NVIDIA®

Performance Tools

Jeff Kiel, NVIDIA Corporation
Sim Dietrich, Composite Studios

Performance Tools Agenda

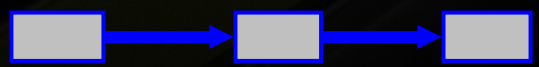
- Problem statement
- GPU pipelined architecture at a glance
- NVPerfKit 2.0: Driver and GPU Performance Data
  - GLExpert: OpenGL API Assistance
  - NVPerfHUD: The GPU Performance Accelerator
  - NVPerfSDK: Integrated into your application
    - NVPerfAPI
    - PDH, NVDevCPL
  - NVIDIA plug-In for Microsoft PIX for Windows
- Solutions to common bottlenecks
- NVShaderPerf: Shader Performance

What's The Problem?

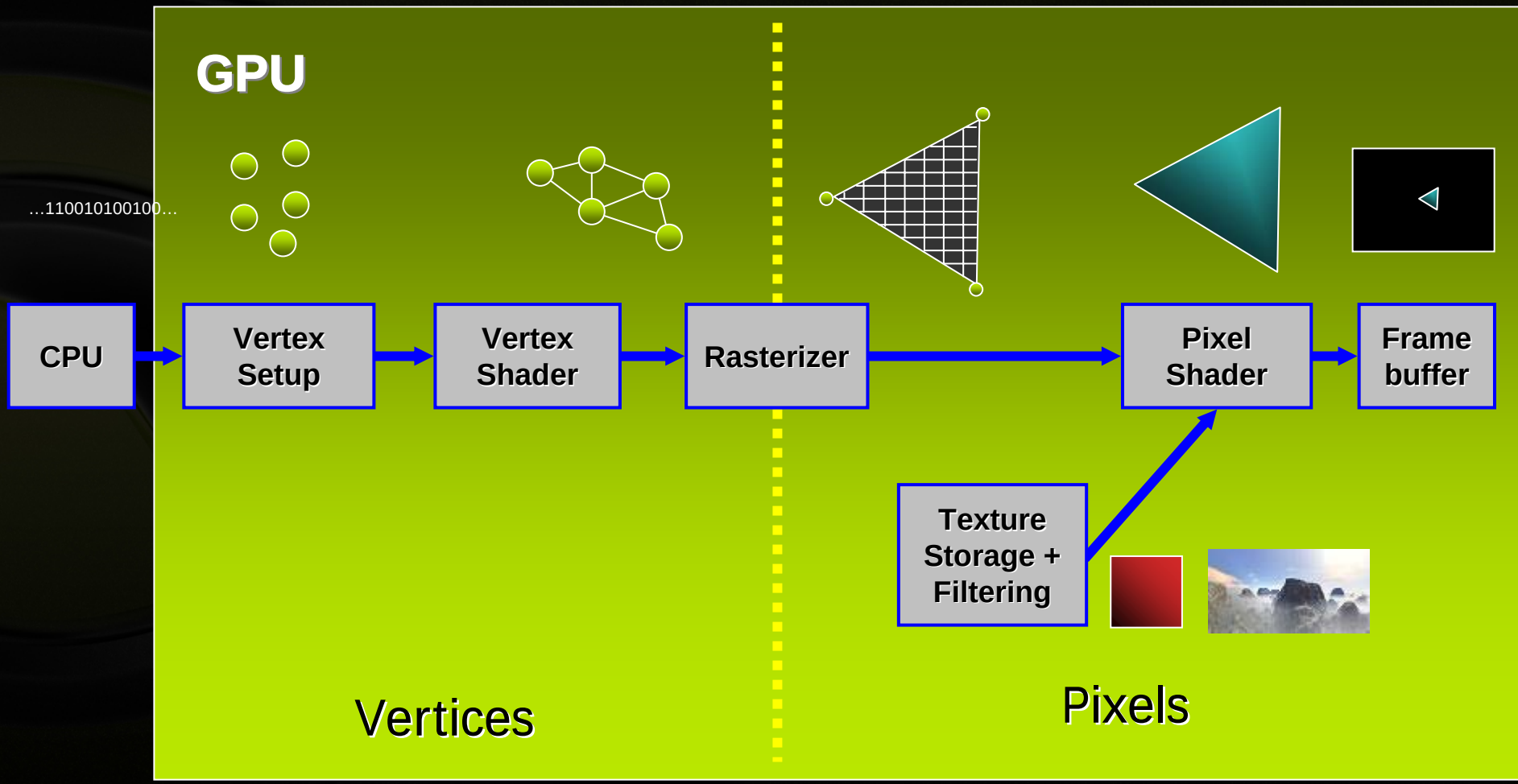
- Why is my app running at 13FPS after CPU tuning?
- How can I determine what is going on in that GPU?
- How come IHV engineers are able to figure it out?

GPU architecture at a glance

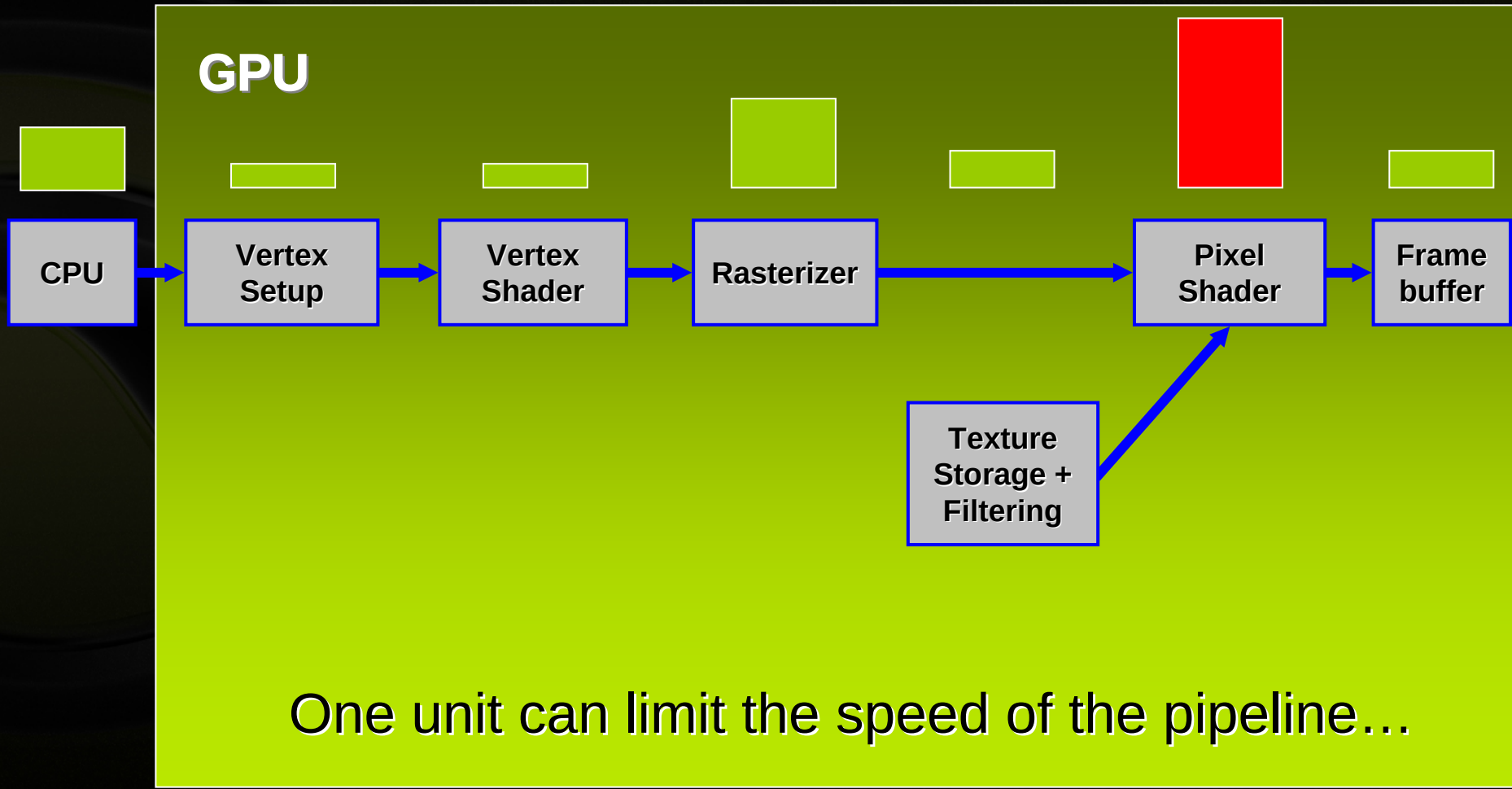
- Pipelined architecture: each unit needs the data from the previous unit to do its job
- Method: Bottleneck identification and elimination
- Goal: Balance the pipeline



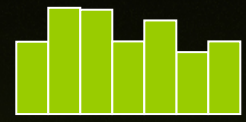
GPU Pipelined Architecture (simplified view)



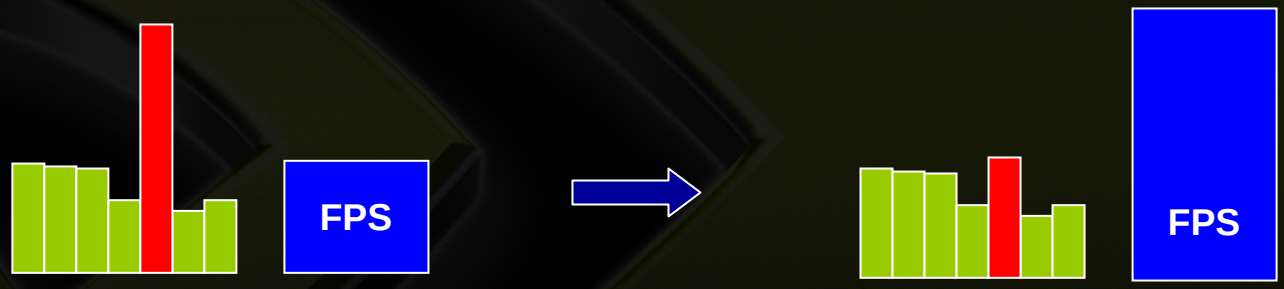
GPU Pipelined Architecture (simplified view)



Classic Bottleneck Identification

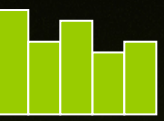


Modify target stage to decrease workload



If performance/FPS improves greatly, this stage is the bottleneck
Careful not to change the workload of other stages!

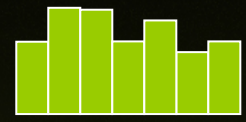
Classic Bottleneck Identification



Rule out other stages, give them little or no work

If performance doesn’t change significantly, this stage is the bottleneck
Careful not to change the workload of target stage!

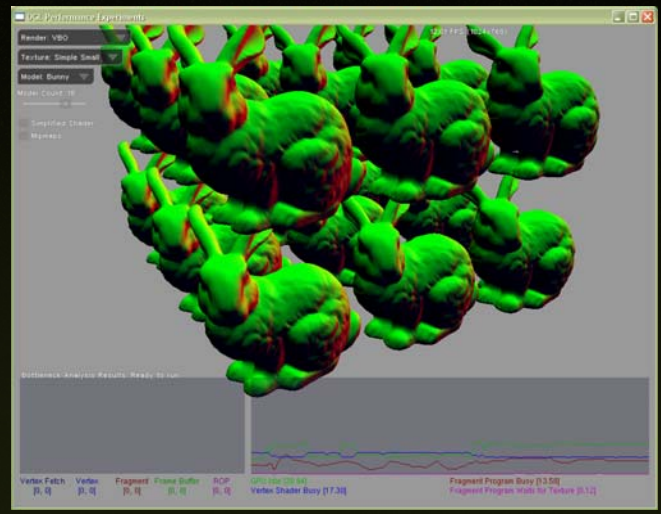
Ideal Bottleneck Identification



- Sample performance data at different points along the pipeline while rendering
  - Compare amount of work done to maximum work possible
  - Query the GPU for unit bottleneck information
- The answer? NVPerfKit!
  - NVPerfHUD: The GPU Performance Accelerator
  - NVPerfAPI: Integrated in your application

Analyze your application like an NVIDIA Engineer!

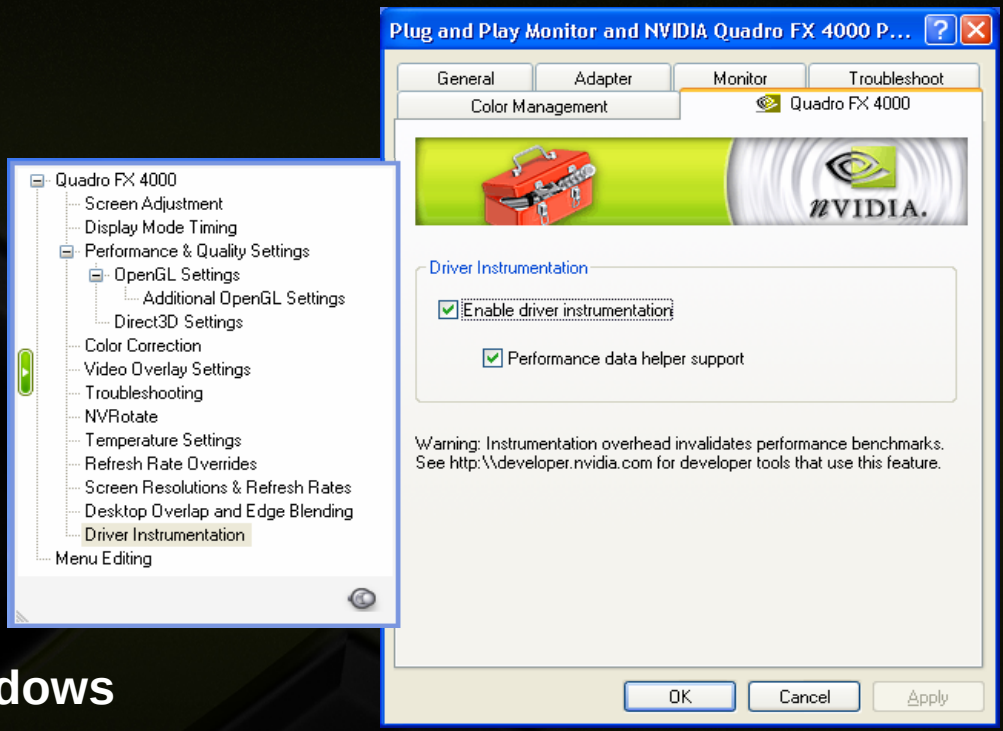
NVPerfKit



- NVPerfKit 2.0
  - The package
  - Software/driver counters
  - GPU counters
  - Simplified Experiments
- NVPerfHUD demo with Sim Dietrich
- Integrating NVPerfKit with NVPerfAPI
- Associated tools

What is in the NVPerfKit package?

