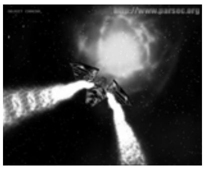
Real-Time Rendering (Echtzeitgraphik)



Dr. Michael Wimmer
wimmer@cg.tuwien.ac.at

Real-Time Rendering Performance Analysis and Characterization

What for?

- If you want to improve performance…
  - ... you have to be able to analyze it!
- Peek at what other people are doing!
- Understand influence of scene design
- Understand influence of hardware
- Will include some optimization tips…

Overview

- Performance Analysis
  - Which tools to measure performance?
- Performance Characterization 1
  - Characterize general properties of *scenes* and *hardware architectures*
- Performance Characterization 2
  - Characterize and find *bottlenecks*
- Optimization
  - Will mostly be result of the above

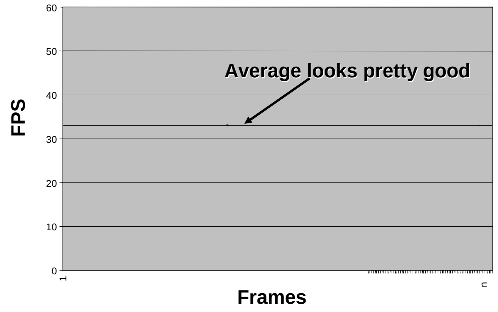
Analysis Tools

- Framerate logging
  - DIY (do it yourself), FRAPS
- Call tracing/logging
  - GLTrace
- External profilers
  - VTune, Quantify
- Internal profiling (fine-grained)
  - RDTSC
- Driver profiling
  - Only available in Direct3D for now…

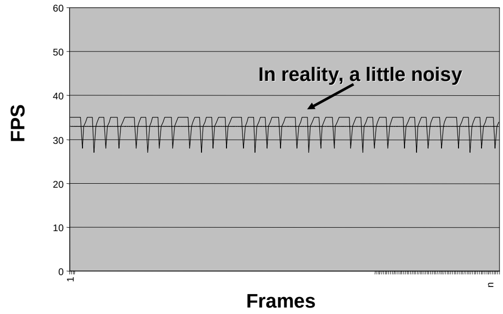
Frame Rate Calculation

- Running average
  - Great for a quick look
  - Obscures spikes over a few frames
- Per frame FPS calculation
  - “Instantaneous FPS”
  - High accuracy
  - Lots of data
  - Graph it out on top of your app
  - Log it to a file

Original Frame Rate

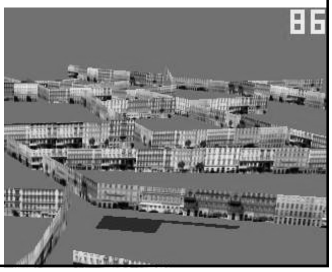


Instantaneous Frame Rate



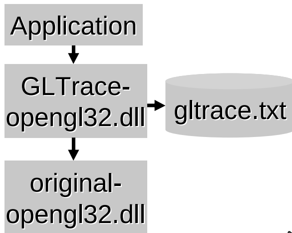
FRAPS

- Displays frame rate for **any** OpenGL app
  - by intercepting calls to opengl32.dll
- Average over last few frames
- Has file logging
- Small performance hit
- Good for quick comparisons
- `www.fraps.com`



GLTrace

- Can log **all** OpenGL calls for any app
- Gives call counts
- Allows reverse engineering (also of models!)
- Cheating… (wireframe)
- See VU-page for link…
- Can use trace for simulation!



