NVPerfHUD 4 | Quick Reference Guide

Overview

Analyze your application like an NVIDIA engineer.

NVPerfHUD is a powerful performance analysis tool that gives you unparalleled insight into how your application uses the GPU. It allows you to analyze your application from a global view to individual draw calls, providing numerous graphics pipeline experiments, graphs of performance metrics, and interactive visualization modes. NVPerfHUD has four modes:

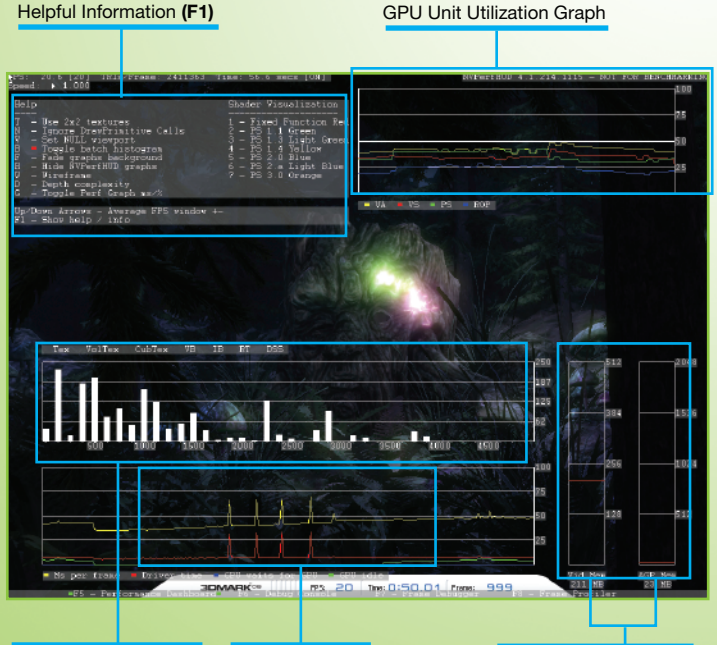
- **F5** **Performance Dashboard:** Identify high-level bottlenecks with graphs and directed experiments.
- **F6** **Debug Console:** Review DirectX debug runtime messages, NVPerfHUD warnings, and custom messages.
- **F7** **Frame Debugger:** Freeze the current frame and step through your scene one draw call at a time, dissecting them with advanced State Inspectors.
- **F8** **Frame Profiler:** Automatically identify your most expensive rendering states and draw calls and improve performance with detailed GPU instrumentation data.

Performance Dashboard (F5)

A global view of your application's GPU performance.

Configure the information displayed on the screen and perform several graphics pipeline experiments:

- **F** Cycle Helpful Information
- **B** Show Batch Size Histogram
- **F** Fade Background
- **T** Isolate the texture unit by forcing 2x2 textures
- **V** Isolate the vertex unit by using a 1x1 scissor rectangle to clip rasterization and shading work
- **N** Eliminate the GPU (and state change overhead) by ignoring all draw calls
- **H** Hide Graphs
- **W** Show Wireframe
- **D** Show Depth Complexity

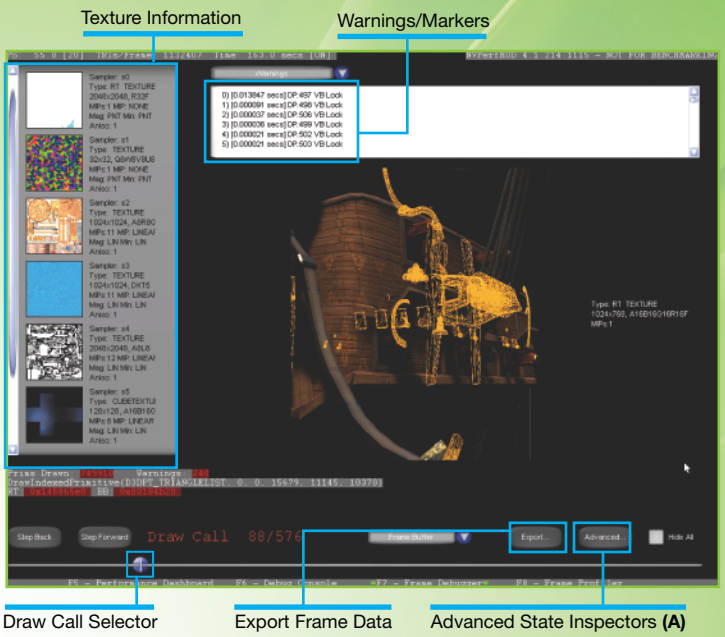


Frame Debugger (F7)

Step through a frame, one draw call at a time.

Use the slider or the left/right arrow keys and the options below to scrub through your scene:

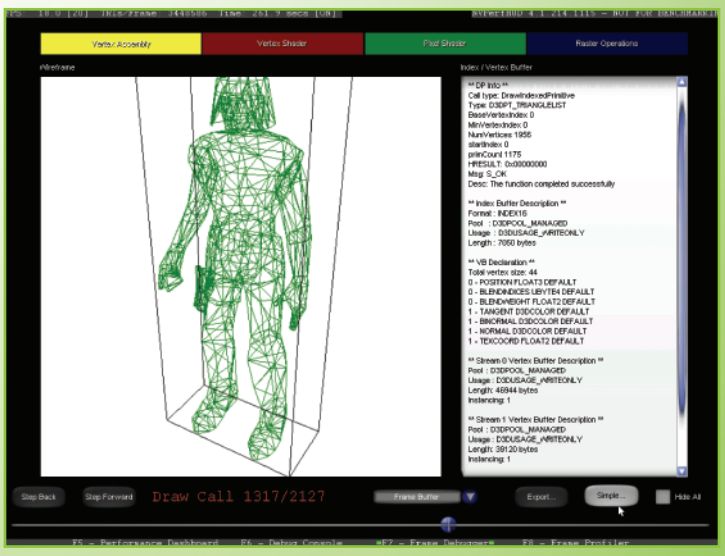
- **A** Toggle Simple/Advanced
- **J** Jump to Warnings
- **W** Show Wireframe
- **D** Show Depth Complexity