- Instrumented Driver
- GLExpert
- NVPerfHUD
- NVPerfSDK
  - NVPerfAPI
  - Sample Code
  - Helper Classes
  - Documentation
- Tools
  - NVIDIA Plug-In for Microsoft PIX for Windows
  - gDEBugger
  - NVDevCPL



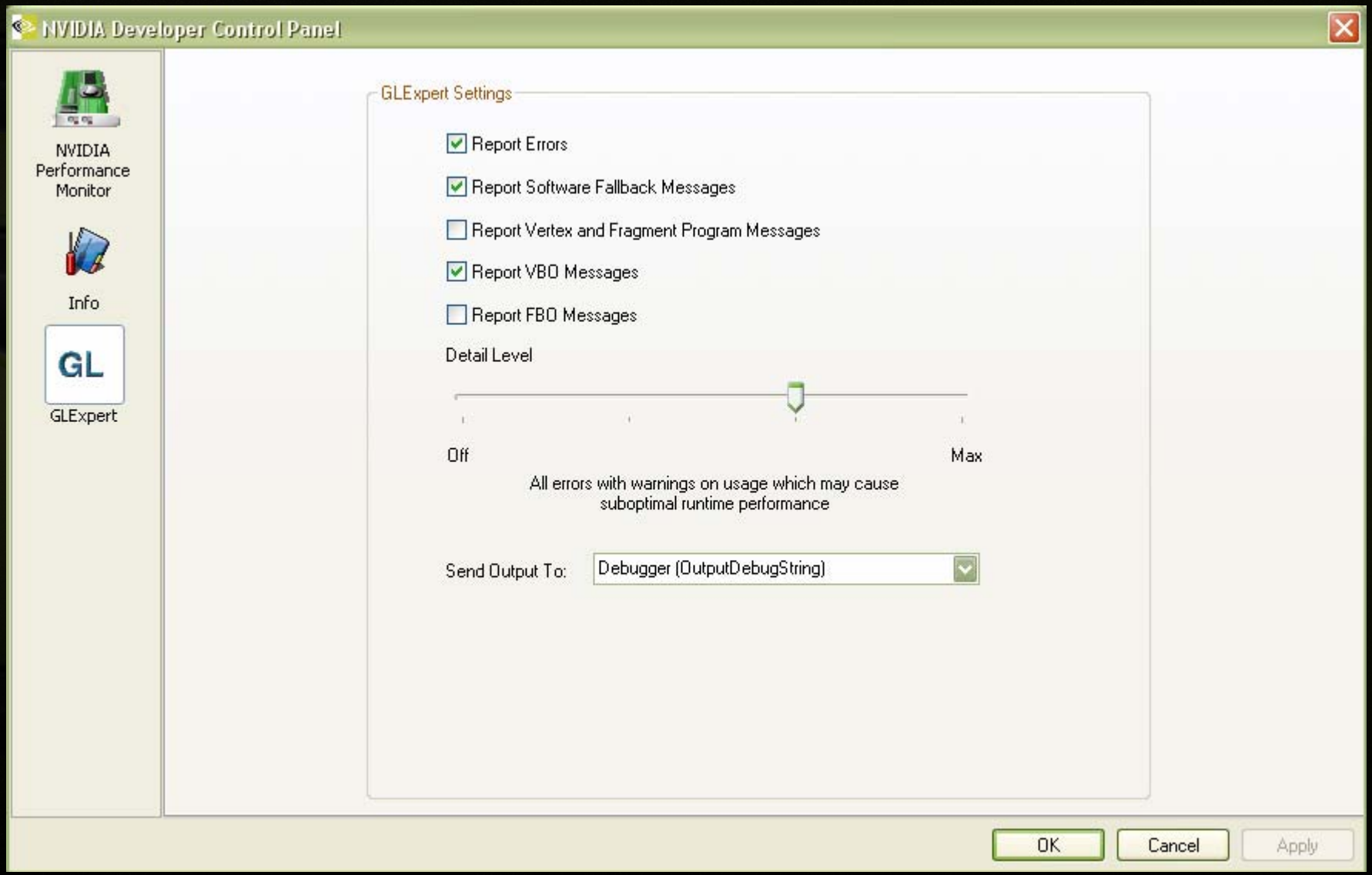
NVPerfKit Instrumented Driver

- Exposes GPU and Driver Performance Counters
- Data exported via NVIDIA API and PDH
- Supports OpenGL and Direct3D
- Simplified Experiments (SimExp)
- Collect GPU and driver data, retain performance
  - Track per-frame statistics
  - Gather and collate at end of frame
  - Performance hit 1-2%

GLExpert: What is it?

- Helps eliminate performance issues on the CPU
- OpenGL portion of the Instrumented Driver
  - Output information to console/stdout or debugger
  - Different groups and levels of information detail
- Controlled using tab in NVDevCPL
- What it can do (today)
  - GL Errors: print when they are raised
  - Software Fallbacks: indicate when the driver is in fall back
  - GPU Programs: errors during compile or link
  - VBOs: show where they reside, mapping details
  - FBOs: print reasons a configuration is unsupported
- Feature list to grow with future drivers

GLExpert: NVDevCPL tab



Project Status

- Shipping with NVPerfKit 2.0
- Windows for now, Linux to follow
- Supports NV3x, NV4x, and G7x architectures
- Integrated into Graphic Remedy‘s gDEBugger
- What types of things are interesting?

NVPerfKit@nvidia.com

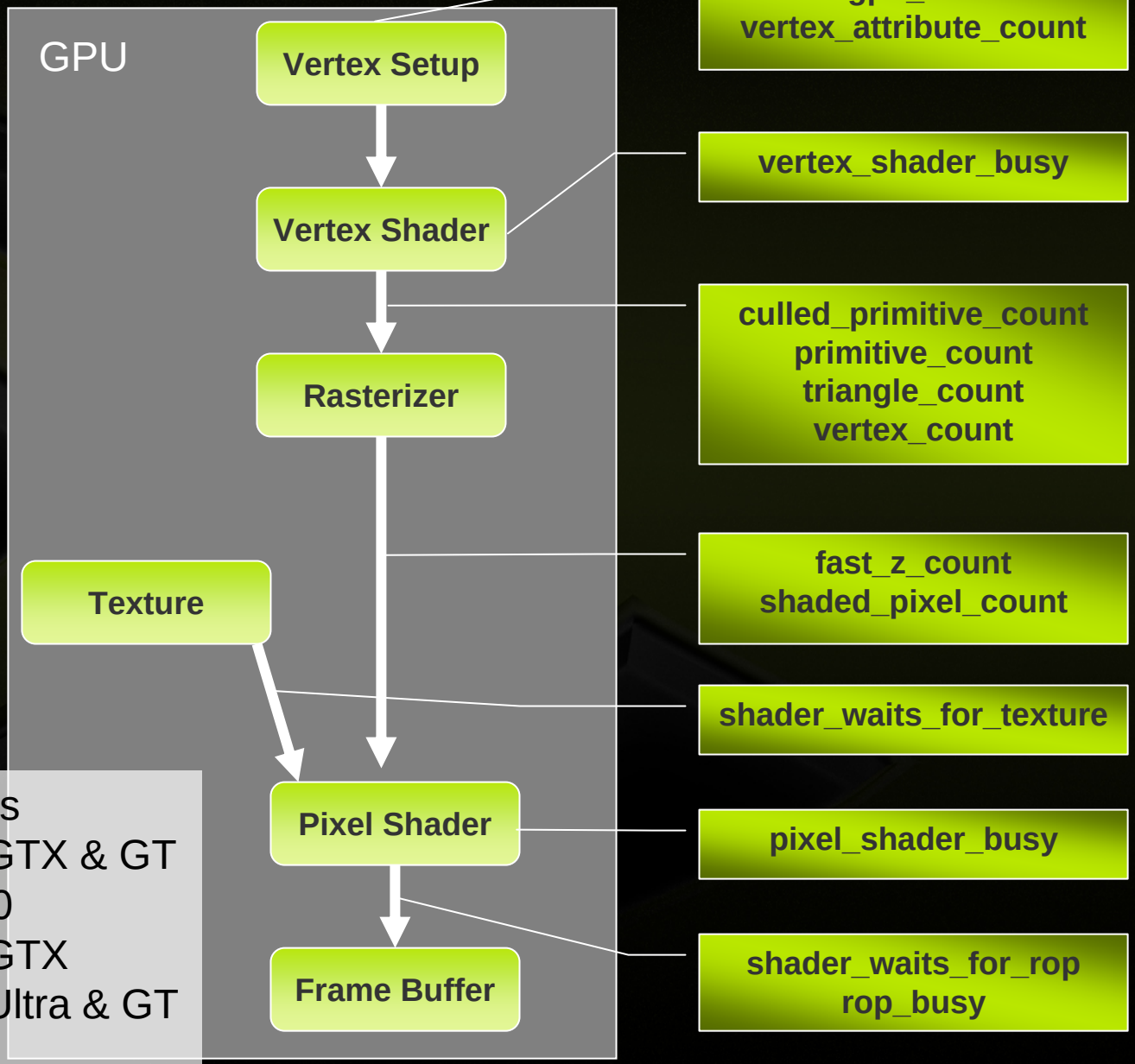
NVPerfKit: Direct3D Counters

- General
  - FPS
  - ms per frame
- Driver
  - Driver frame time (total time spent in driver)
  - Driver sleep time (waiting for GPU)
- Counts
  - Triangles
  - Instanced triangle
  - Batches
  - Locked render targets
- Memory
  - AGP memory used in MB and bytes
  - Video memory used and total in MB and bytes

NVPerfKit: OpenGL Counters

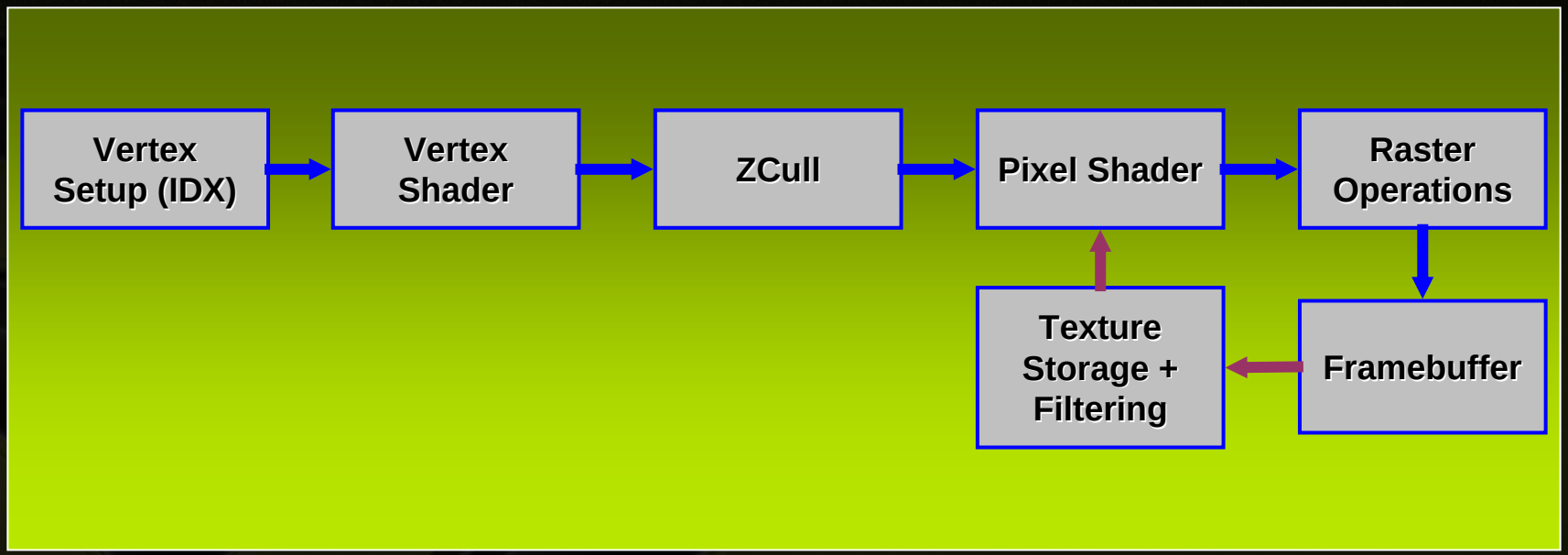
- General
  - FPS
  - ms per frame
- Driver
  - Driver frame time (total time spent in driver)
  - Driver sleep time (waiting for GPU)
  - % of the frame time driver is waiting
- Counts
  - Batches
  - Vertices
  - Primitives
- Memory
  - AGP memory used in MB and bytes
  - Video memory used and total in MB and bytes

NVPerfKit: GPU Counters



Supported GPUs
GeForce 7900 GTX & GT
Quadro FX 4500
GeForce 7800 GTX
GeForce 6800 Ultra & GT
GeForce 6600

NEW! Simplified Experiments



- Utilization and bottleneck experiments for each unit
- GPU Bottleneck experiment
  - Adds all bottleneck and utilization experiments
  - Expert system analyzes the results
- Exposed via NVPerfAPI

What is NVPerfHUD?

- Direct3D PERFormance Heads Up Display
  - Overlay graphs and debugging tools on top of your application
  - Interactive tools for debugging and performance tuning
- 4 different HUDs
  - Performance Dashboard
  - Debug Console
  - Frame Debugger
  - Frame Profiler (New in 4.0)

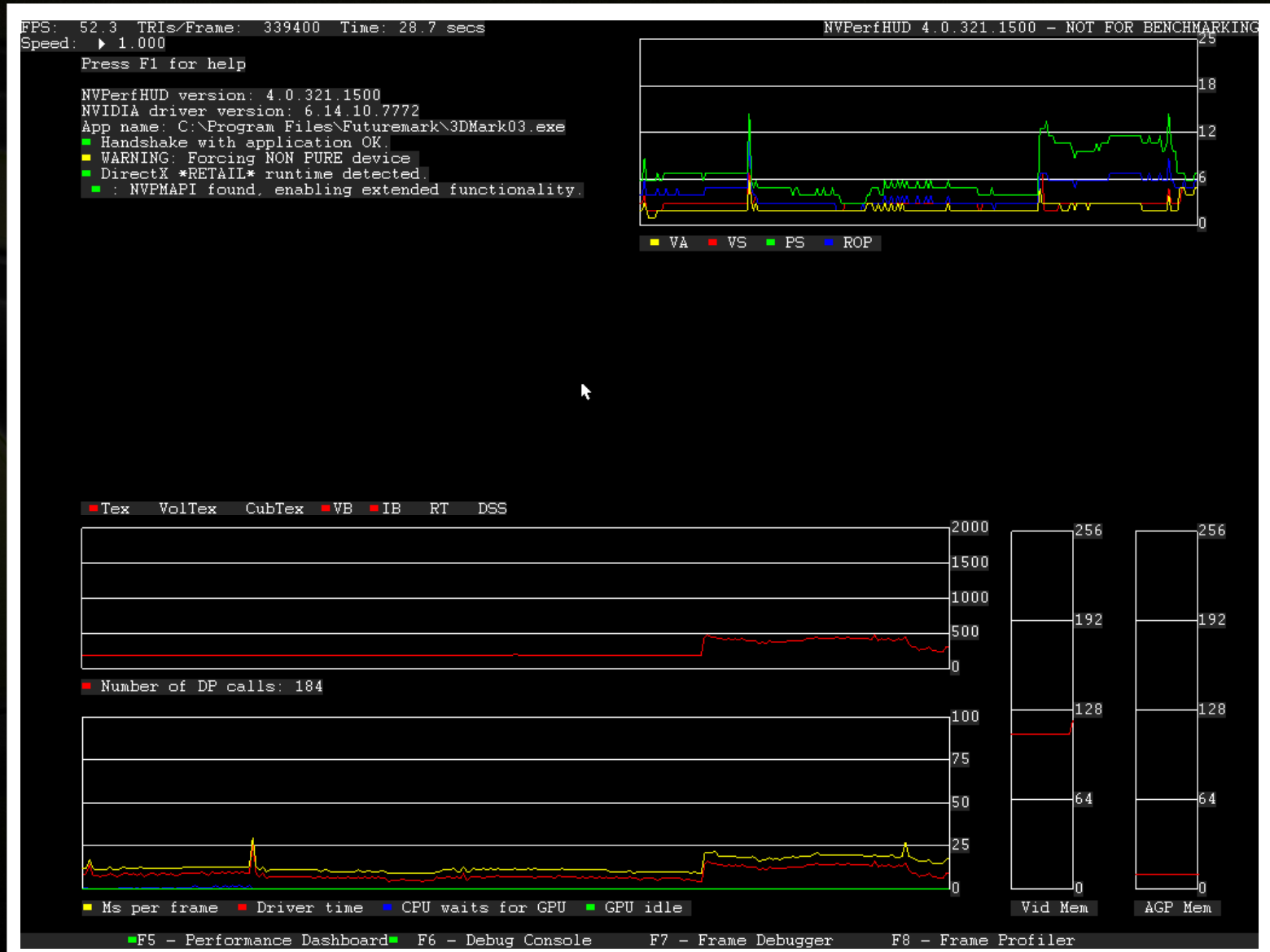
How to use it

- Drag and drop your application onto the NVPerfHUD icon
- Run through your application as you normally do until you find:
  - Functional problems: use the Frame Debugger
  - Performance problems: use the Dashboard graphs and Frame Profiler