Example Trace (1338 Frames)

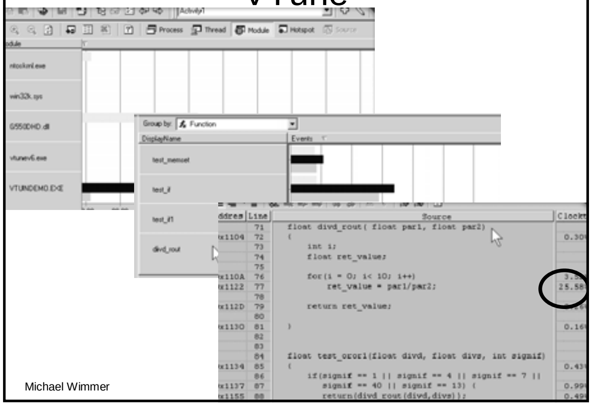
Vertices	4326.8
Triangles (3D)	2535.3
Triangles (2D)	939.0
Fragments	1353892
Image	1024×768

```
738541   glVertex3fv
728673   glTexCoord2fv
224682   glColor4fv
206474   glNormal3fv
201074   glCallList
180574   glBegin
180574   glEnd
168356   glBindTextureEXT
22659    glEnable
21150    glMaterialfv
20557    glDisable
9622     glShadeModel
5706     glPopMatrix
5706     glPushMatrix
4216     glBlendFunc
3478     glMatrixMode
3164     glLoadIdentity
3010     glDepthMask
2546     glAlphaFunc
2546     glMultMatrixf
2105     glTexEnvf
1676     glEndList
1676     glNewList
```


External Profiling – Sampling

- Based on sampling at regular intervals
- Example: Intel VTune
  - Expensive, only Intel processors
- How much time is spent in…
  - OS
  - Other applications
  - Driver (kernel- and user-mode)
  - Application (which function, which line of code)
- Pros
  - works with any program, no rebuild necessary
  - no slowdowns

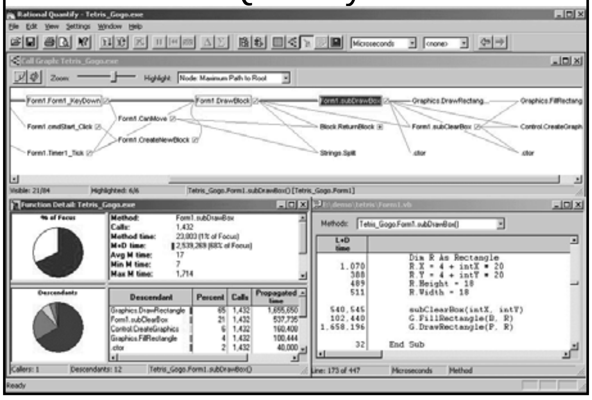
VTune



External Profiling – Instrumentation

- Inserts logging directly into code
- Example: Rational Quantify
- Pros
  - Very accurate
  - True call list and call graph
- Cons
  - Need to rebuild code
  - Really slows down execution
  - So slow, it invalidates all off-CPU interaction
    - Example: main memory, GPU

Quantify



Internal Profiling – RDTSC

- Current clock cycle counter
- Fine-grained timing (microseconds)
- Calibrate using **`GetTickCount()`**
- Take into account overhead of rdtsc itself!
- Warm up caches (for tight loops)

```
LARGE_INTEGER val; // 64-bit integer defined in Win32
__asm
{
    cpuid          // for serialization of out-of-order instructions
    rdtsc
    mov val.LowPart,  eax
    mov val.HighPart, edx
}
```


Profiling – Multitasking effects

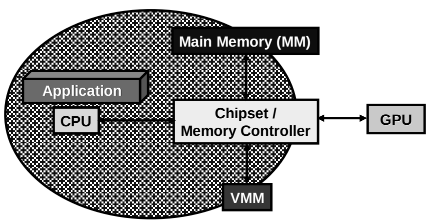
- Be aware of multitasking! Win2K examples:
  - Clock tick every 10 ms → scheduler called
  - Thread quantum ~60 ms for foreground apps
  - > 1000 interrupts per clock tick!
  - Accuracy not better than 1 ms for longer runs
- Consider using higher priority for timing

```
SetPriorityClass(hProcess,
  REALTIME_PRIORITY_CLASS);
SetThreadPriority(hThread,
  THREAD_PRIORITY_TIME_CRITICAL);
```

  - Beware thread starvation!