State Inspectors

Click on the **Advanced**... button to use the Advanced State Inspectors. You can click on the colored bar or use the shortcut keys below to switch between State Inspectors:

1. **Vertex Assembly:** examines vertex data
2. **Vertex Shader:** examines vertex shaders and textures
3. **Pixel Shader:** examines pixel shaders and textures
4. **Raster Operations:** examines post-shading operations



NVPerfHUD 4 | Quick Reference Guide

Frame Profiler (F8)

Automatically find expensive draw calls and render states.

Group draw calls into “state buckets” based on pixel shader, vertex shader, ROP, render target, and pixel shader constants. Click on column headings to sort state buckets and draw calls.

State Bucket Grouping Criteria | State Buckets | Draw Calls

Graph Legend

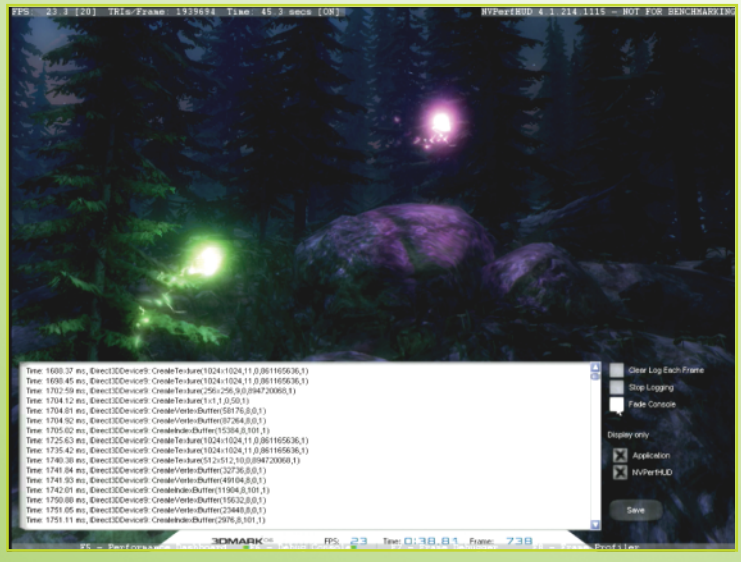
Total frame time | Cost of selected state bucket | Cost of selected draw call

Other available visualizations:

- Draw call duration
- Unit utilization graphs
- Shaded pixel count
- Per-unit bottlenecks
- Double-speed Z/Stencil

Debug Console (F6)

See debug runtime errors, warnings, and custom messages.



All images used with permission from Futuremark Corporation.

Optimizing with NVPerfHUD

1. Navigate your application to the area you want to analyze.
2. If you notice any rendering issues, use the Debug Console and Frame Debugger to solve those problems first.
3. Check the Debug Console for performance warnings.
4. When you notice a performance issue, switch to the Frame Profiler (if you have an NVIDIA GeForce 6 Series or later GPU) and use the advanced profiling features to identify the bottleneck. Otherwise, use the pipeline experiments in the Performance Dashboard to identify the bottleneck.

Methodology

1. Identify the bottleneck.
2. Optimize the bottleneck stage.
3. Repeat steps 1 and 2 until desired performance level is achieved.

Suggested CPU Optimizations

- **Reduce Resource Locking**
  - Avoid locking/reading from a surface you were previously rendering to.
  - Avoid writing to a surface that the GPU is reading from, like a texture or a vertex buffer.
- **Minimize Number of Draw Calls**
  - Use degenerate triangles to stitch together disjoint triangle strips.
  - Use texture pages.
  - Use the vertex shader constant memory as a lookup table of matrices.
  - Use geometry instancing if you have multiple copies of the same mesh in your scene.
- **Reduce Cost of Vertex Transfers**
  - Use the smallest possible vertex format.
  - Generate potentially derivable vertex attributes inside the vertex program instead of storing them inside of the input vertex format.
  - Use 16-bit indices instead of 32-bit indices.
  - Access vertex data in a relatively sequential manner.
- **Allocate Resources Efficiently**
  - Create all vertex buffers, index buffers, and textures ahead of time, and reuse them with DISCARD.

Suggested GPU Optimizations

- **Optimize Vertex Processing**
  - Pull out per-object computations onto the CPU.
  - Use vertex processing LOD.
  - Use correct coordinate space.
  - Use vertex branching to early-out of computations.
- **Speed up Pixel Shading**
  - Render depth first.
  - Help early-Z optimizations throw away pixel processing.
  - Store complex functions in textures.
  - Move per-pixel work to the vertex shader.
  - Use the lowest precision necessary.
  - Avoid unnecessary normalization.
  - Consider using pixel shader level-of-detail.
- **Reduce Texture Bandwidth**
  - Reduce the size of your textures.
  - Use mipmapping on any surface that may be minified.
  - Compress all color textures.
  - Avoid floating-point texture formats if not necessary.
- **Optimize Frame Buffer Bandwidth**
  - Render depth first.
  - Reduce alpha blending.
  - Turn off depth writes when possible.
  - Avoid extraneous color buffer clears.
  - Render roughly front to back.
  - Only use floating point frame buffers when necessary.