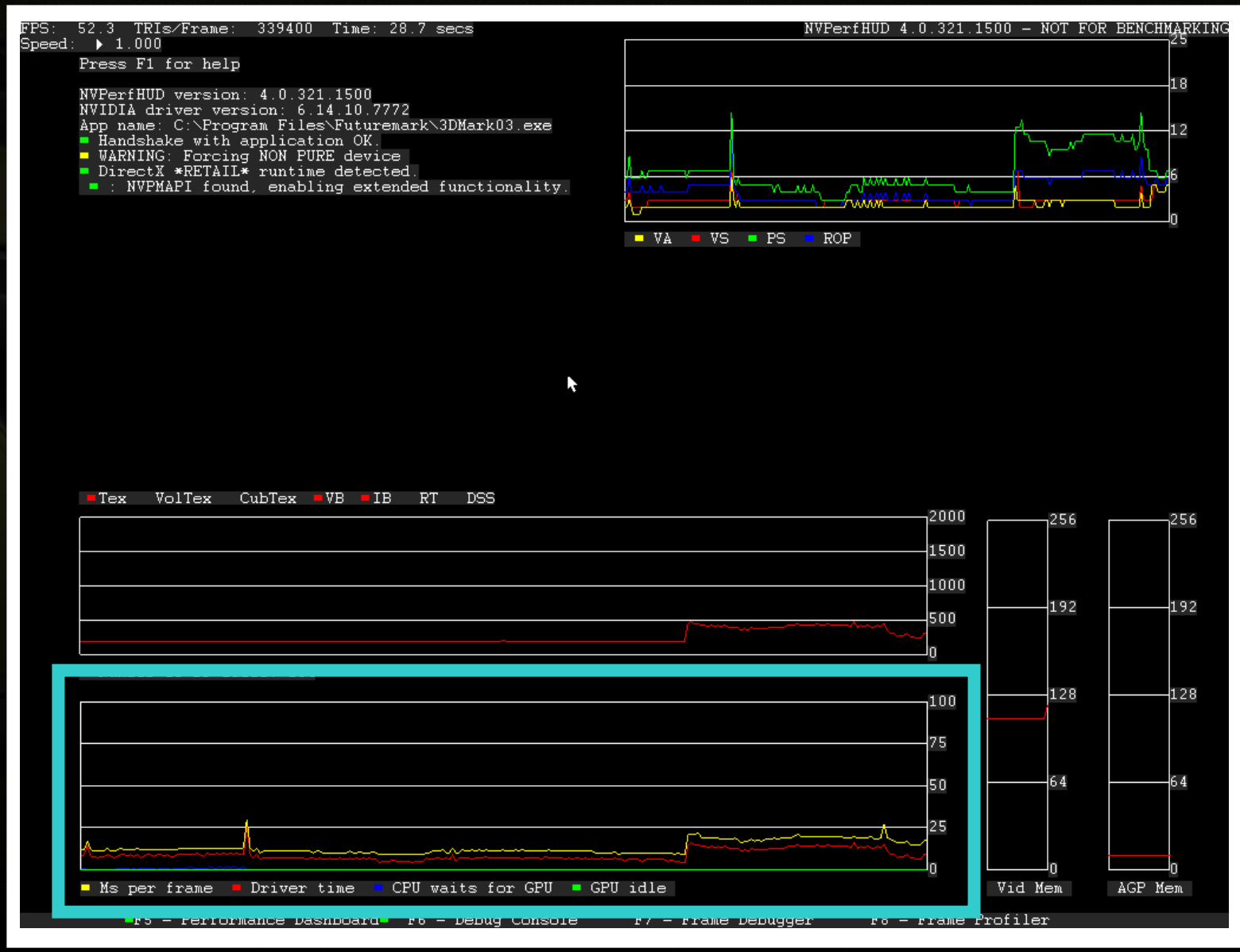
Demo: NVPerfHUD

Welcome Sim Dietrich!!