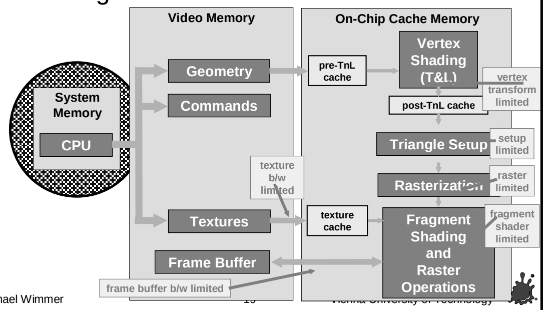
Profiling: Seeing Half the Picture

- Profiler runs on the CPU
- GPU is a black box



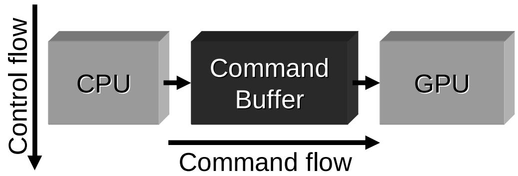
Profiling: Seeing Half the Picture

- GPU is a black box
- How to guess hidden bottlenecks?



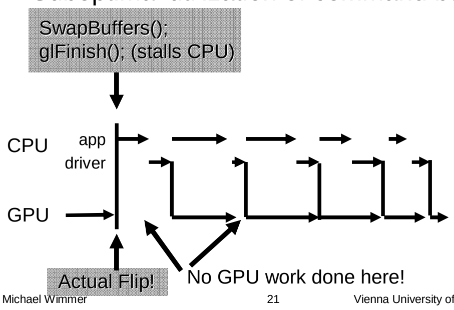
Profiling Graphics Calls

- RDTSC works reasonably for CPU
  - With multitasking caveats
- Not so for graphics calls (GPU)
- CPU and GPU run in parallel
- Commands are buffered for GPU



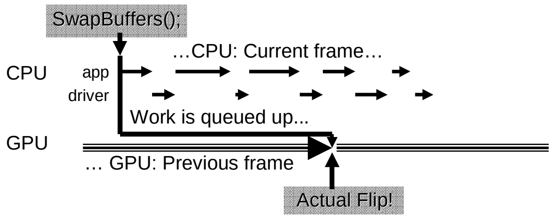
Command Buffering

- Synchronized rendering
- Suboptimal utilization of command buffer



Command Buffering

- Asynchronous rendering
- Great for load balancing
- Can introduce latency



Profiling Graphics Calls

Case 1: command buffer not full

- RDTSC will measure CPU stuff
  - unpack command and parameters
  - prepare for GPU
  - maybe texture transfers
  - maybe vertex transfers (driver decides on buffering)
  - queue command

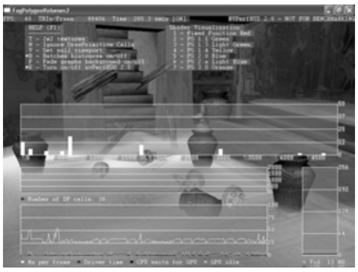
Profiling Graphics Calls

Case 2: command buffer full (GPU busy)

- Example: render many large triangles stored in vertex buffer on card
- RDTSC will measure…
  - same CPU stuff as before
  - PLUS additional wait time for GPU
- Conclusion:
  - Both are useless!
  - Profiling graphics calls is almost impossible
  - Use glFinish() to empty command buffer

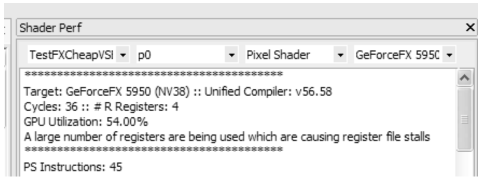
Driver Profiling

- NVPerfHud (only Direct3D)
- Information about driver internals
  - Batch sizes
  - Wait times
- Bottleneck identification



