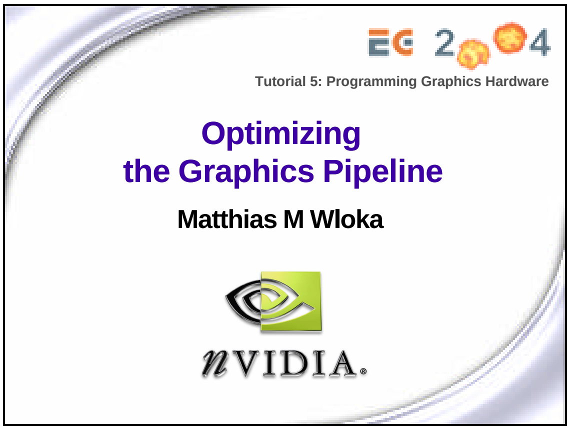
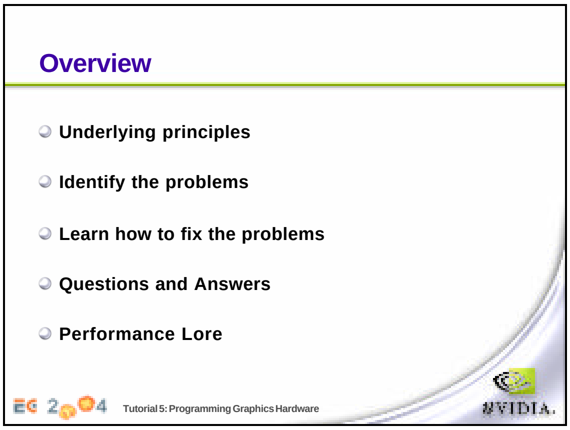


Tutorial 5: Programming Graphics Hardware

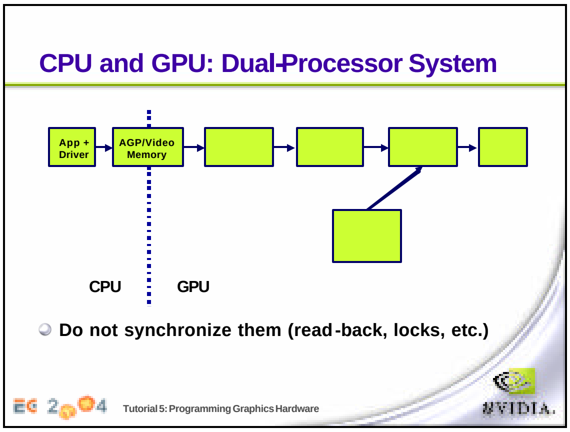
Optimizing the Graphics Pipeline

Matthias M Wloka



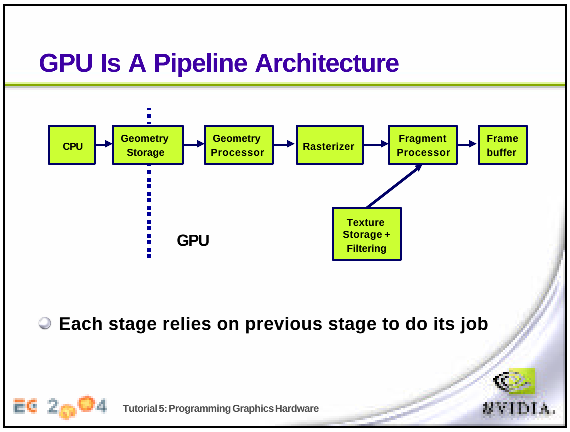
Overview

- Underlying principles
- Identify the problems
- Learn how to fix the problems
- Questions and Answers
- Performance Lore



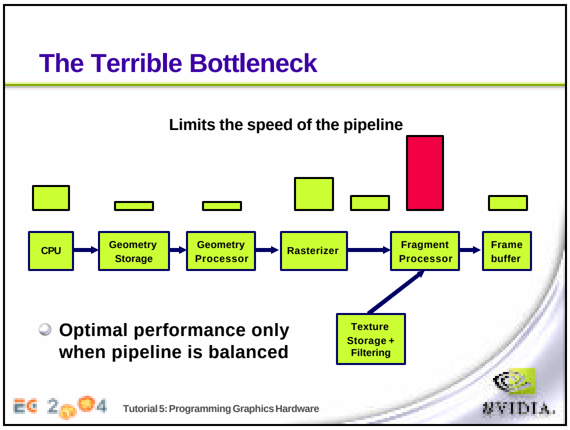
CPU and GPU: Dual-Processor System

- Do not synchronize them (read-back, locks, etc.)



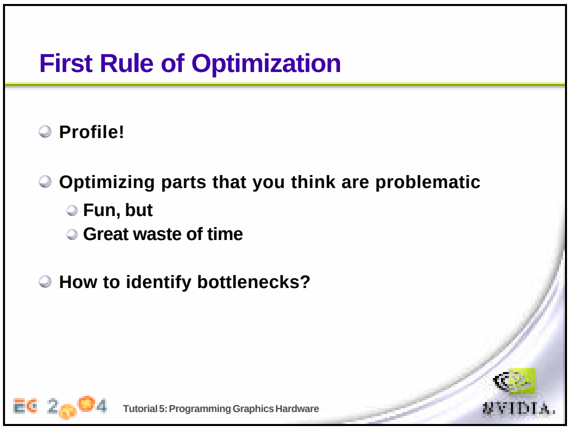
GPU Is A Pipeline Architecture

- Each stage relies on previous stage to do its job



The Terrible Bottleneck

- Optimal performance only when pipeline is balanced



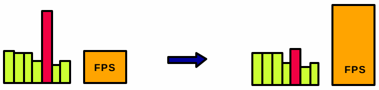
First Rule of Optimization

- Profile!
- Optimizing parts that you think are problematic
  - Fun, but
  - Great waste of time
- How to identify bottlenecks?