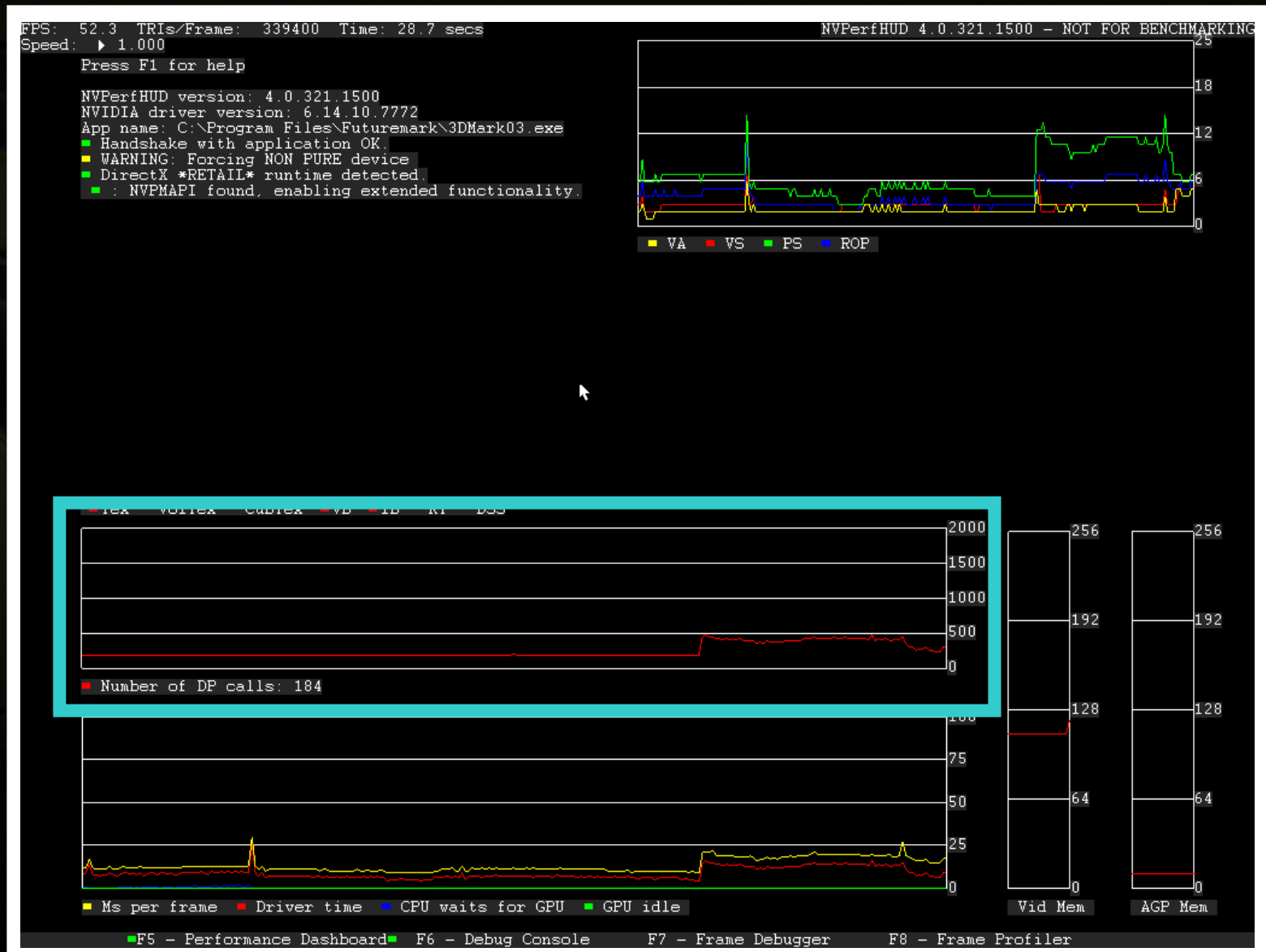
Demo: Performance Dashboard



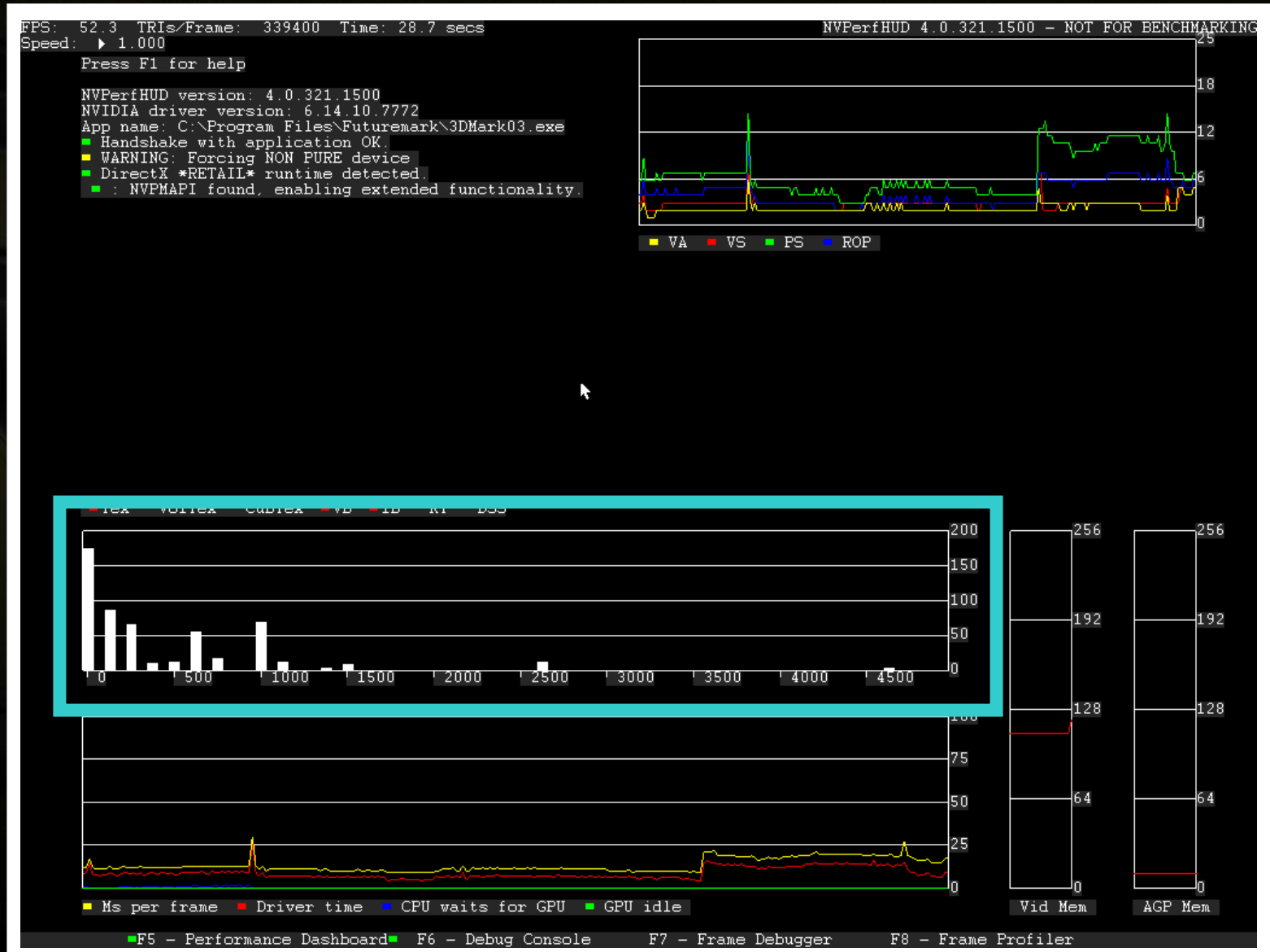
Demo: Performance Dashboard



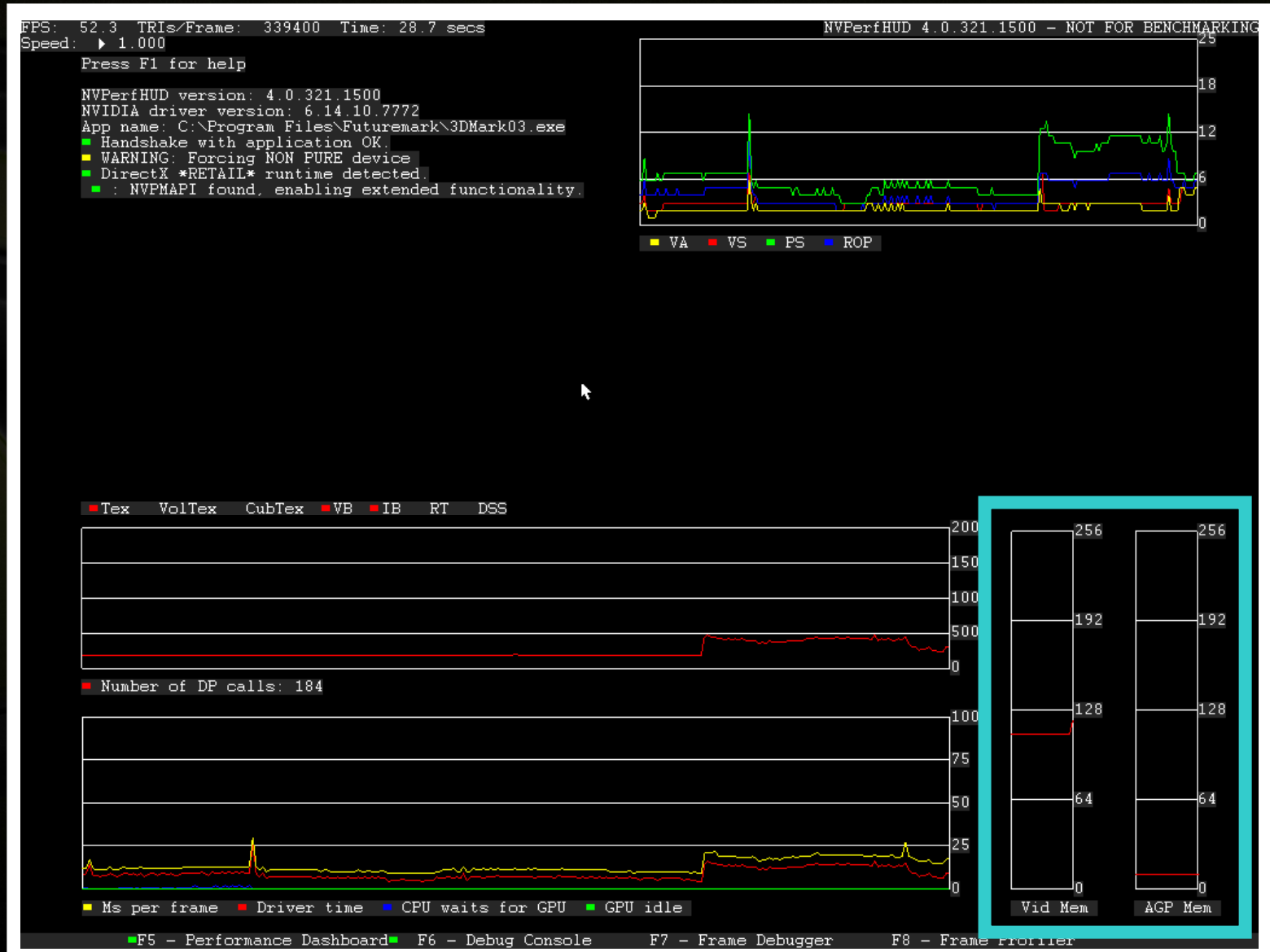
Demo: Performance Dashboard



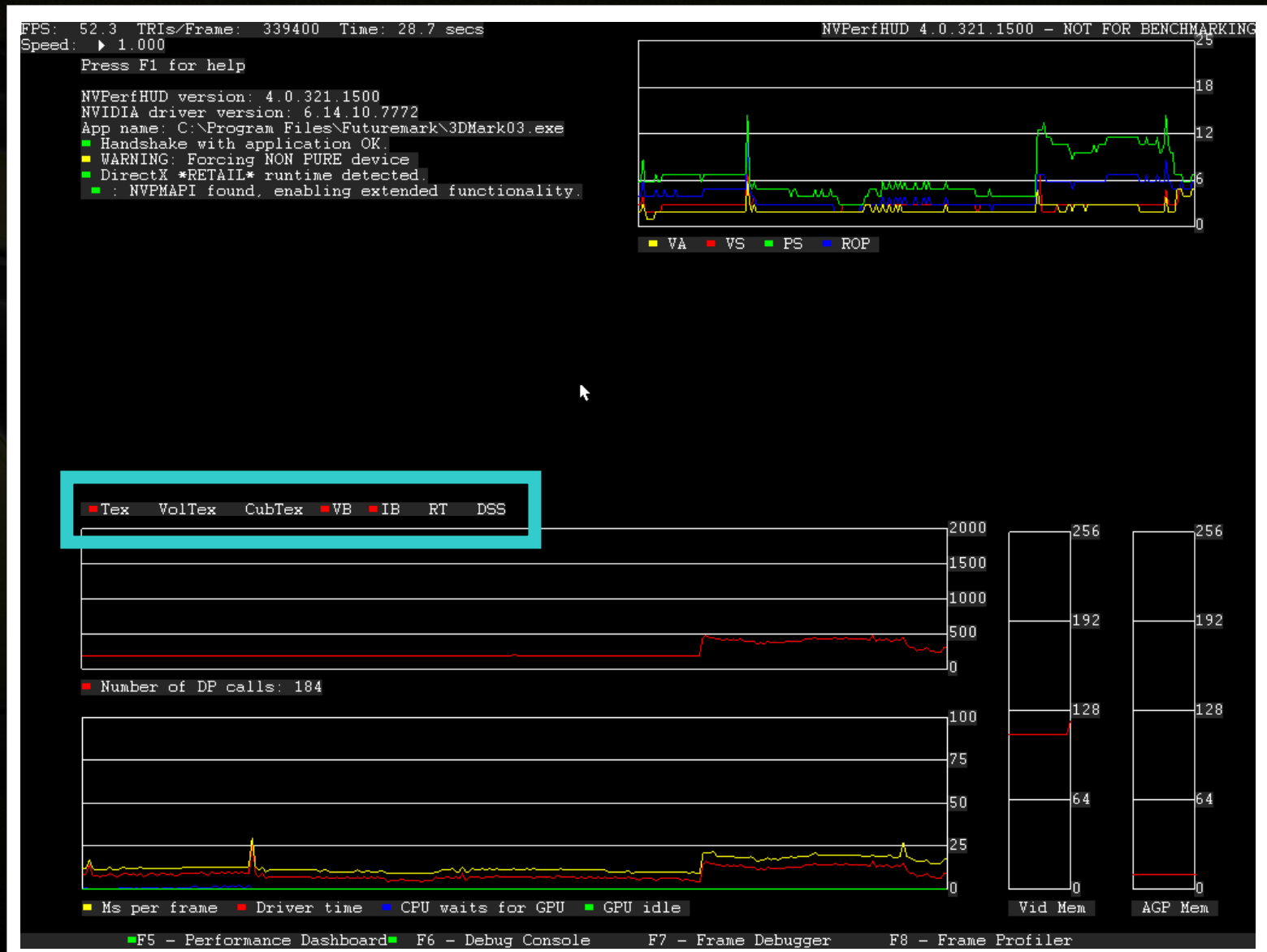
Demo: Performance Dashboard



Demo: Performance Dashboard

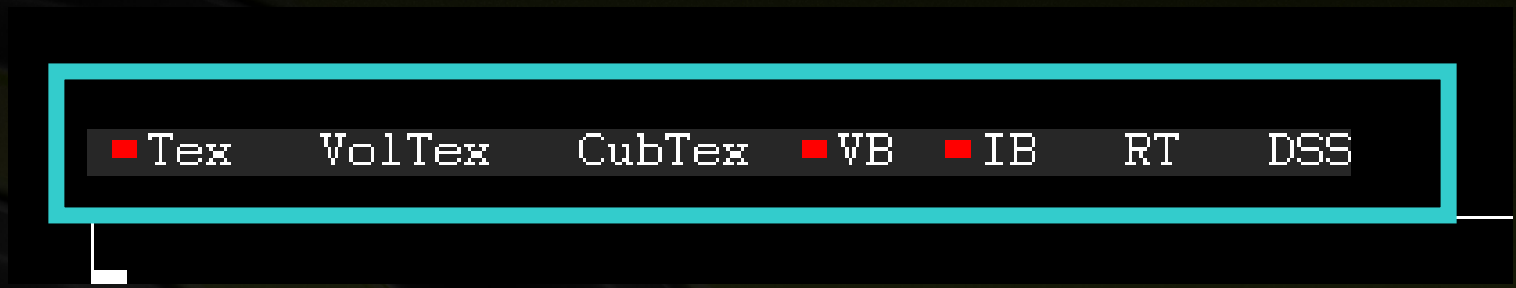


Demo: Performance Dashboard



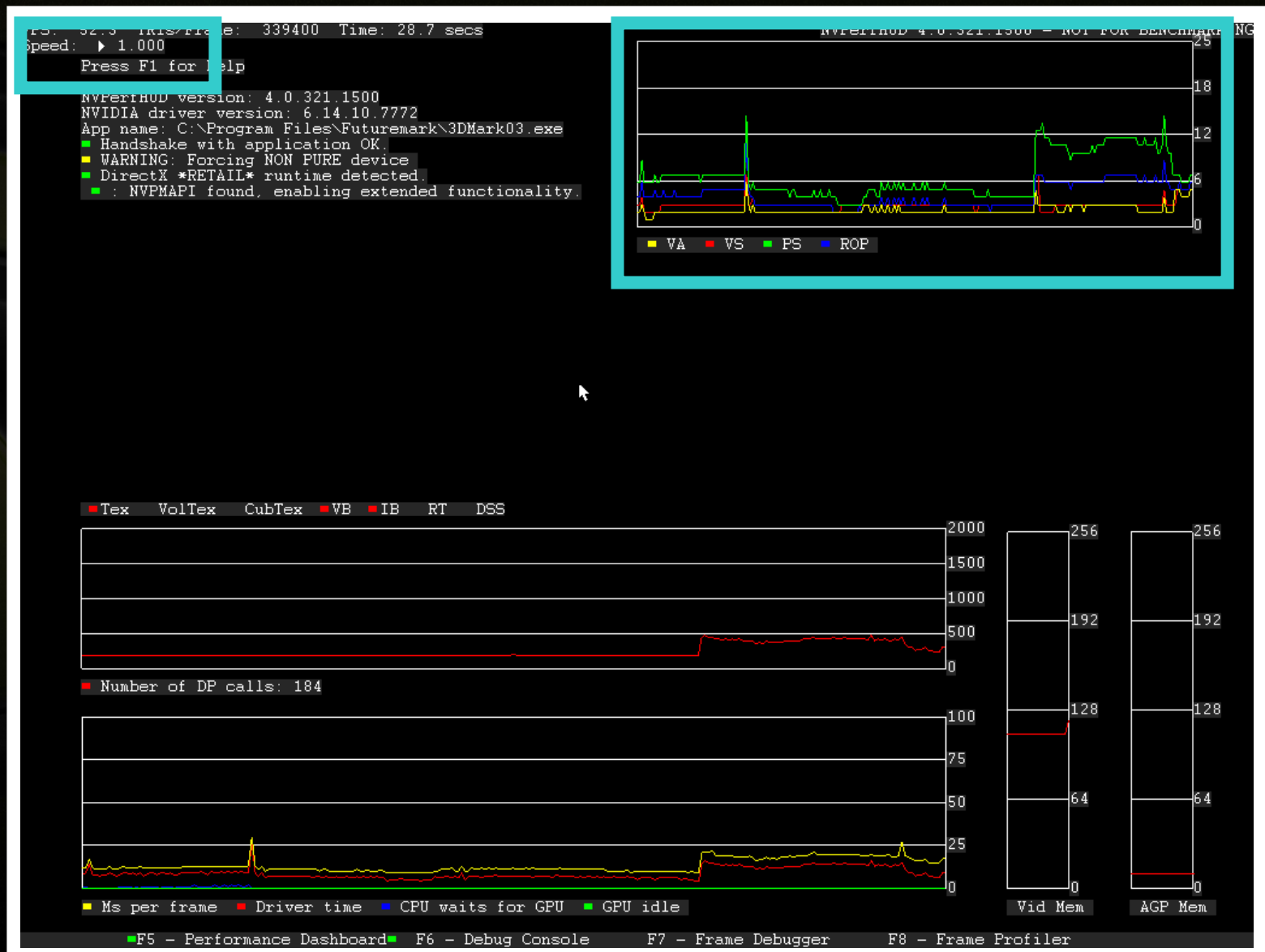
Demo: Performance Dashboard

Resource creation monitor



- **Resources monitored**
  - **Textures**
  - **Volume Textures**
  - **Cube textures**
  - **Vertex Buffers**
  - **Index buffers**
  - **Stencil and depth surfaces**