Driver Profiling

- FxComposer
- Internal information about pixel shaders
  - Cycle count



Performance Characterization 1

- Performance tuning = finding bottlenecks
- First, need to understand characteristics of scene (as related to hardware)

- Fragment formula
- Depth complexity
- Design strategies

Fragment Formula

- Relates geometry and fragment processing

$$a = \frac{F}{T}$$

- Parameters:
  F = number of fragments
  T = number of triangles
  a = number of fragments per triangle

Fragment Formula – Meaning

- Different meanings for scenes and hardware
- Scenes
  - Characterizes triangle distribution in scene
  - *a* = average triangle size
- Hardware
  - Typical SGI performance figure: "*T* *a*-pixel triangles per second"
  - *a* = optimal triangle size
  - *F*, *T* are rates ("per second")
  - Per-frame and per-second related by fps

$$a = \frac{F}{T}$$

Triangle Area Implications

- Triangle with *a* pixels is a balance point between:
  - Geometry computations per triangle
  - Fragment pipeline fill capacity
- Triangles larger than *a*:
  - are fill limited (dominated), rate less than *T*
- Triangles smaller than *a*:
  - are geometry limited, rate no faster than *T*

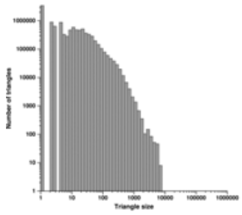
Triangle Area Distribution

Deering Study

- Scenes: Triangle distribution roughly exponential towards smaller triangles
  - Already for individual objects with LOD
  - Even stronger for whole scenes!
- Hardware: historical development
  - For SGI, *a* went from ~1000 to ~50
  - For NVidia hardware, *a* was typically 8 (assuming 4-sample AA)
  - Today: depends on specific vertex/fragment program complexity!

Deering Study

- Triangle distribution for architectural scene
  - roughly a power function (see log/log plot)



Triangle Area Distribution Caveats

- Small and large triangles in the same scene!
- Triangles are geometry/fill limited, not scenes!!!
- Even if app is fill limited overall, increasing geometric detail will slow it down
- Even if app is geometry limited overall, increasing pixel complexity will slow it down
- Except if triangle areas are roughly equal!

Triangle Area Caveats

- Don’t trust vendor-quoted triangle rates
- Typically only achieved under optimal conditions
  - E.g., large batch sizes (>200 triangles)
- However, will see how to get near

Depth Complexity

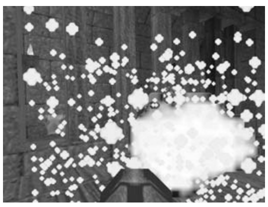
- Typical scene characterization figure:

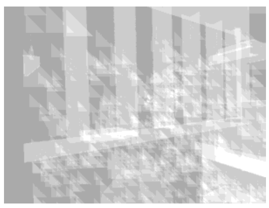
$$d = \frac{F}{I}$$

- Parameters:
  - I = number of image pixels
  - d = depth complexity (or “overdraw”)

Depth Complexity

- Measure using stencil buffer
  - **`glStencilOp(GL_KEEP, GL_INCR, GL_INCR);`**





Z-Buffer Reads and Writes

- Read-Modify-Write cycle – potentially slow

```
if (f.z < z[f.x][f.y])
{
    color[f.x][f.y] = blend(f);
    z[f.x][f.y] = z;
}
```

- Expected number of writes?
  - 1 + 1/2 + 1/3 + 1/4 + … + 1/d
  - Harmonic numbers; O(log(n))
  - Homework assignment (combinatorial problem)

Z-Buffer Reads and Writes

- Important for fillrate
  - Read-only is faster than read-modify-write
- Even more so with “Deferred Shading”
  - Pixel shading after z-test
  - ATI, NVidia call this “Early Z” or “Occlusion Test”
- Different cases for d = 4:
  - Best case: 1 overwrite
  - Worst case: 4 (=d) overwrites
  - Expected case for random order: 2 overwrites

→ Sorting by depth is important for new cards!

