Left 4 Dead 2 Linux - From 6 to 300 FPS in OpenGL *

* or How We Learned to Stop Worrying and Love OpenGL

Rich Geldreich, Rick Johnson,
Mike Sartain

Introduction

- The Steam team took the initiative and bootstrapped the Linux version of Left 4 Dead 2
- Recently formed Linux team has focused on improving performance
- Collaborative effort between Valve and engineers from NVIDIA, AMD, and Intel
  - Valve + Driver Devs together in same room
- This quick talk focuses on the NVIDIA GTX 680 with driver multithreading enabled
  - Our performance is currently highest on NVIDIA's GL driver

short demo

- “short” timedemo created by Rick Johnson at Valve
- Reproducible workload, representative of actual gameplay, used for all of our recent experiments
  - Fairly deterministic – dead zombie limb positions seem to vary somewhat between runs
- Total Frames (excluding loading screen): 1497
- Total Batches: 837,044
- Total Primitives: 940,994,686

State of Source Engine OpenGL Support

- Avg. 11% faster in GL vs. D3D9 on GTX 680
  - ~5% higher performance should be achievable by reducing overhead in our D3D->GL layer
- Targets a D3D9-like API with extensions, translates to GL API calls on the fly, supports shader model 2.0b, soon 3.0
  - Mostly 1:1 mapping between D3D and GL concepts
- Non-deferring, locally optimizing translation layer
  - Calls to `DrawIndexedPrimitive` immediately result in a state flush and call to `glDrawRangeElementsBaseVertex`
- Reasonable D3D->GL translation overhead
  - Multithreaded drivers: Avg. 50/50 split between CPU cycles spent calling GL vs. translation overhead
  - Single threaded drivers: 80% GL vs. 20% translation overhead

Optimization Effort

- Linux Team started with little practical OpenGL experience, so we needed help
- We shared builds and invited all vendors to our offices
- Focused on why the D3D9 vs. OpenGL builds had such vastly different performance on the same hardware
- Process: Devise/conduct experiment, test results with known workload, refine/update mental model of system’s behavior, repeat
- Goal was to account for every microsecond spent in our D3D->GL translation layer and render thread
- Interpreting experimental results can be challenging:
  - Game is multithreaded – bottlenecks can shift around in unintuitive ways
  - Driver’s server thread is mostly invisible to our profiling tools
  - Source Engine is extremely configurable/scalable – easily misconfigured
- Primary profiling/debugging tools used:
  - Telemetry: Profiling/visualization system from RAD Game Tools
  - Custom batch trace recording mode in our D3D and D3D->GL translation layers
  - AMD’s GPU PerfStudio for GL state debugging/API call tracing

Telemetry

- Cross platform performance visualization system from RAD Game Tools
- Three components:
  - Visualizer app (Linux/OSX/Windows)
  - Run-time component – trivial to drop in, very low footprint
  - Server
- Intrusive system – requires adding calls to the Telemetry API
  - We use Telemetry zones for CPU profiling, timespans for GPU
- We added several modes useful for graphics debugging/profiling:
  - Telemetry zones generated for *all* GL calls
  - Plumbed renderer’s named begin/end “PIX” events to Telemetry zones
  - GPU timestamp queries visualized as Telemetry timespans
- We’ve really only scratched the surface of its capabilities

CPU Profiling with the Telemetry Zone API

- Zones define an “area of execution” on a single thread
  - Usually corresponds to a single function – but not always
- Zones are hierarchical, i.e. if you define a zone inside of another it will nest naturally in the Visualizer
- Example code:

```
void some_function( int x )
{
  tmEnter( g_tm_context, TMZF_NONE, "some_function: %i”, x );
  do_work();
  cleanup();
  tmLeave( g_tm_context );
}
```