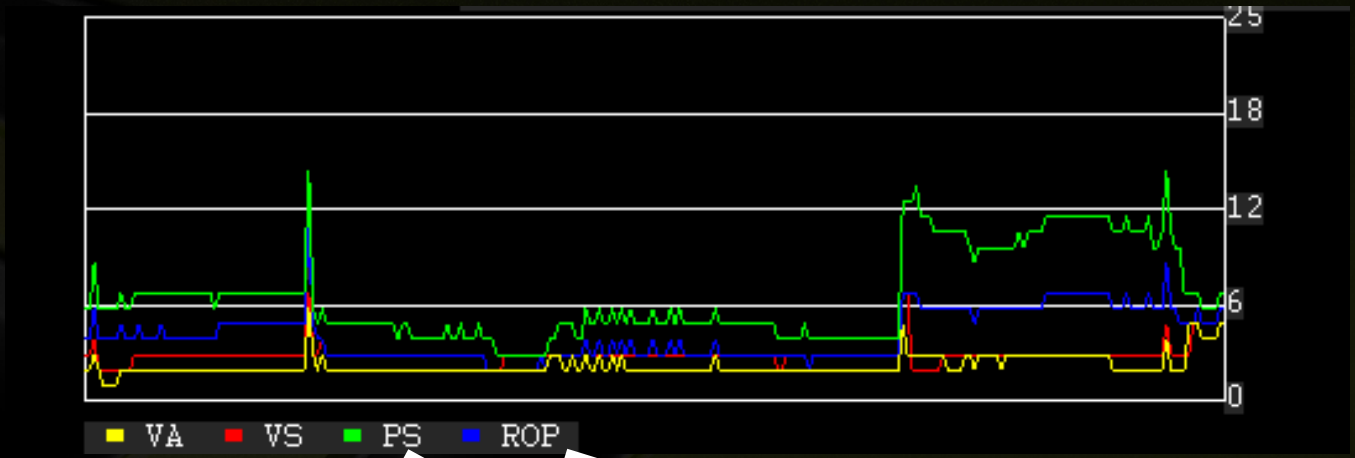
Demo: Performance Dashboard

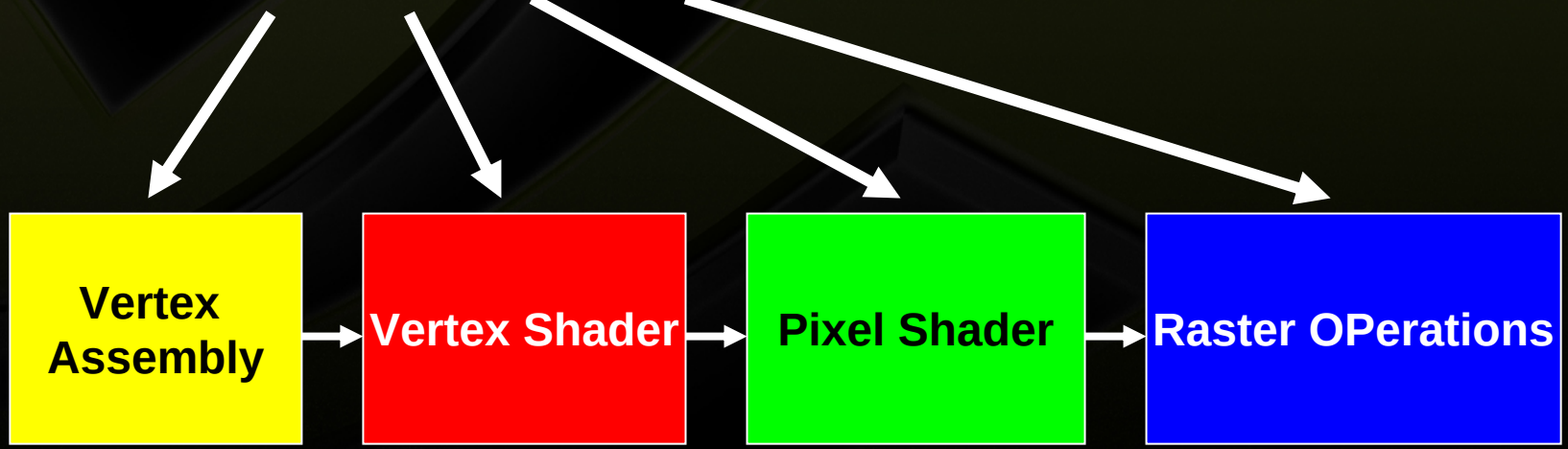


Demo: Performance Dashboard

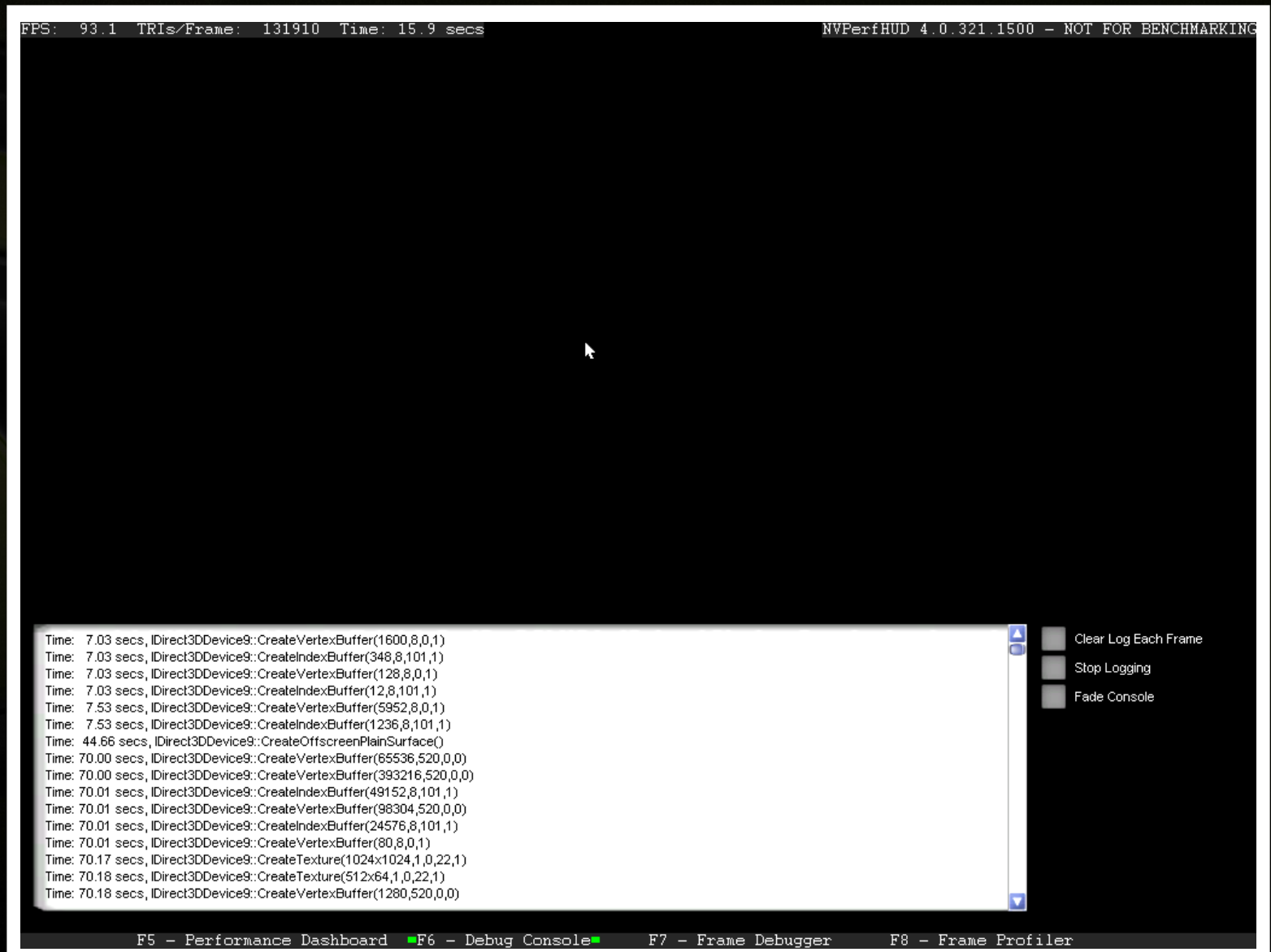
Speed control

Demo: The simplified graphics pipeline

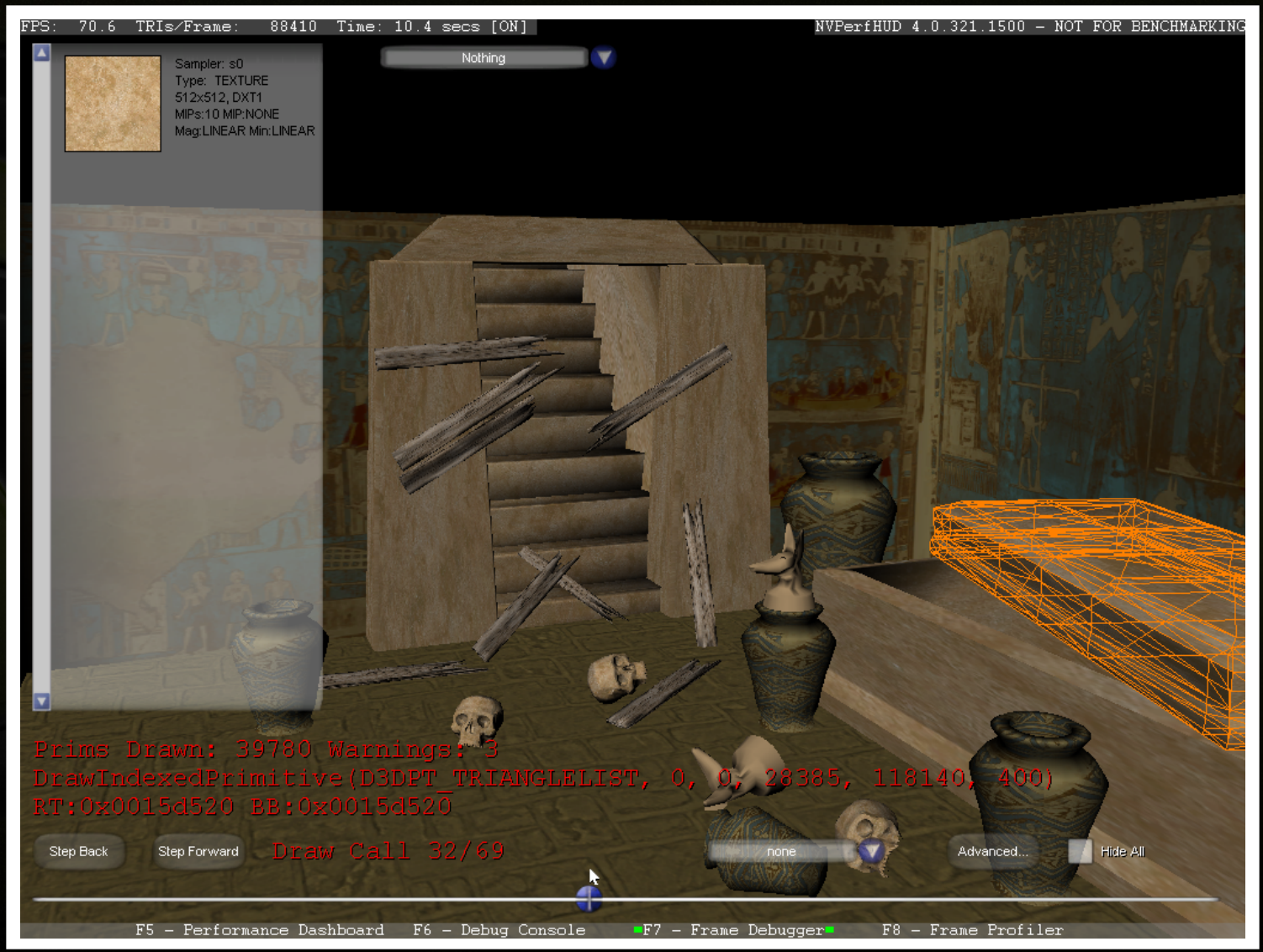




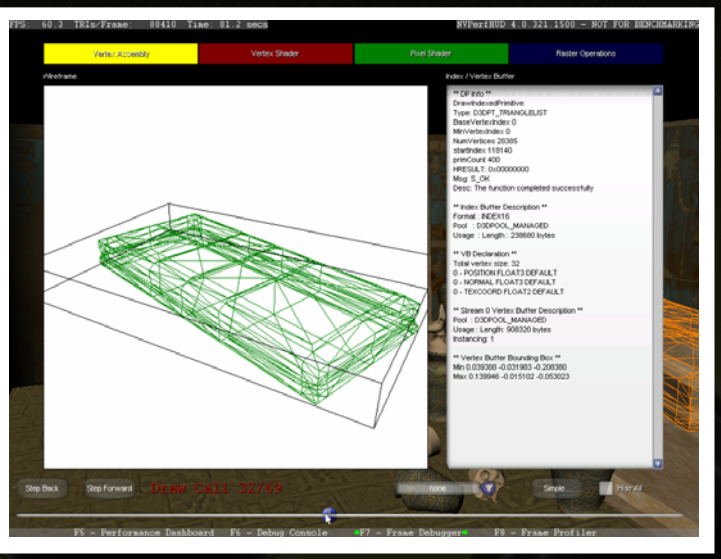
Demo: Debug Console

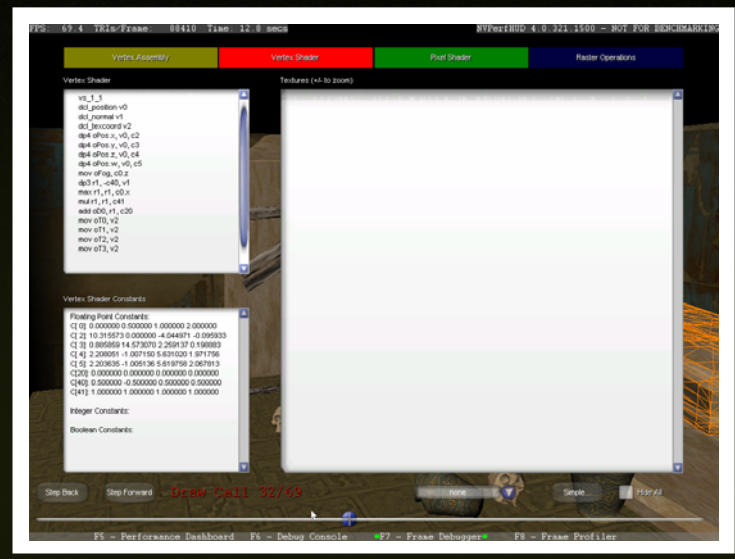


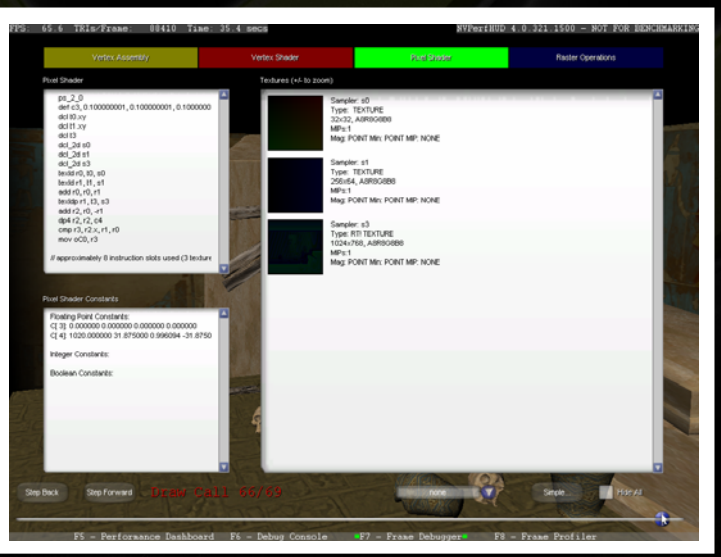
Demo: Frame Debugger

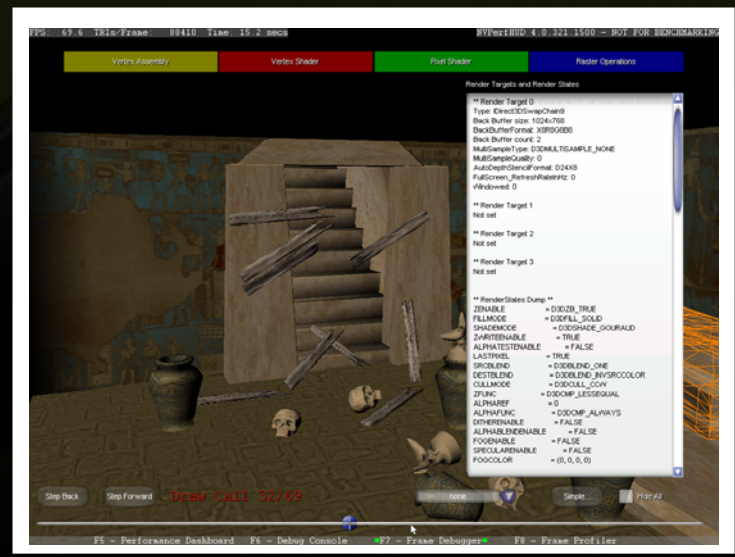


Demo: Advanced Frame Debug









Frame Profiler

- NVPerfHUD uses NVPerfKit and SimExp
- Samples ~40 Performance Counters (PCs)
- Can not read all of them at the same time
- Need to render THE SAME FRAME until all the PCs are read

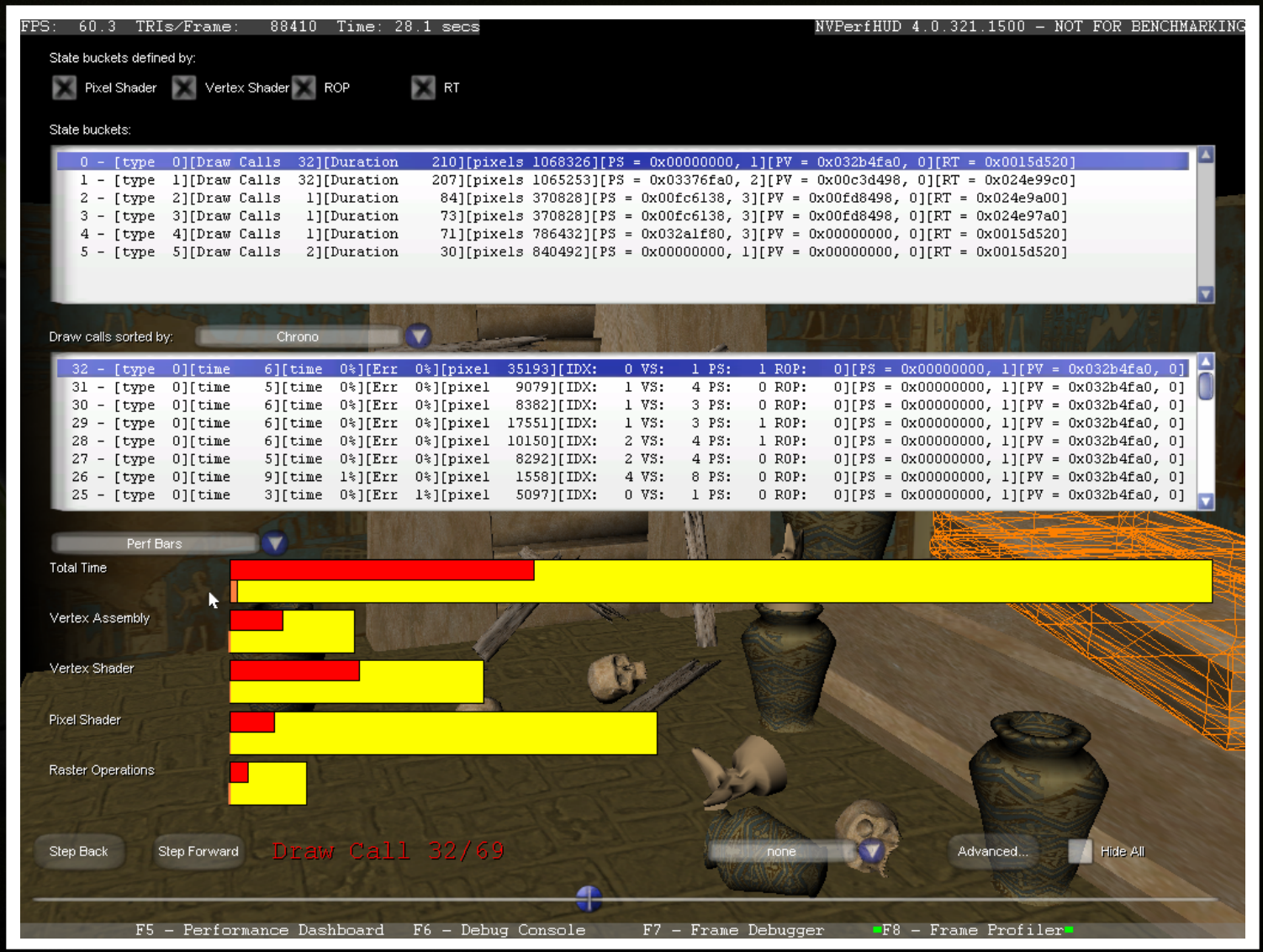
Frame Profiler: Optimization Strategy

- Group by render state (“state buckets”): helps show most expensive states to render
- Identify the bottleneck for the most expensive state bucket
- Curing the bottleneck with a common corrective action should result in increased performance
- Iterate…