Design Space

- Triangle area vs. depth complexity

$$a = \frac{F}{T} \longrightarrow F = aT = dI \longleftarrow d = \frac{F}{I}$$

- Parameters:
  - T = Number of triangles
  - a = Average area of a triangle
  - F = Number of fragments
  - I = Image size
  - d = Depth complexity

Designing an 80 Million Triangle Scene

- Assume movie quality image
  - I = 4K by 2.5K = 10 MP
  - F = d I = 4 x 10 MP = 40 MF
- Assume maximum geometric detail
  - a = 0.5 F/T (Nyquist limit)
  - → T = 40 MF / 0.5 = 80 MT
- Scaling up to 60 Hz:
  - 60 I/s * 80 MT/I = **4.8 Billion triangles/s**
  - 60 I/s * 40 MF/I = **2.4 Billion fragments/s**
- Not quite there yet…

Design Strategies

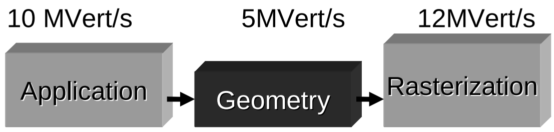
- Previous example assumes:
  - Culling limits d to 4 (visibility, occlusion)
  - Level of detail removes really small triangles
- More realistic scene design:
  - Do Culling and LOD
  - Hardware determines average triangle area!
- Very difficult to achieve peak triangle and fill rate simultaneously!

Performance Characterization 2

- Performance tuning = finding bottlenecks
  - (for pipelined architectures)
- Need to understand characteristics of rendering pipeline
- Bottlenecks
- Bottleneck identification

What Is a Bottleneck?

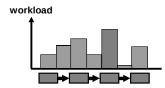
- Recall: rendering pipeline
- As fast as slowest unit → bottleneck!
- Example: total throughput is only 5 million vertices/s!



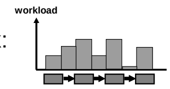
→ Geometry stage is bottleneck!

Locating and Eliminating Bottlenecks

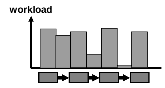
- Location: For each stage
  - Vary workload (or remove)
    - Measure performance impact
  - Clock down
    - Measure performance impact



- Elimination:
  - Decrease workload of bottleneck:



  - Increase workload of non-bottleneck stages:



Common Bottlenecks

A graphical application can be (one or all of)

- Application-limited
  - Almost all applications
  - AI, collision detection, vertex copies, …
- Fill- (Rasterization-)limited
  - Today's games in high resolutions
- Geometry- (Transformation-)limited
  - Typical for scientific applications: polygons used "as is" or generated automatically

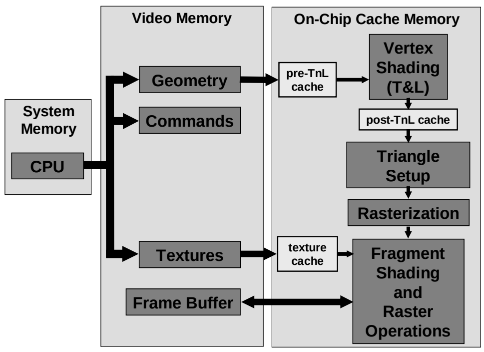
Bottleneck Analysis

- Iterative optimization process
  - New bottlenecks appear when removing old ones
  - Don't trust performance increase: 20% increase here could include 10% decrease elsewhere
- Remember: bottlenecks shift
  - Can be both geometry and fill limited in the same frame
  - Need to do bottleneck analysis for different parts of scene (scene decomposition)