Telemetry Visualizer 64-bit

Session | Timeline | Zones | Messages | Reports | Options | License/About | Help

localhost:9180

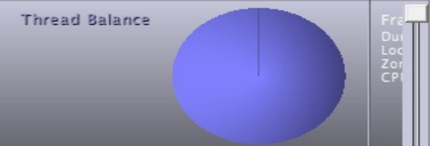
-all +all

Frame: 485 [273.39ms] 100.000ms 200.000ms 00:29.517

Summary

Frame 485
Duration: 273.39 ms
Location: 00:29.517
Zones: 15717
CPU Speed: 3340.914 MHz

Thread Balance



<main thread>

CEngine::Frame (273.31ms)
Frame (273.31ms)
_Host_RunFrame (273.30ms)
_Host_RunFrame_Render (258.86ms)
_Host_RunFrame_Render - UpdateScreen (258.85ms)
SCR_UpdateScreen (258.85ms)
View_Render (256.63ms)
CViewRender::Render (256.62ms)
Render (256.62ms)
RenderView (255.78ms)
ViewDrawScene (234.71ms) | DoEnginePostProces
DrawWorldAndEntities (228.50ms) | Generate8BitBloom
bCheapWater (228.50ms) | Downsamp
CClientSha | CModelRenderSystem::DrawModels (67.12ms) | DrawModelArray (111.99ms) | DrawModelArray (18.6 | SetRender
NORMAL 40 (66.67ms) | DrawModelArray 1 (111.99ms) | DrawModelArray 1 (18.
DrawModelArray 40 (66.67ms) | DrawModelArray 2 (111.99ms) | DrawModelArray 2 (18.
DrawModelArray2 (66.67ms) | DrawMeshRenderData 2 (49.15ms) | DrawModelArrayFlashlight (62.68ms) | DrawMeshRenderData
DrawModelArrayFlashlight (61.56ms) | DrawInstances (26.59ms) | DrawInstances (22.56ms) | DrawMeshRenderData 2 (62.67ms)
DrawMeshRenderData (61.56ms) | DrawInstances (26.58ms) | DrawInstances (22.56ms) | DrawInstances (28.15ms) | DrawInstances (34.51ms)
DrawIndexedPrimitive (26.55ms) | DrawInstances (28.15ms) | DrawInstances (34.51ms)
DrawRangeElements (26.53ms) | DrawIndexedPrimitive (28.09ms) | DrawIndexedPrimitive (34.46ms)
DrawRangeElements (28.05ms) | DrawRangeElements (34.44ms)
DrawRangeElements 0-1673 count:7 | DrawRangeElements 0-834 count:3867 mode

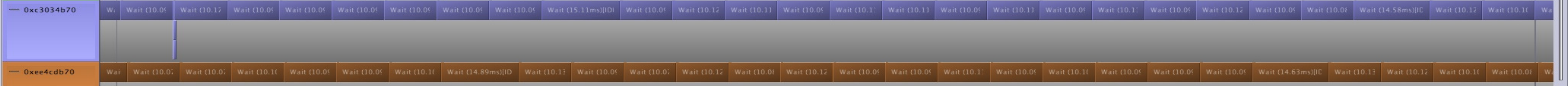
0xc3034b70

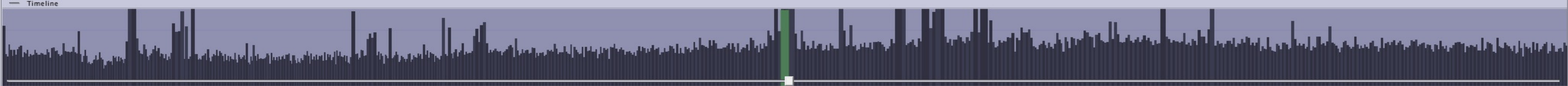
Wait (15.11ms)

0xee4cdb70

Wait (14.89ms) | Wait (14.63ms)