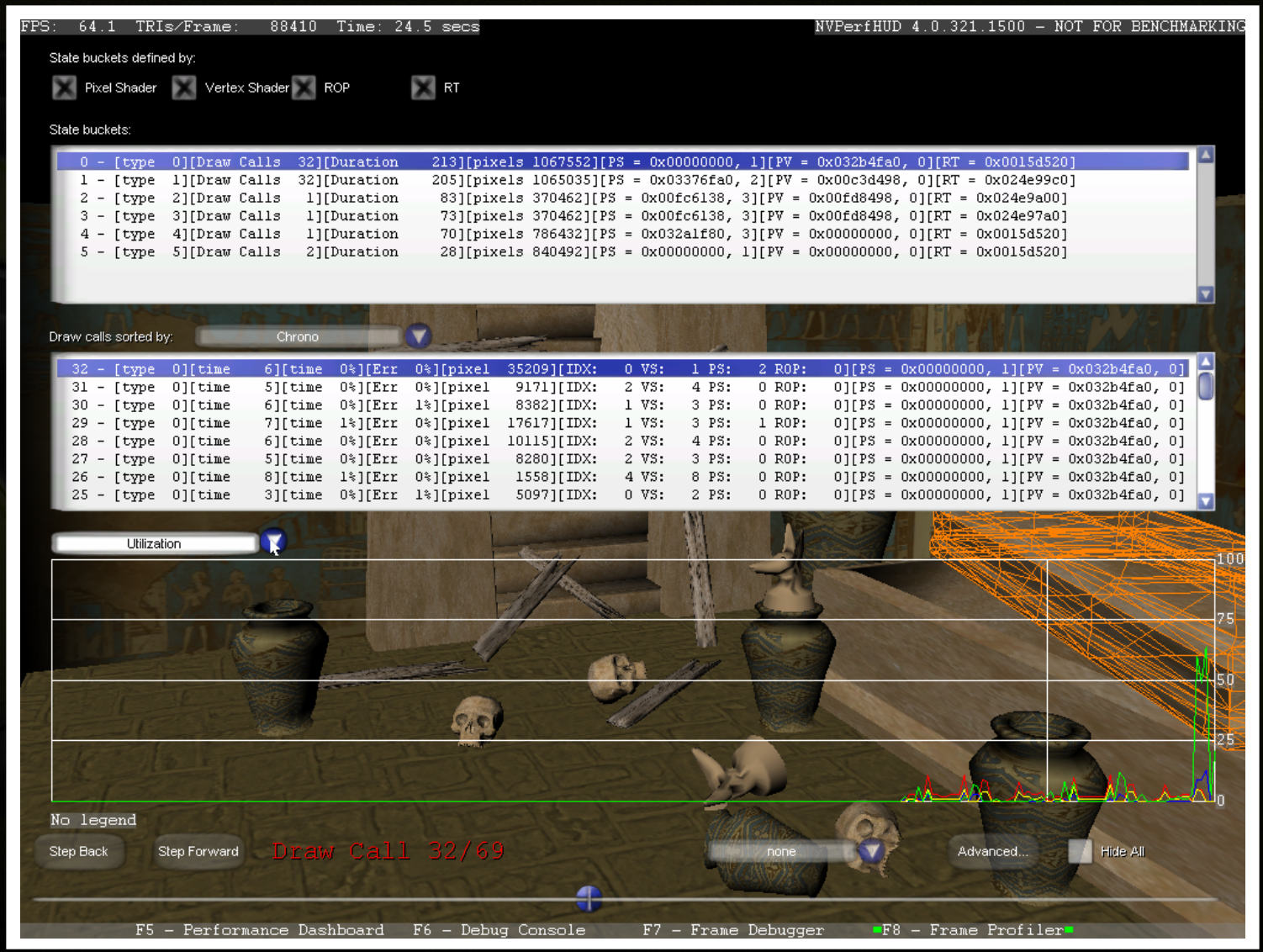
- NEED TO ADD INFO ABOUT THE GRAPHS

Demo: Frame Profiler measuring

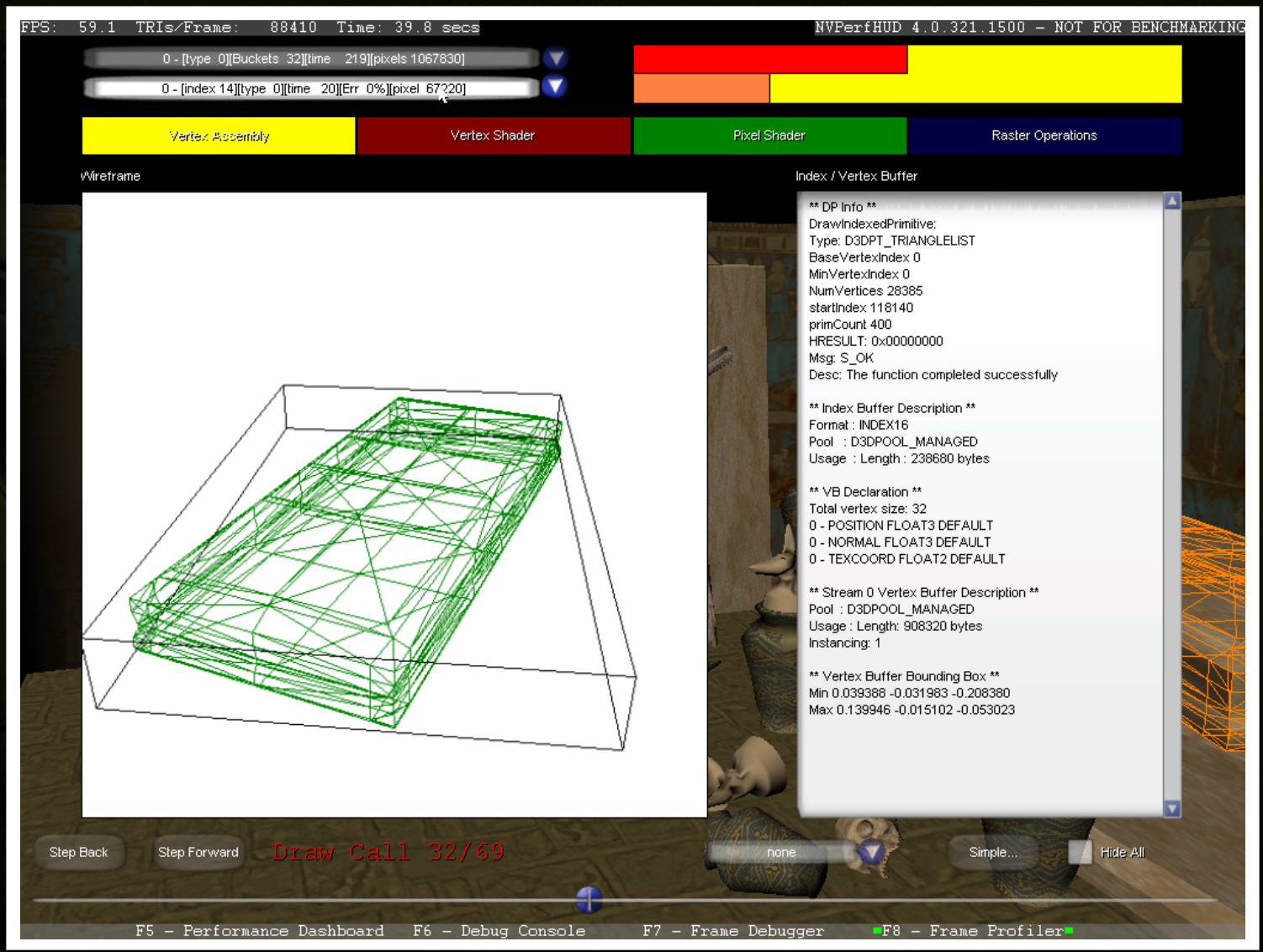
Demo: Frame Profiler



Demo: Frame Profiler



Demo: Advanced Frame Profiler



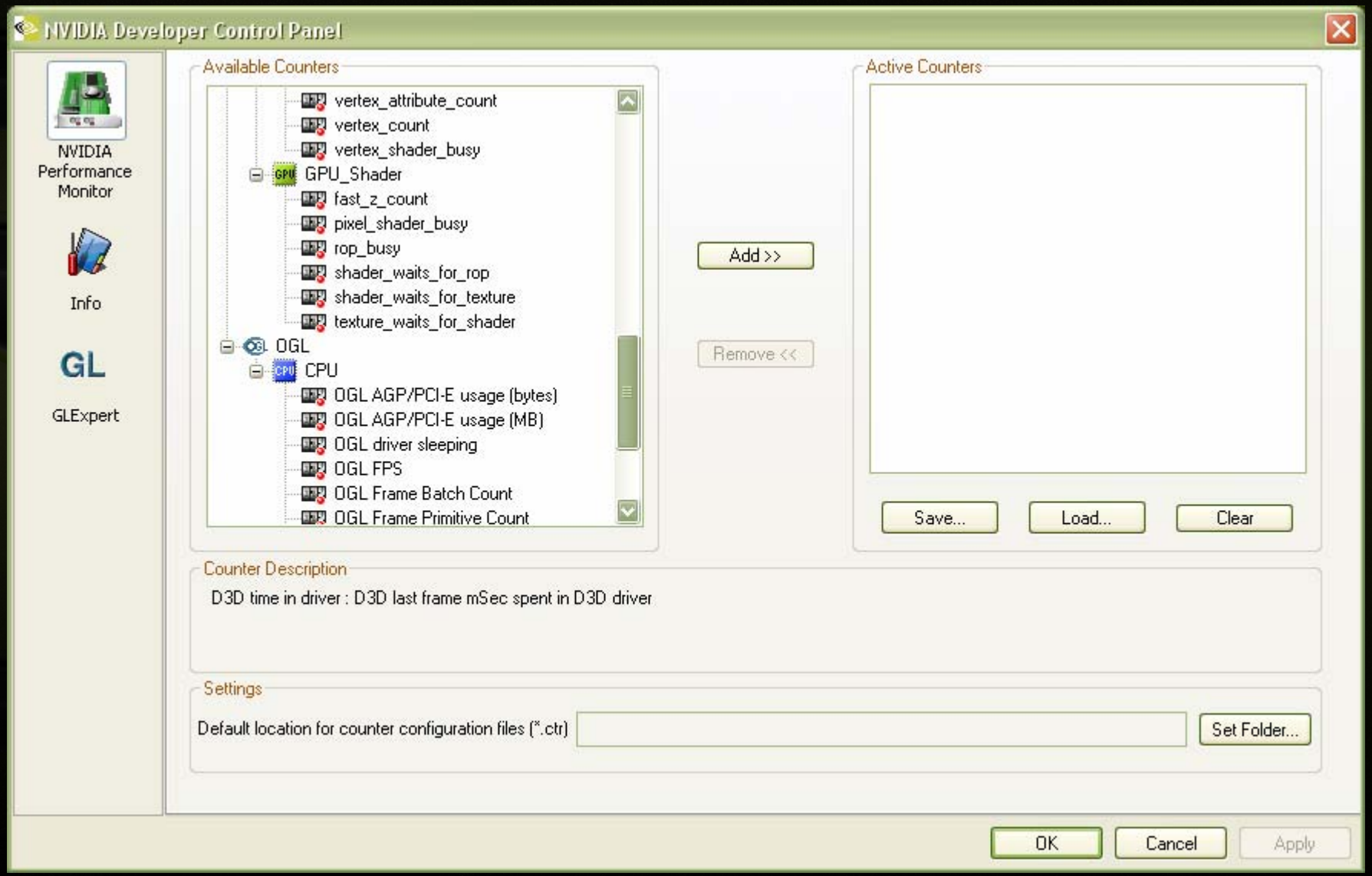
Freezing the application

- Only possible if the application uses time-based animation
- Stop the clock
  - Intercept: QueryPerformanceCounter(), timeGetTime()
  - NO RDTSC!!
- Pos += V * DeltaTime

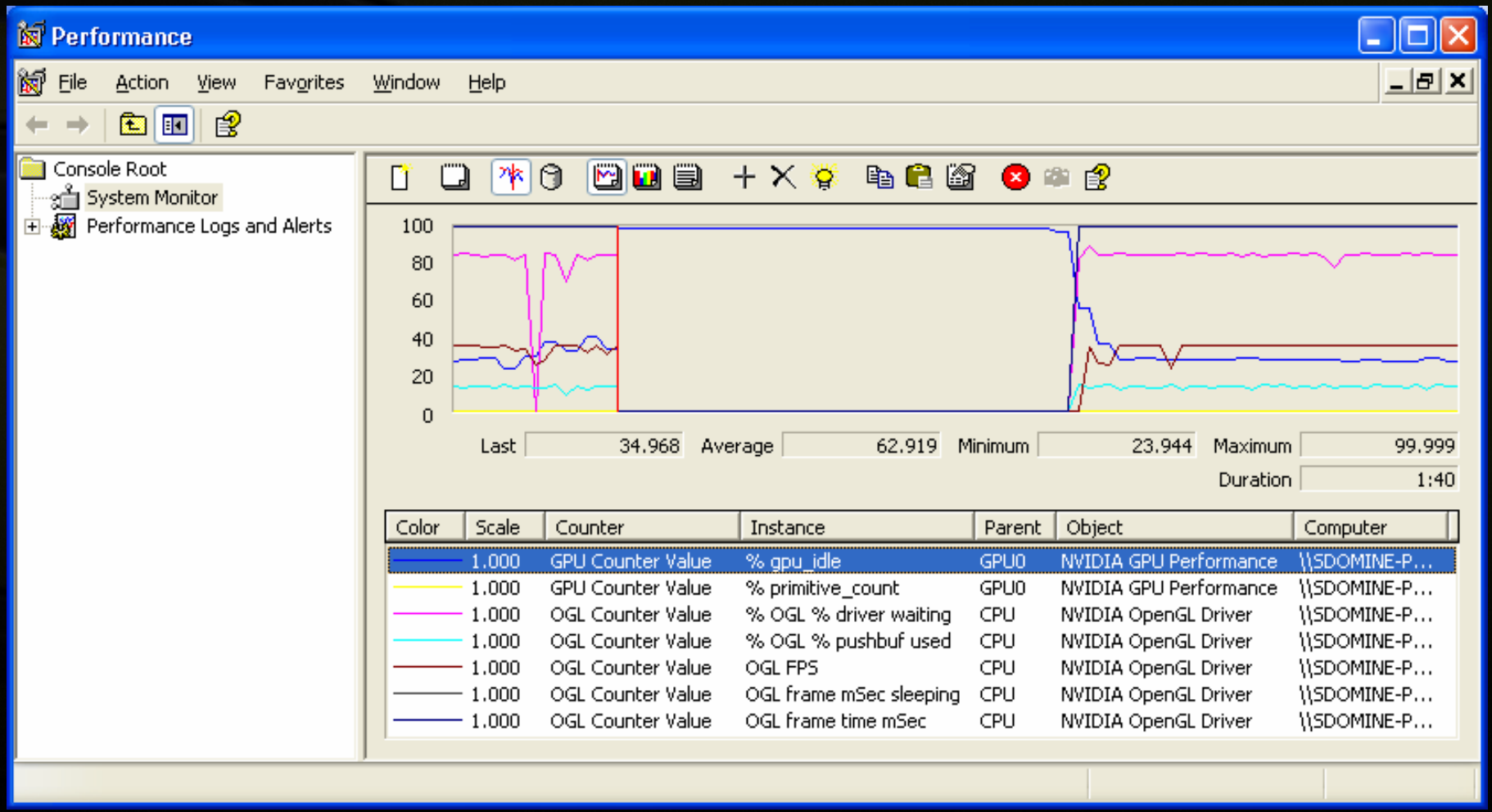
How do I use NVPerfKit counters?

- PDH: Performance Data Helper for Windows
  - Win32 API for exposing performance data to user applications
  - Standard interface, many providers and clients
  - Sample code and helper classes provided in NVPerfSDK
- Perfmon: (aka Microsoft Management Console)
  - Win32 PDH client application
  - Perfmon's sampling frequency is low (1X/s)
  - Displays PDH based counter values:
    - OS: CPU usage, memory usage, swap file usage, network stats, etc.
    - NVIDIA: all of the counters exported by NVPerfKit
- Good for rapid prototyping

Enable counters: NVDevCPL



Graphing results: Perfmon



NEW! NVPerfAPI

- NVIDIA API for easy integration of NVPerfKit
  - No more enable counters in NVDevCPL, run app separately
  - No more lag from PDH
- Simplified Experiments
  - Targeted, multipass experiments to determine GPU bottleneck
  - Automated analysis of results to show bottlenecked unit
- Use cases
  - Real time performance monitoring using GPU and driver counters, round robin sampling
  - Simplified Experiments for single frame analysis

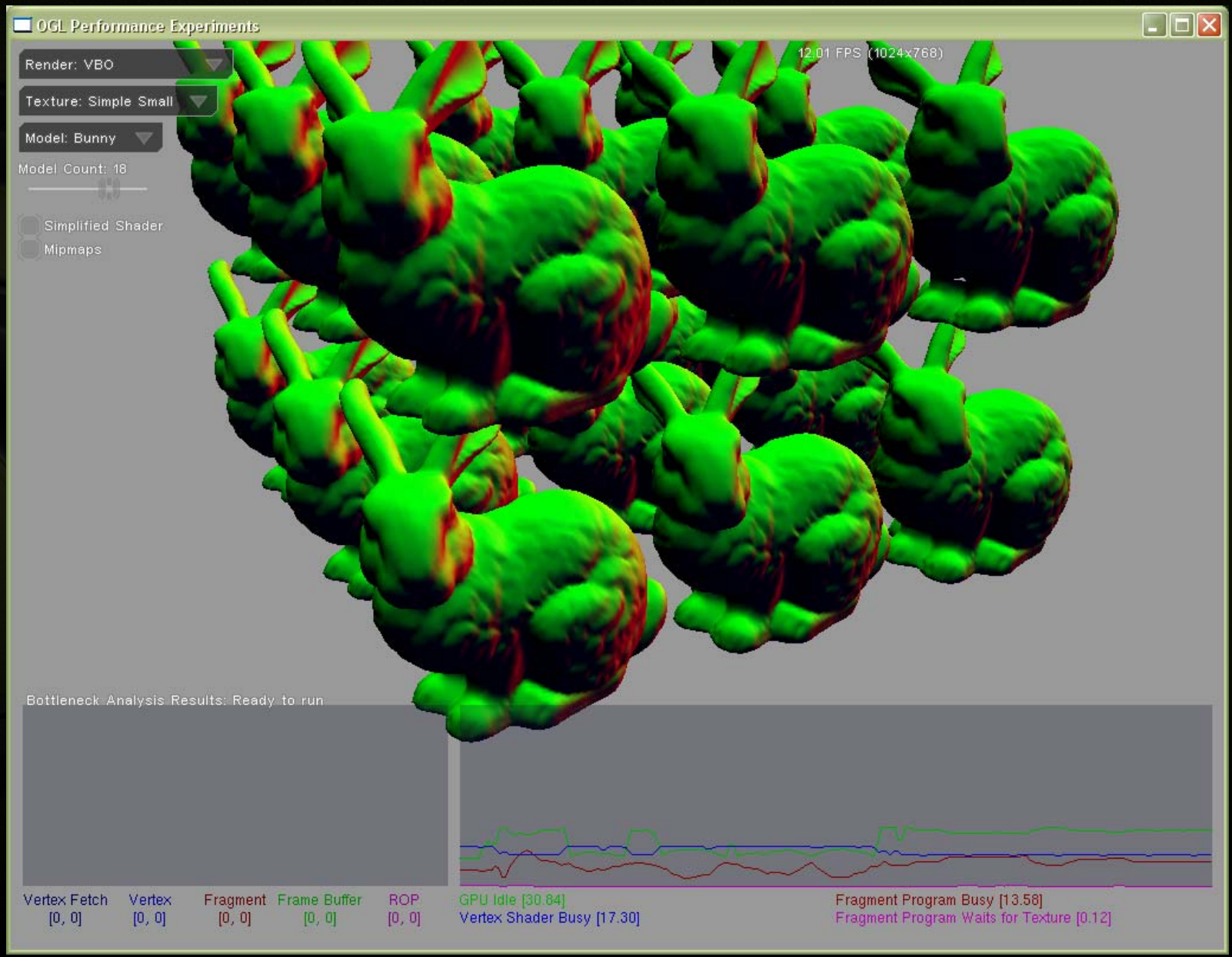
NVPerfAPI: Real Time

```
// Somewhere in setup
NVPMAddCounterByName(“vertex_shader_busy”);
NVPMAddCounterByName (“pixel_shader_busy”);
NVPMAddCounterByName (“shader_waits_for_texture”);
NVPMAddCounterByName (“gpu_idle”);

// In your rendering loop, sample using names
NVPMSample(NULL, &nNumSamples);
NVPMGetCounterValueByName(“vertex_shader_busy”, 0, &nVSEvents, &nVSCycles);
NVPMGetCounterValueByName(“pixel_shader_busy”, 0, &nPSEvents, &nPSCycles);
NVPMGetCounterValueByName(“shader_waits_for_texture”, 0, &nTexEvents,
    &nTexCycles);
NVPMGetCounterValueByName(“gpu_idle”, 0, &nIdleEvents, &nIdleCycles);
```