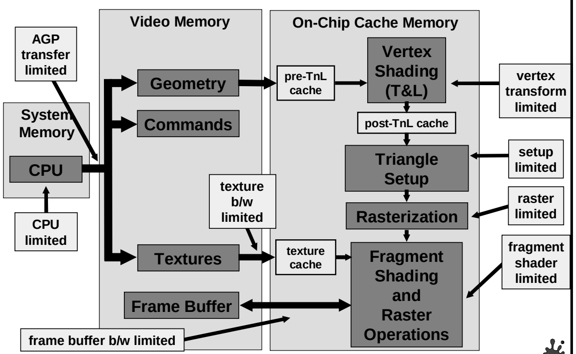
A Glimpse at PC Architecture

- API calls write to buffers (commands and data)
- Buffers pulled by DMA from GPU
- Vertex data in indexed arrays
  - AGP or video memory
  - Efficient pull of data
  - Post-TnL vertex cache eliminates redundant vertex transfers and transforms
- Conclusion: include memory transfers in bottleneck considerations!

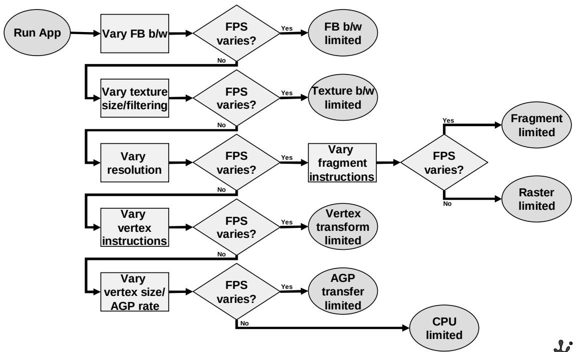
A Glimpse at PC Architecture



Potential Bottlenecks

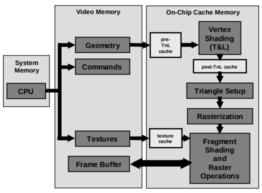


Bottleneck Identification



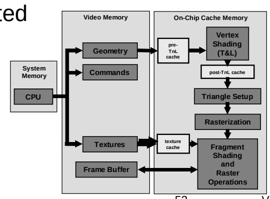
Frame Buffer B/W Limited

- Vary all render target color depths (16-bit vs. 32-bit)
  - If frame rate varies, application is frame buffer b/w limited



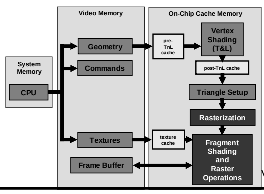
Texture B/W Limited

- Otherwise, vary texture sizes or texture filtering
  - Force MIPMAP LOD Bias to +10
  - Point filtering versus bilinear versus tri-linear
  - If frame rate varies, application is texture b/w limited



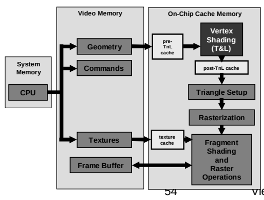
Fragment or Raster Limited

- Otherwise, vary all render target resolutions
  - If frame rate varies, vary number of instructions of your fragment programs (for newer HW)
    - If frame rate varies, application is fragment shader limited
    - Otherwise, application is raster limited



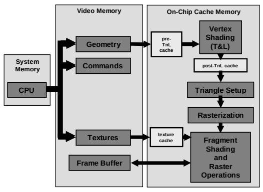
Vertex Transform Limited

- Otherwise, vary the number of instructions of your vertex programs (turn on/off lighting, texture transform for fixed function)
  - If frame rate varies, application is vertex transform limited



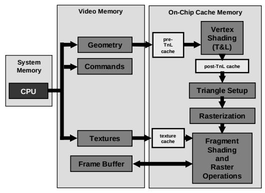
AGP Transfer Limited

- Otherwise, vary vertex format size or AGP transfer rate (for geometry in AGP memory)
  - If frame rate varies, application is AGP transfer limited