Timeline



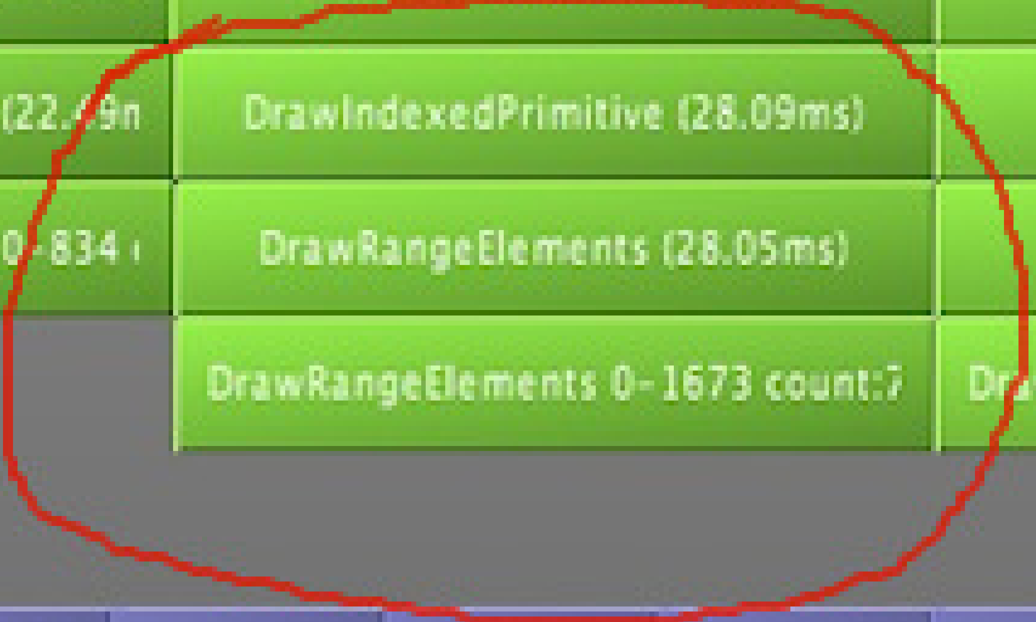


l4d2

valve-System-Product-Name (x86/Linux - 12 cores 32-bits)
/home/valve/valvesrc/console/l4d2/game/hl2_linux
Mar 09, 2012 6:43:47 PM

Zone Cutoff: 0.01 ms
Max Depth: 32
Show Idle Zones:

Frames: 859
Duration: 00:59.661
Buffer: 32768K

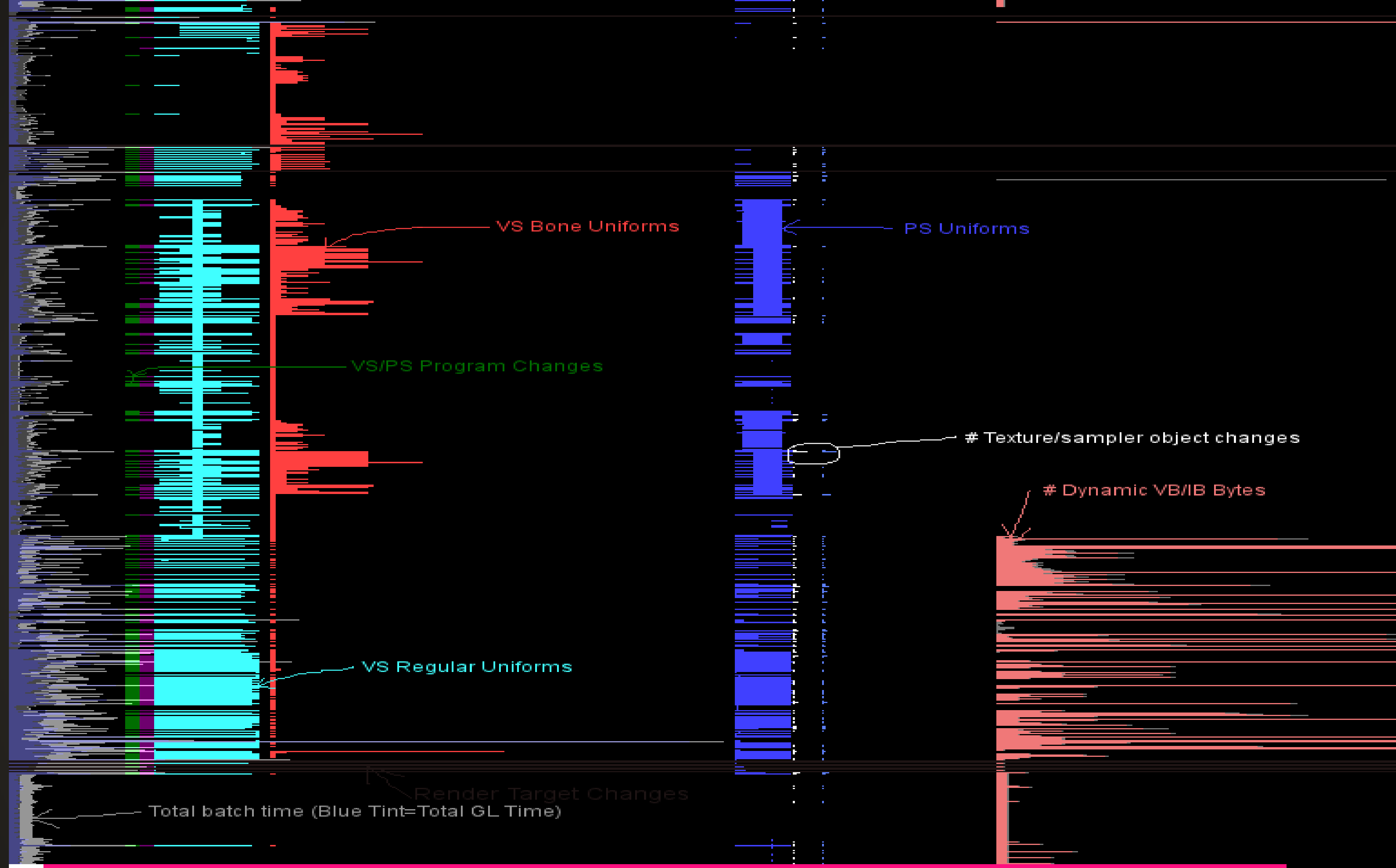

DrawInstances (26.59ms)
DrawInstances (22.56ms)
DrawMeshRenderData 2 (62.67ms)
DrawInstances (26.58ms)
DrawInstances (22.56ms)
DrawInstances (28.15ms)
DrawInstances (34.51ms)
DrawIndexedPrimitive (26.55ms)
DrawInstances (28.15ms)
DrawInstances (34.51ms)
DrawRangeElements (26.53ms)
DrawIndexedPrimitive (28.09ms)
DrawIndexedPrimitive (34.46ms)
DrawRangeElements (28.05ms)
DrawRangeElements (34.44ms)
DrawRangeElements 0-834 count:3867 mode