NVPerfAPI: Real Time

```
// Somewhere in setup
nVSBusy = NVPMGetCounterByName(“vertex_shader_busy”);
NVPMAddCounter(nVSBusy);
nPSBusy = NVPMGetCounterByName(“pixel_shader_busy”);
NVPMAddCounter(nPSBusy);
nWaitTexture = NVPMGetCounterByName(“shader_waits_for_texture”);
NVPMAddCounter(nWaitTexture);
nGPUIdle = NVPMGetCounterByName(“gpu_idle”);
NVPMAddCounter(nGPUIdle);


// In your rendering loop, sample using IDs
NVPMSample(aSamples, &nNumSamples);
for(ii = 0; ii < nNumSamples; ++ii) {
    if(aSamples[ii].index == nVSBusy) {
    }
    if(aSamples[ii].index == nPSBusy) {
    }
    if(aSamples[ii].index == nWaitTexture) {
    }
    if(aSamples[ii].index == nGPUIdle) {
    }
}
```

NVPerfAPI: Real time sampling



NVPerfAPI: Simplified Experiments

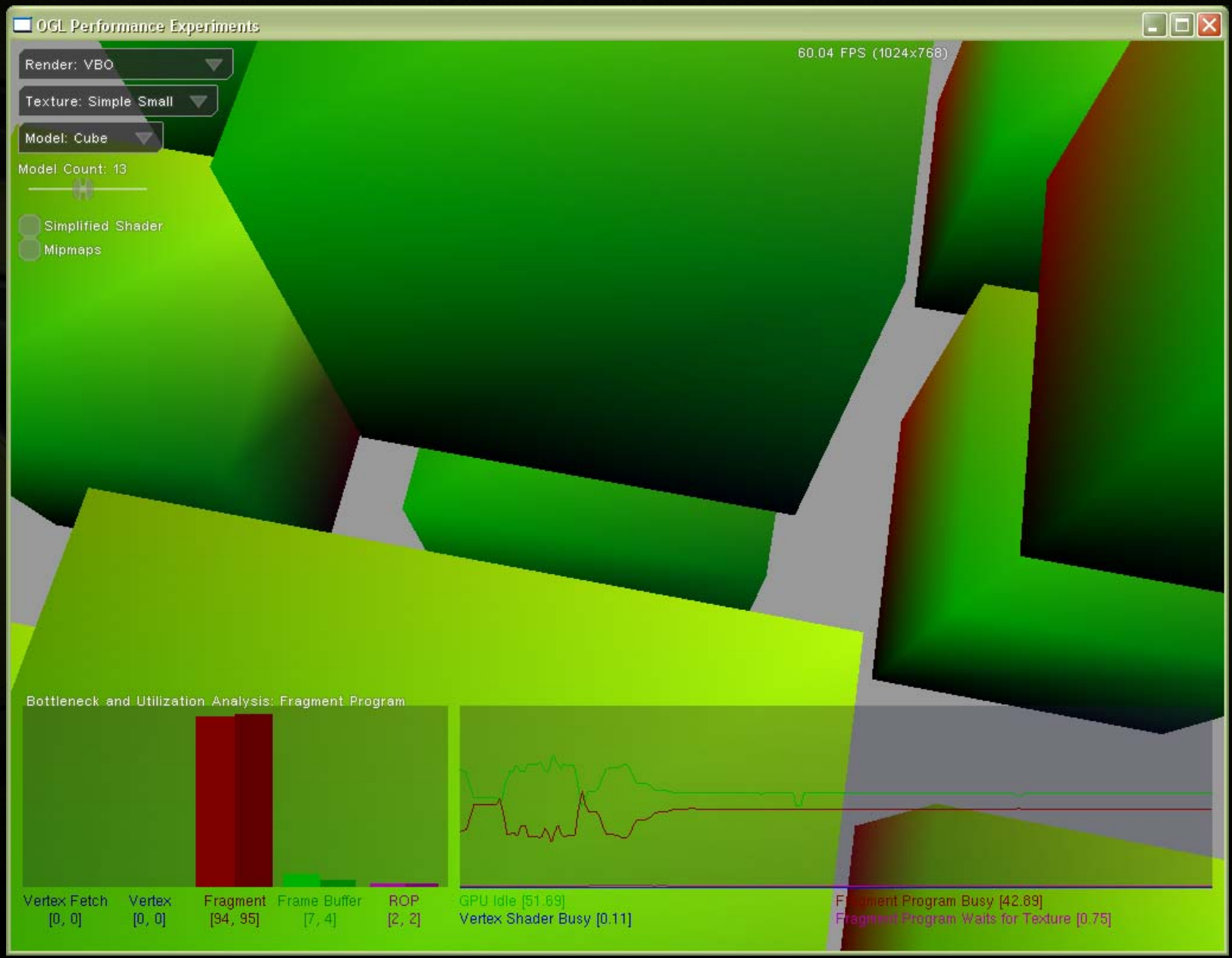
```
NVPMAddCounter(“GPU Bottleneck”);
NVPMAllocObjects(50);

NVPMBeginExperiment(&nNumPasses);
for(int ii = 0; ii < nNumPasses; ++ii) {
     // Setup the scene, clear Zbuffer/render target

     NVPMBeginPass(ii);

     NVPMBeginObject(0);
     // Draw calls associated with object 0 and flush
     NVPMEndObject(0);

     NVPMBeginObject(1);
     // Draw calls associated with object 1 and flush
     NVPMEndObject(1);

     // ...

     NVPMEndPass(ii);
}

NVPMEndExperiment();
NVPMGetCounterValueByName(“GPU Bottleneck”, 0, &nGPUBneck, &nGPUCycles);
NVPMGetGPUBottleneckName(nGPUBneck, pcString); // Convert to name

// End scene/present/swap buffers
```

NVPerfAPI: Simplified Experiments

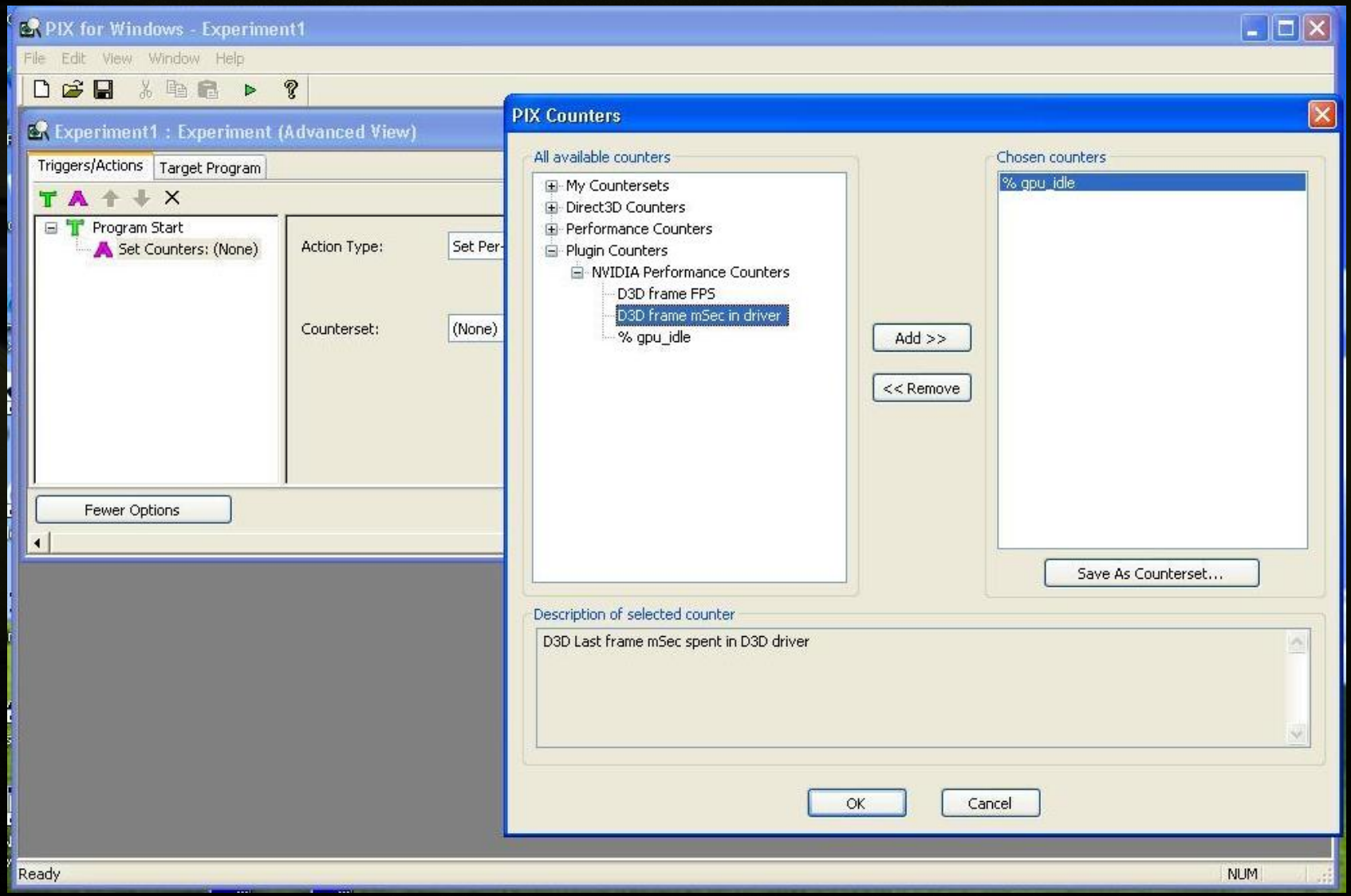
- GPU Bottleneck experiment
  - Run bottleneck and utilization experiments on all units
  - Process results to find bottlenecked unit
- Individual unit information can be queried
- Can run individual unit experiments
- Events: % utilization or % bottleneck…best way to visualize data
- Cycles: microseconds that the experiment ran, helps recompute the numerator for sorting

```
NVPMGetCounterValueByName(“IDX BNeck”, 0, &nIDXBneckEvents, &nIDXBNeckCycles);
NVPMGetCounterValueByName(“IDX SOL”, 0, &nIDXSOLEvents, &nIDXSOLCycles);
```

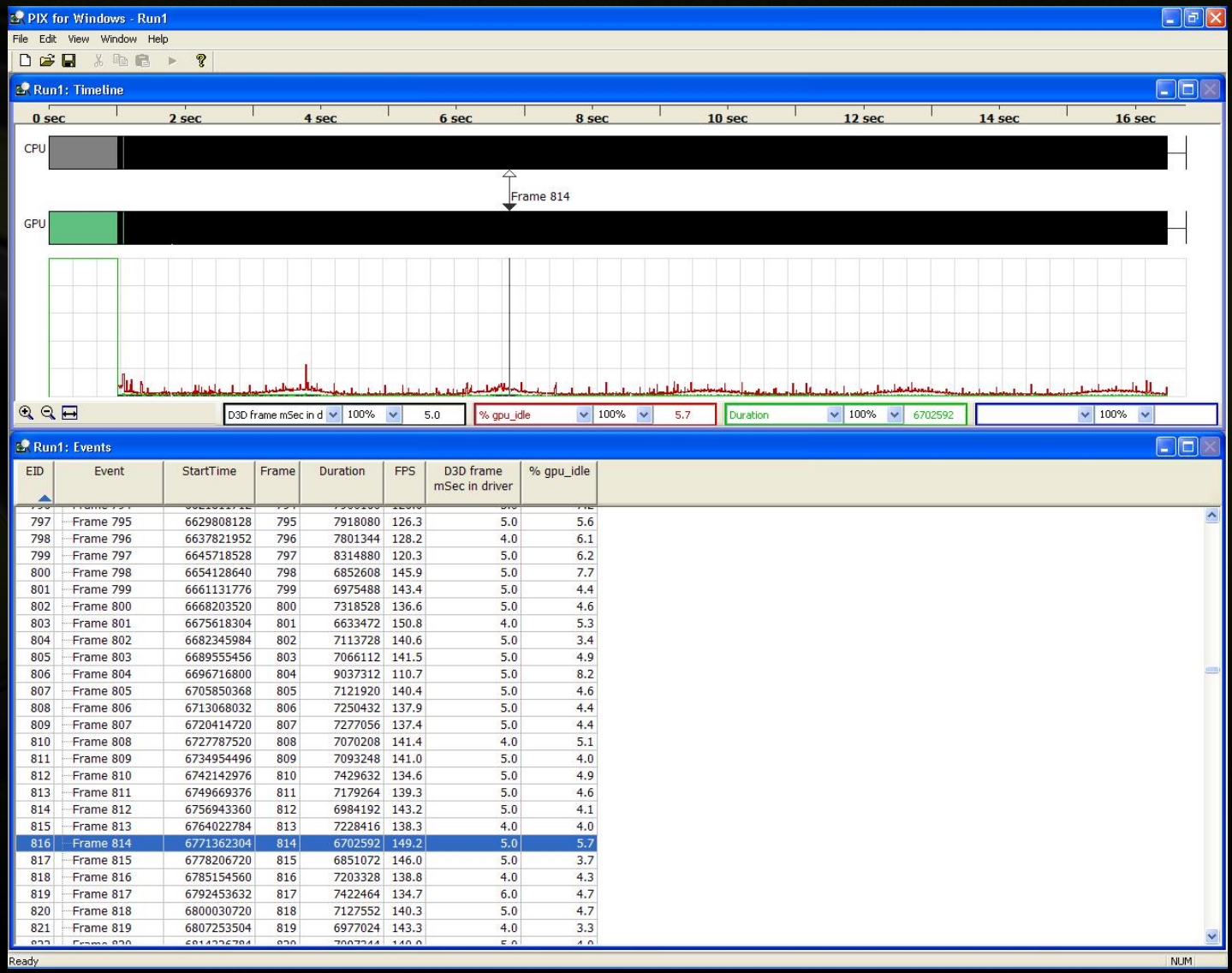
NVPerfAPI: SimExp



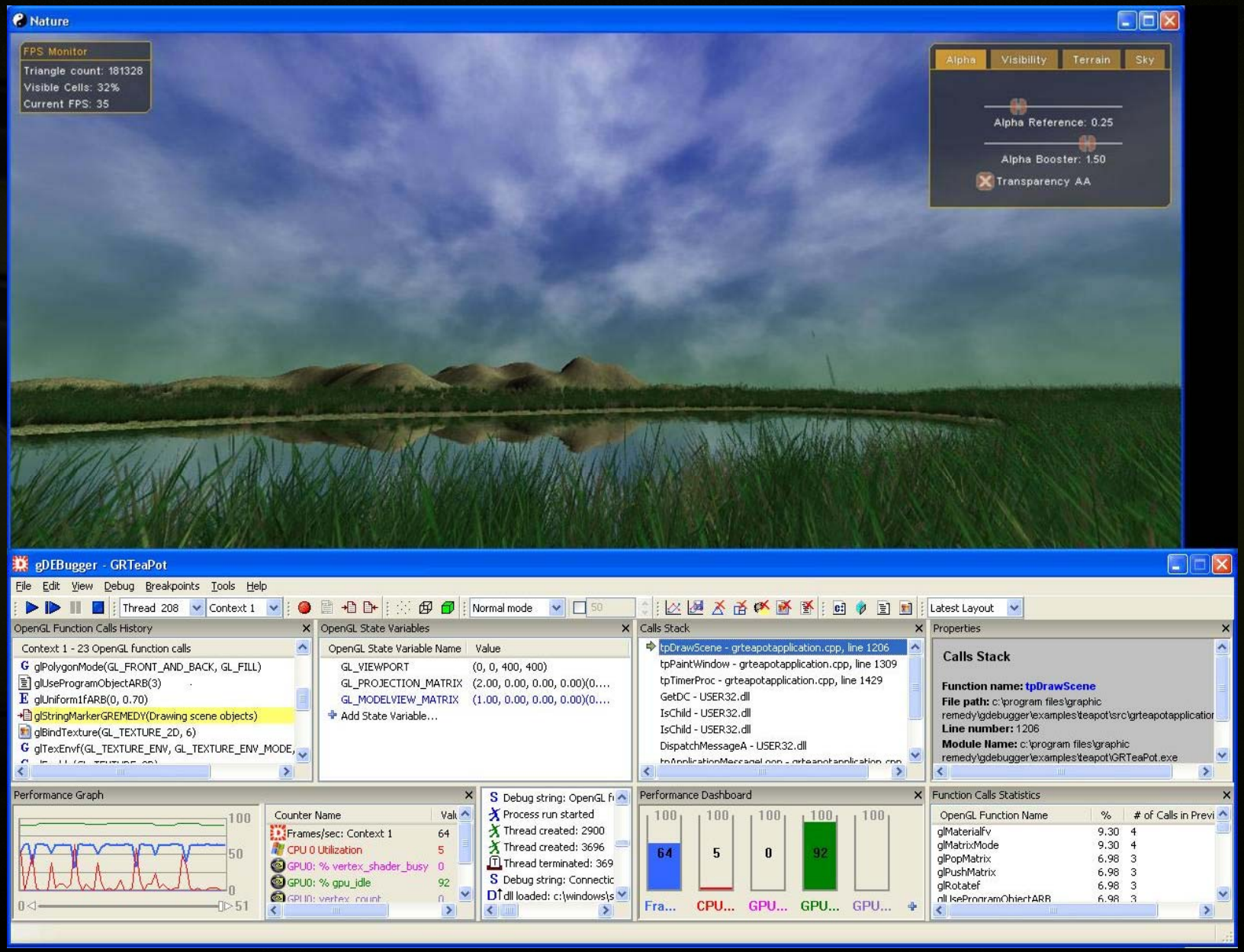
Associated Tools: NVIDIA Plug-In for Microsoft PIX for Windows



