NVPerfHUD 3
Quick Reference

When NVPerfHUD is activated, you can perform graphics pipeline experiments, display graphs of performance metrics, and explore potential problems using several performance visualization modes. You can also switch to the Debug Console or Frame Analysis Mode for deeper analysis. Use these shortcut keys to switch modes:

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

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

F7 **Frame Analysis Mode** - Freeze the current frame and step through your scene one draw call at a time, using advanced State Inspectors for state of the graphics pipeline.

Performance Analysis Mode

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

F1 Cycle display of helpful information

B Show Batch Size Histogram

F Fade Background

H Hide Graphs

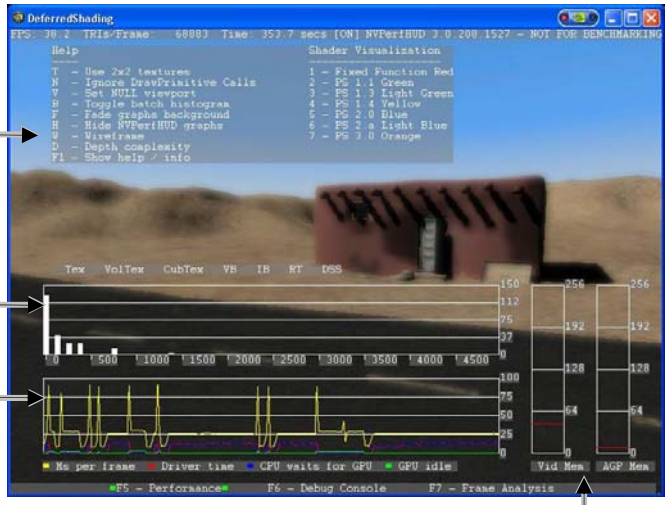
W Show Wireframe

D Show Depth Complexity

T Isolate the texture unit by forcing the GPU to use 2x2 textures

V Nullify workload of units after the vertex shader using a 1x1 scissor rectangle to clip all rasterization and shading work

N Eliminate the GPU (and state change overhead) by ignoring all draw calls



Frame Analysis Mode

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

A Toggle Simple/Advanced display

S Show Warnings

W Show Wireframe

D Show Depth Complexity

State Inspectors

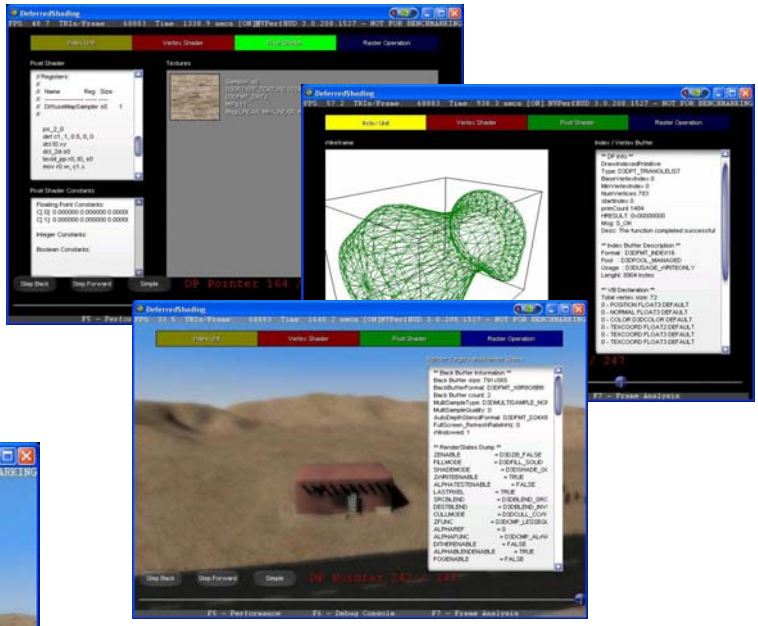
Click on the Advanced… button to use the advanced State Inspectors. You can click on the colored bar or use the shortcut keys below to switch between State Inspectors:

1 **Index Unit –** fetches vertex data

2 **Vertex Shader –** executes vertex shaders

3 **Pixel Shader –** executes pixel shaders

4 **Raster Operations –** post-shading operations



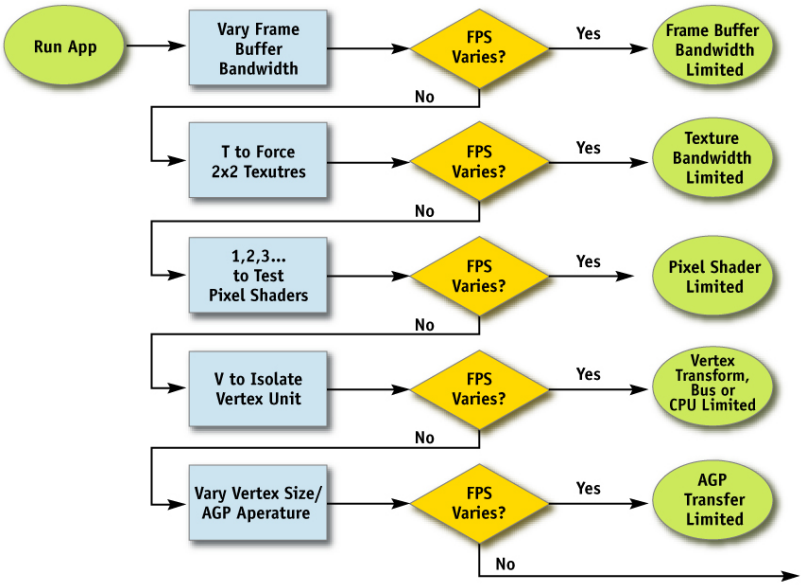
Using the **Index Unit State Inspector** you can verify all the information used to fetch the vertex data and make sure the geometry associated with the current draw call is correct.

Use the **Vertex** and **Pixel Shader State Inspectors** to verify that the shader program constants and textures are correct for the current draw call. Make sure the constants are not **#NAN** or **#INF**.

Using the **Raster Operations State Inspector** you confirm that the back buffer format has an alpha component when blending doesn't seem to be working properly, verify that opaque objects are not drawn with **blendEnable**, etc.

NVPerfHUD 3 Quick Reference

Identifying Bottlenecks



Note: If you suspect that you are CPU limited, press **N** at any time. If the frame rate of your application does not change, you are CPU limited.

GPU Optimizations

- **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
  - Use half precision normalizes when possible (e.g. `nrm_pp`)
  - Consider using pixel shader level-of-detail
  - Disable trilinear filtering when unnecessary
- **Reduce Texture Bandwidth**
  - Reduce the size of your textures
  - Always use mipmapping on any surface that may be minified
  - Compress all color textures
  - Avoid expensive texture formats if not necessary
- **Optimize Framebuffer Bandwidth**
  - Render depth first
  - Reduce alpha blending
  - Turn off depth writes when possible
  - Avoid extraneous color buffer clears
  - Render front-to-back clears
  - Optimize skybox rendering
  - Only use floating point framebuffers when necessary
  - Use a 16-bit depth buffer when possible
  - Use a 16-bit color when possible

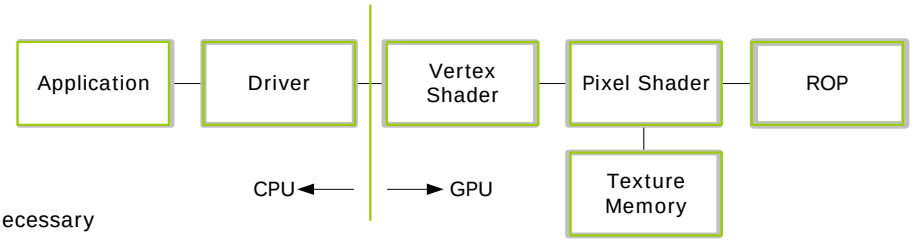
Methodology

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

CPU Optimizations

- **Reduce Resource Locking**
  - Avoid lock or read from a surface you were previously rendering to
  - Avoid write to a surface the GPU is reading from, like a texture or a vertex buffer
- **Minimize Number of Draw Calls**
  - If using triangle strips, use degenerate triangles to stitch together disjoint 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
  - Use CPU shader branching to increase batch size
  - Defer decisions as far down in the pipeline as possible
- **Reduce Cost of Vertex Transfers**
  - Use the fewest number of bytes possible in 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
- **Optimize Vertex Processing**
  - Pull out per-object computations onto the CPU
  - Optimize for the post-TnL vertex cache
  - Reduce the number of vertices processed
  - Use vertex processing LOD
  - Use correct coordinate space
  - Use vertex branching to *early-out* of computations

Graphics Pipeline



Refer to the *NVPerfHUD User Guide* for optimization details

developer.nvidia.com
The Source for GPU Programming

NVIDIA, the NVIDIA logo, NVPerfHUD are trademarks or registered trademarks of NVIDIA Corporation.
Other company and product names may be trademarks of the respective companies with which they are associated.
© 2005 by NVIDIA Corporation. All rights reserved