GPU Profiling with the Telemetry Timespan API

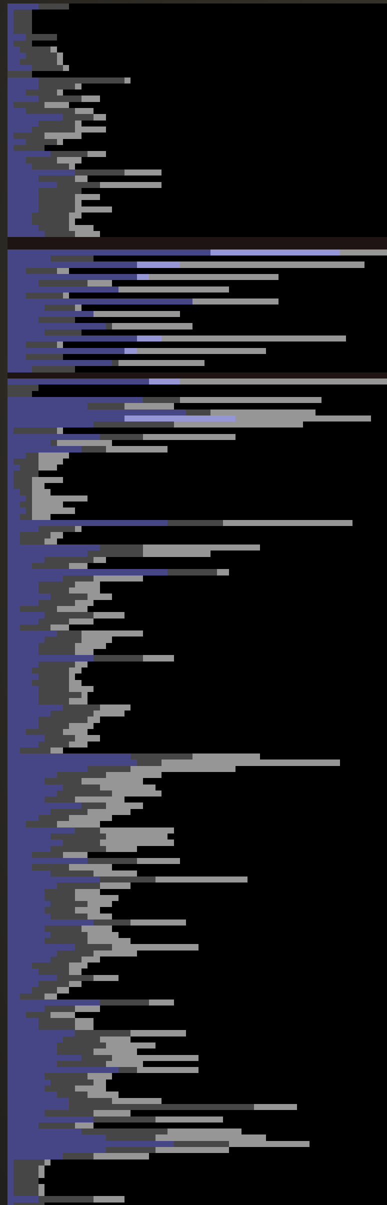
- Timespans are like zones, but are not nestable, are independent of zone hierarchies, and may span multiple frames
- GPU activity is profiled by generating **`GL_TIMESTAMP`** queries, using the **`GL_ARB_timer_query`** extension API's
- **`glQueryCounter`** is called to retrieve the begin/end timestamp of zones marked for GPU profiling
  - GPU time stamps are converted to absolute time and fed to the Telemetry timespan API
- The Telemetry SDK has several examples on how to do this with various rendering API's

Batch Tracing

- Visualizes the D3D/GL state changed in each batch, and the CPU time spent processing each batch
  - Timings: Total GL time, Total `DrawIndexedPrimitive` time, Total D3D API time
- X axis = time, full scanline width = 50us
- Y axis = batches, 1 batch per scanline, topmost scanline = first batch, Present() time is visualized at bottom
- Major state changes visualized as colorized columns
  - Easy to visually correlate state changes with increased GL processing time
  - Easy to visualize overhead of translation layer vs. GL, spikes, lock time
  - Easy to compare D3D9 vs. OpenGL performance – just compare traces
- Trace videos created in real-time (1 PNG/frame using miniz open source library)
- VirtualDub used to create batch trace videos from multiple PNG's
- Can easily share videos with vendors using YouTube


VS Bone Uniforms
PS Uniforms
VS/PS Program Changes
Texture/sampler object changes
Dynamic VB/IB Bytes
VS Regular Uniforms
Render Target Changes
Total batch time (Blue Tint=Total GL Time)
Total Present() Time
OpenGL Frame: 9811, Batches+Clears: 729, Prims: 830859, Program Changes: 215
Frame: D3D Calls: 11061, D3D Time: 1.186ms
Frame: GL Calls: 7493, GL Time: 0.538ms
Overall: Batches: 745420, Prims: 834651176, Program Changes: 209686
Overall: D3D Calls: 10443427 D3D Time: 1444.364ms
Overall: GL Calls: 7316155 GL Time: 745.102ms
50us

Batch Tracing – Time Visualization



Batch Timing Legend

- Total time spent calling OpenGL
- Total time spent calling `DrawIndexedPrimitive`
- Overall time spent processing the batch (includes all DrawIndexedPrimitive+OpenGL time)

Summary of Optimizations We’ve Done So Far

- Multithreading: Now enabled in GL mode, removed most calls to `glXMakeCurrent`, pthreads usage fixes
- Reduced translation overhead: Rewrote hottest D3D->GL code paths for higher performance
- Uniform updating: Improved dirty range tracking, added separate uniform array for bone matrices
- Dynamic buffer updating: glMapBufferRange vs. glBufferSubData, handling of lock DISCARD and NOOVERWRITE, added D3D UnlockActualSize API
- gcc compiler options: added -ffast-math, removed –fPIC

Resources/Links

- Tools to check out:
  - Telemetry: http://ww.radgametools.com/telemetry.htm
  - GPU PerfStudio: http://developer.amd.com/tools/PerfStudio
- Valve Linux Blog: http://blogs.valvesoftware.com/linux/
  - We'll be releasing several blog posts with lots of technical details over the next couple months or so
- Valve is hiring!
  - http://www.valvesoftware.com/jobs/
  - Our team is looking for Linux kernel, driver, and OpenGL developers
- We'll be at the OpenGL party to answer any questions