CPU Limited

- Otherwise, application is CPU limited
- Replace all OpenGL calls with dummy calls
  - If frame rate varies, app is driver limited
  - Otherwise, app is application limited



Bottleneck Identification

- NULL 3D caveat:
  - Speedup may also come from missing parallelism
- Testing parallelism
  - Null 3D
    - Absolute **best** case
  - Serialization
    - Insert glFinish() at several points
    - No more parallel execution
    - Absolute **worst** case

Bottleneck Identification Shortcuts

- Run identical GPUs on different speed CPUs
  - If frame rate varies, application is CPU limited
- Underclock your GPU
  - If slower core clock affects performance, application is vertex-transform, raster, or fragment-shader limited
  - If slower memory clock affects performance, application is texture or frame-buffer b/w limited

Optimization

- Always after bottleneck analysis
- Eliminate bottlenecks by
  - Making more efficient use of resources
    - Untapped GPU capabilities
    - Optimized memory transfers
  - Changing scene properties
- Will look at some optimization tricks for modern GPUs

Use Efficient API Calls

- Don’t:
  - glBegin()/glEnd() for geometry
  - Simple vertex arrays
  - glTexImage2D() for each frame
- Do:
  - Vertex buffer objects (recent ARB extension)
    - Allows storing geometry in AGP/Video mem
  - Index buffers
    - Drawing a complex object: only a single call!
  - Texture objects

Batching

- GPUs require large batches
  - Large driver overhead for each vertex buffer/array!
- ~50k glDrawTriangles/DrawIndexedPrimitive calls/s COMPLETELY saturate 1.5GHz Pentium 4
  - At 50fps this means 1k buffers/frame!
- Use thousands of vertices per vertex buffer/array
- Use thousands of triangles per call as possible
  - Use degenerate triangles to join strips together
  - Or: NV_restart_primitive extensions (send -1 for new strip)
  - Or don't use strip, but vertex cache

Indexing, Sorting

- Use indexed primitives (strips or lists)
  - Only way to use the pre- and post-TnL cache!
  - Not useful in some cases (leaves of a tree)
- Re-order vertices to be sequential in use
  - To maximize pre-TnL cache usage!
- (Approximately) sort front to back
  - Exploits early occlusion tests
- Sort per texture, shader and render state
- Avoid pipeline stalls (glReadPixels, …)
  - Exploit parallelism!

CPU Bottlenecks

- Application limited
  - AI, collision detection, network, file I/O
  - Graphics should be negligible!
    - Use brute-force GPU algorithms
    - Avoid smart algorithms to reduce load
- Driver/API limited
  - Too many OpenGL calls
  - Unoptimized driver paths (no "fast path")
  - Small batches
  - Driver should spend most time idling (VTune)

AGP Transfer Bottlenecks

- Unlikely…
- Use 16 bit indices
- Eliminate unused vertex attributes (e.g., color when normals are specified)
- Eliminate dynamic vertices
  - Use vertex shaders for animation instead!
- Use the right API calls (VBO = vertex buffer object)
  - Prefer static (write once) buffers)
- Vertex size should be multiples of 32 bytes

Vertex Transform Bottleneck

- Unlikely (usually, bottleneck is before!)
- Eliminate expensive lights
- Reorder vertices for cache, use NVTriStrip

Fragment Bottleneck

- Fragment shader too long
- Move per-fragment to per-vertex
- Use rough front-to-back order
  - Or even a z-only pass

Texture Bottlenecks

- Use texture compression and 16-bit maps
- Use mipmaps (help cache locality)
- Beware dependent texture lookups
- Anisotropic/trilinear filtering is slower

Hardware Fast Paths

- Fast buffer clears
  - But: need to clear stencil and depth at the same time, or turn off stencil
- Lots of other issues
- …

High-Level Optimizations

- Visibility culling
  - Don’t draw what you don’t see
- Levels of detail
  - Draw only as complex as necessary
- Image-based rendering
  - Replace geometry with images