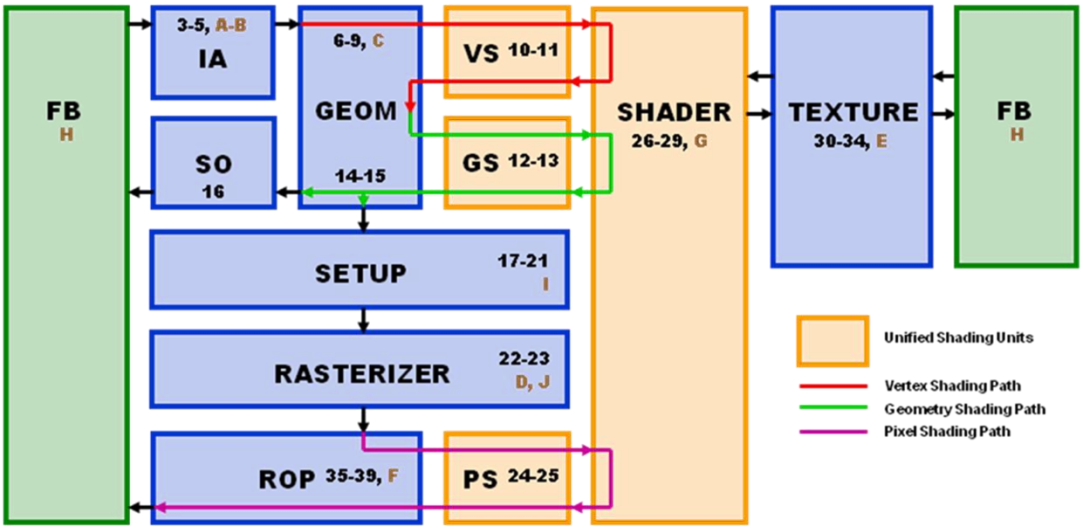
Chapter 8. Appendix B. The NVIDIA Software Improvement Program

The first time you run PerfHUD, you'll be prompted to join the NVIDIA Software Improvement Program (or SIP), which you can learn more about at http://developer.nvidia.com/object/SIP.html).

If you opt-in to the SIP, PerfHUD will record which product features you're using, and we will use this information to make the product better in future versions of the product. **At no time is content of any kind (such as models/textures/shaders/scripts) sent to NVIDIA**. We encourage you to opt-in, as you will be helping to guide future software improvements towards your usage scenarios.

The SIP also allows you to immediately send feedback to NVIDIA at any time by pressing F4. This will bring up an Instant Feedback dialog box where you can enter suggestions or bug reports.

Chapter 9. Appendix C. Signals Reference



1. gpu_idle (Global)
2. gpu_busy (Global)
3. input_assembler_busy
4. input_assembler_waits_for_fb
5. vertex_attribute_count
6. geom_busy
7. geom_waits_for_shader
8. geom_vertex_in_count
9. geom_primitive_in_count
10. vertex_shader_busy
11. vertex_shader_instruction_rate
12. geometry_shader_busy
13. geometry_shader_instruction_rate
14. geom_vertex_out_count
15. geom_primitive_out_count
16. stream_out_busy
17. setup_primitive_culled_count
18. setup_primitive_count
19. setup_triangle_count
20. setup_point_count
21. setup_line_count
22. shaded_pixel_count
23. rasterizer_pixels_killed_zcull_count
24. pixel_shader_busy
25. pixel_shader_instruction_rate
26. shader_busy
27. shader_waits_for_texture
28. shader_waits_for_geom
29. shader_waits_for_rop
30. texture_busy
31. texture_waits_for_fb
32. texture_waits_for_shader
33. texture_sample_base_level_rate
34. texture_sample_average_level
35. rop_busy
36. rop_waits_for_fb
37. rop_waits_for_shader
38. rop_pixels_killed_earlyz_count
39. rop_pixels_killed_latez_count

A. 2D Bottleneck and SOL
B. IDX Bottleneck and SOL
C. GEOM Bottleneck and SOL
D. ZCULL Bottleneck and SOL
E. TEX Bottleneck and SOL
F. ROP Bottleneck and SOL
G. SHD Bottleneck and SOL
H. FB Bottleneck and SOL
I. Primitive Setup Bottleneck and SOL
J. Rasterization Bottleneck and SOL

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