Associated Tools: NVIDIA Plug-In for Microsoft PIX for Windows



Graphic Remedy's gDEBugger



Solutions to common bottlenecks

- CPU Bound?
  - In your code:
    - VTune…VTune…VTune… Don't assume!
    - LOD all calculations: Physics, animation, AI, you name it!
  - In driver code:
    - Create all resources up front: textures, VBs, IBs, shaders
    - Reduce locking resources on the fly (discard VBs/IBs, don't write to a surface the GPU is reading from)
    - Create bigger batches: texture atlas, stitch strips together with degenerates
    - Vertex shader constants = lookup table for matrices
    - Instancing
  - Transferring data to GPU
    - Smallest vertex format possible
      - Remove unnecessary data
      - Use smallest data type possible
    - Derive attributes in vertex shader
    - 16 bit indices

Solutions to common bottlenecks

- IDX Bound, Vertex Shader Bound?
  - Reduce vertex attribute count
    - Compute some attributes
    - Combine attributes (2 2D tex coords per attribute)
  - Use geometry LOD
  - Move invariant calculations to the CPU
  - Use indexed primitives, more cache friendly
  - Don’t do unnecessary matrix multiplies
  - Use vertex shader branching to bypass expensive calculations
  - Use NVShaderPerf!

Solutions to common bottlenecks

- Pixel Shader Bound?
  - Render depth first (no color writes = 2X speed)
  - Prebake complex math into textures
  - Move per pixel calculations to the vertex shader
  - Use partial precision where possible, try it you may like the result
  - Avoid unnecessary normalizations
  - Use LOD specific pixel shaders
  - Use NVShaderPerf!

Solutions to common bottlenecks

- Texture bound?
  - Prefilter textures to reduce size
  - Mipmap on any texture/surface that might be minified
  - Compressed textures
  - Use float textures only when needed

Solutions to common bottlenecks

- Frame buffer bound?
  - Render depth first (no color writes = 2X speed)
  - Only use alpha blending when necessary
  - Use alpha test
  - Disable depth writes when possible
  - Avoid clearing the color buffer if you touch every pixel (but do clear Z)
  - Render front to back to get better z culling
  - Use float textures only when needed

NVShaderPerf

- What is NVShaderPerf?
- What's new with version 1.8?
- What's coming with version 2.0?

NVShaderPerf

Inputs:
- HLSL
- PS1.x,PS2.x,PS3.x
- VS1.x,VS2.x, VS3.x
- GLSL (fragments)
- !!FP1.0
- !!ARBfp1.0
- Cg

NVShaderPerf

Outputs:
- Resulting assembly code
- # of cycles
- # of temporary registers
- Pixel throughput
- Test all fp16 and all fp32

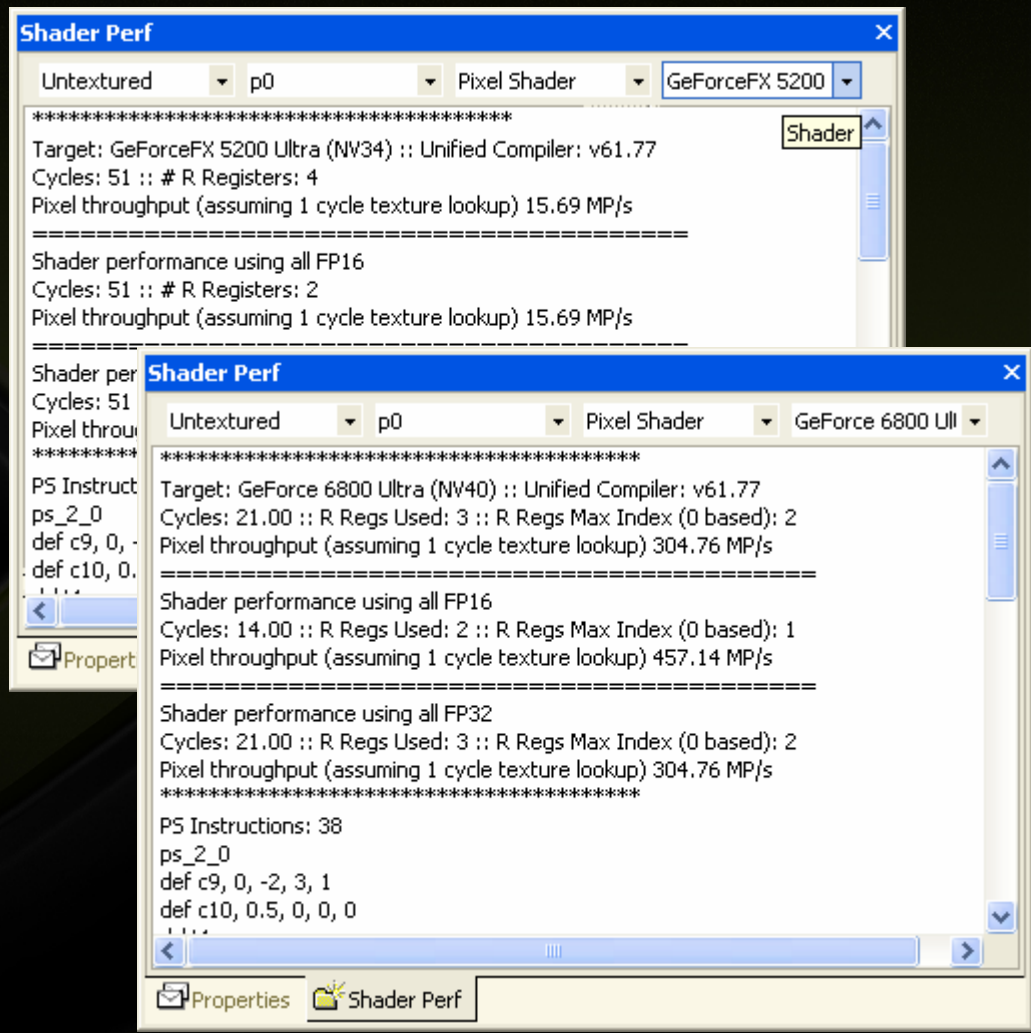
GPU Arch:
- GeForce 7X00
- GeForce 6X00
- Geforce FX series
- Quadro FX series

NVShaderPerf: In your pipeline

- Test current performance
  - against shader cycle budgets
  - test optimization opportunities
- Automated regression analysis
- Integrated in FX Composer 1.8

FX Composer 1.8 – Shader Perf

- Disassembly
- Target GPU
- Driver version match
- Number of Cycles
- Number of Registers
- Pixel Throughput
- Forces all fp16 and all fp32 (gives performance bounds)



NVShaderPerf 1.8

- Support for GeForce 7800 GTX and Quadro FX 4500
- Unified Compiler from ForceWare 8X.XX driver
- Better support for branching performance
  - Default computes maximum path through shader
  - Use –minbranch to compute minimum path

NVShaderPerf 1.8

```
////////////////////////////////////////////////////////////////////
 // determine where the iris is and update normals, and lighting parameters to simulate iris geometry
 ////////////////////////////////////////////////////////////////////

 float3 objCoord = objFlatCoord;
 float3 objBumpNormal = normalize( f3tex2D( g_eyeNermel, v2f.UVtex0 ) * 2.0 - float3( 1, 1, 1 ) );
 objBumpNormal = 0.350000 * objBumpNormal + ( 1 - 0.350000 ) * objFlatNormal;
 half3 diffuseCol = h3tex2D( g_irisWhiteMap, v2f.UVtex0 );
 float specExp = 20.0;
 half3 specularCol = h3tex2D( g_eyeSpecMap, v2f.UVtex0 ) * g_specAmount;

 float tea;

 float3 centerToSurfaceVec = objFlatNormal; // = normalize( v2f.objCoord )
 float firstDot = centerToSurfaceVec.y; // = dot( ce
 if( firstDot > 0.805000 )
 {
  // We hit the iris.  Do the math.

  // we start with a ray from the eye to the surface
  float3 ray_dir = normalize( v2f.objCoord - objEy
  float3 ray_origin = v2f.objCoord;

  // refract the ray before intersecting with the iris
  ray_dir = refract( ray_dir, objFlatNormal, g_refra

  // first, see if the refracted ray would leave the e
  float t_eyeballSurface = SphereIntersect( 16.0, r
  float3 objPosOnEyeBall = ray_origin + t_eyeball
  float3 centerToSurface2 = normalize( objPosOn

  if( centerToSurface2.y > 0.805000 )
  {
   // Display a blue color
   diffuseCol = float3( 0, 0, 0.7 );
   objBumpNormal = objFlatNormal;
   specularCol = float3( 0, 0, 0 );
   specExp = 10.0;
  }
  else
  {
   // transform into irisSphere space
   ray_origin.y -= 5.109000;

   // intersect with the Iris sphere
   float t = SphereIntersect( 9.650000, ray_origin, ray_dir );
   float3 SphereSpaceIntersectCoord = ray_origin + t * ray_dir;
   float3 irisNormal = normalize( -SphereSpaceIntersectCoord );
```

Eye Shader from Luna
Maximum branch takes 674 cycles
Minimum branch takes 193 cycles.

```
C:\WINDOWS\System32\cmd.exe

T:\tmp>t:\sw\devrel\sdk\tools\bin\release_pdb\nvshperf\nvshaderperf -a NV40 corn
ea2.txt
----------------------------------------------------------------------------

Running performance on file Cornea2.txt
--------------------- NV40 ---------------------
Target: GeForce 6800 Ultra (NV40) :: Unified Compiler: v77.72
Cycles: 674.25 :: R Regs Used: 12 :: R Regs Max Index (0 based): 11
Pixel throughput (assuming 1 cycle texture lookup) 9.50 MP/s

T:\tmp>t:\sw\devrel\sdk\tools\bin\release_pdb\nvshperf\nvshaderperf -minbranch -
a NV40 cornea2.txt
----------------------------------------------------------------------------

Running performance on file Cornea2.txt
--------------------- NV40 ---------------------
Target: GeForce 6800 Ultra (NV40) :: Unified Compiler: v77.72
Cycles: 192.82 :: R Regs Used: 12 :: R Regs Max Index (0 based): 11
Pixel throughput (assuming 1 cycle texture lookup) 33.33 MP/s

T:\tmp>_
```

NVShaderPerf – Version 2.0

- Vertex throughput
- GLSL vertex program
- Multiple driver versions from one NVShaderPerf
- Much smaller footprint
- New programmatic interface
- Integration into FX Composer 2.0
- What else do you need?

NVShaderPerf@nvidia.com

Questions?

- Developer tools DVDs available at our booth
  - NVPerfKit 2.0
  - NVPerfHUD 4.0 Overview Video
  - NVPerfHUD 4.0 Quick Reference Card
  - User Guides
- Online:
  - http://developer.nvidia.com/NVPerfKit
  - http://developer.nvidia.com/NVPerfHUD

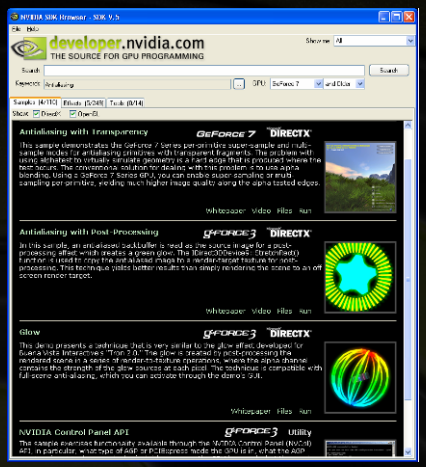
NVPerfKIT@nvidia.com
NVPerfHUD@nvidia.com
NVShaderPerf@nvidia.com
FXComposer@nvidia.com

NVIDIA Session Raffle

- GPU Gems 2 books
  - Also available at the GDC store
- 7900GT Board
  - Fast!
  - Cool, silent, 1-slot solution
- Didn’t win?
  - Swipe your card at the entrance to enter our post-GDC raffle!

NVIDIA SDK

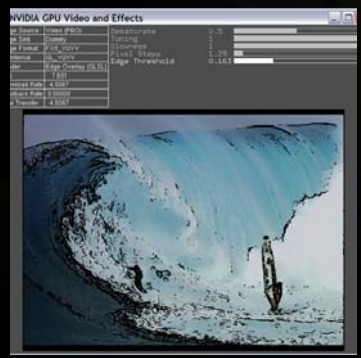
The Source for GPU Programming



Hundreds of code samples and effects that help you take advantage of the latest in graphics technology.

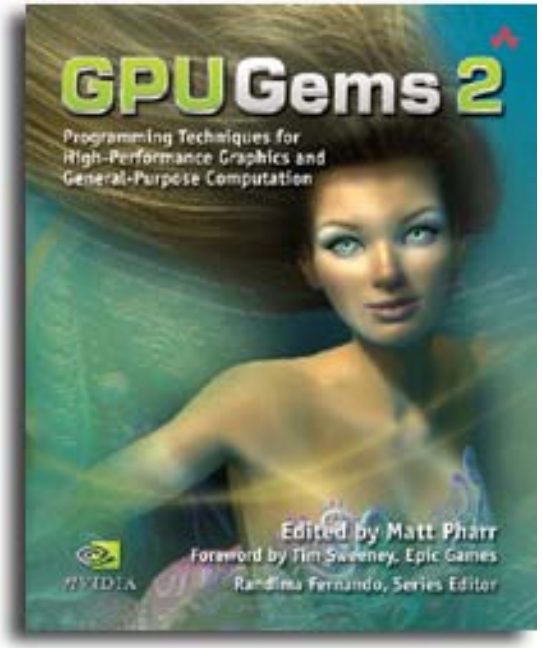
gamedeveloper FRONTLINE AWARD '04

- Tons of updated and all-new DirectX and OpenGL code samples with full source code and helpful whitepapers:
  - **Transparency AA, GPU Cloth, Geometry Instancing, Rainbow Fogbow, 2xFP16 HRD, Perspective Shadow Maps, Texture Atlas Utility, ...**
- Hundreds of effects, complete with custom geometry, animation and more:
  - **Shadows, PCSS, Skin, Plastics, Flame/Fire, Glow, Image Filters, HLSL Debugging Techniques, Texture BRDFs, Texture Displacements, HDR Tonemapping, and even a simple Ray Tracer!**



GPU Gems 2

Programming Techniques for High-Performance Graphics and General-Purpose Computation



- **880 full-color pages**
- **330 figures**
- **Hard cover**
- **$59.99**
- **Experts from universities and industry**

Graphics Programming

- **Geometric Complexity**
- **Shading, Lighting, and Shadows**
- **High-Quality Rendering**

GPGPU Programming

- **General Purpose Computation on GPUs: A Primer**
- **Image-Oriented Computing**
- **Simulation and Numerical Algorithms**