Bottleneck Identification

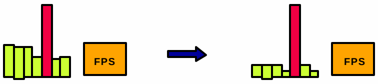
- Tools
  - nvPerfHUD
  - nvShaderPerf
  - CPU profilers (e.g., AMD CodeAnalyst)
  - Under-clock various domains (CPU, FSB, AGP, GPU)
- Modify workload of stages:
  - Modify suspected bottleneck stage itself
  - Rule out all other stages

Modify Suspected Bottleneck Stage



- If performance changes proportionally, you found the bottleneck
- Careful not to alter workload of other stages!

Ruling Out Other Stages



- If performance doesn't change significantly, you found the bottleneck
- Careful not to alter workload of stage under investigation!

Caveats

- Changes to one stage often affect other stages
- Often requires multiple tests to pinpoint bottleneck
  - See slide "Bottleneck Identification Flowchart" in printed proceedings for this talk
- Let's go over the various stages

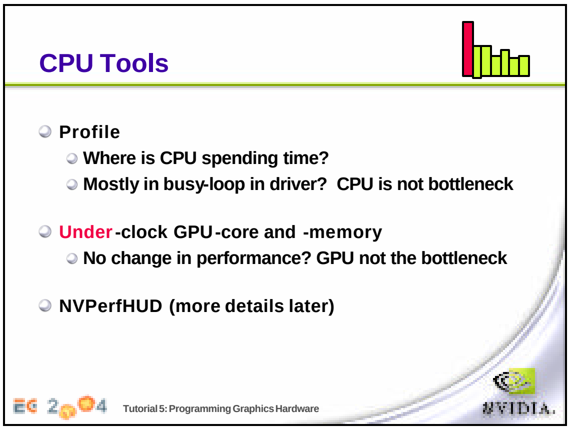
CPU Bottleneck

- Application
  - Complex physics, AI, or logic
  - Memory management (cache misses, disk)
- 3D API Usage
  - DirectX debug runtime: any errors or warnings?
  - Thousands of draw calls per frame

Reducing CPU Workload

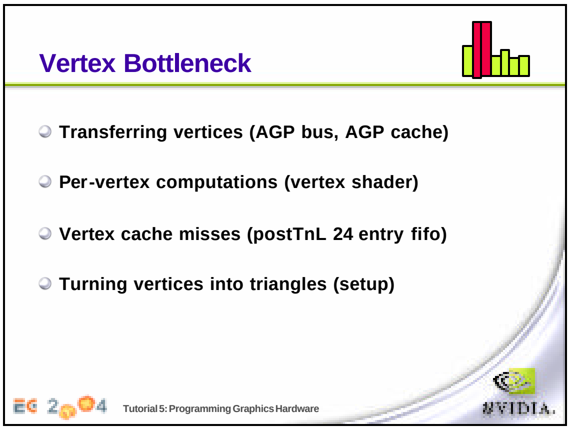
- Turn off parts of the application
  - Physics, AI, or logic
  - But don't change rendering workload
- Rule out GPU
  - Skip all DrawPrimitive() calls!

Bzzzt! Wrong: also reduces driver workload
    - Driver also runs on the CPU
  - Issue DrawPrimitive() calls as before
    - But only draw first triangle with each call



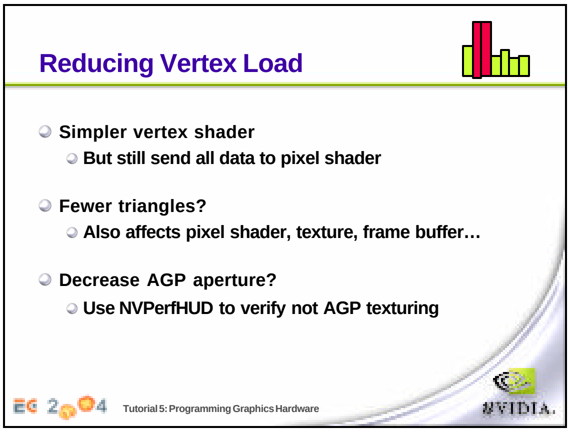
CPU Tools

- Profile
  - Where is CPU spending time?
  - Mostly in busy-loop in driver? CPU is not bottleneck
- Under-clock GPU-core and -memory
  - No change in performance? GPU not the bottleneck
- NVPerfHUD (more details later)



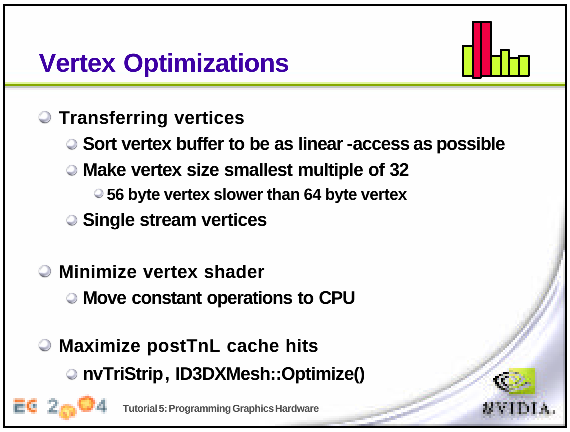
Vertex Bottleneck

- Transferring vertices (AGP bus, AGP cache)
- Per-vertex computations (vertex shader)
- Vertex cache misses (postTnL 24 entry fifo)
- Turning vertices into triangles (setup)



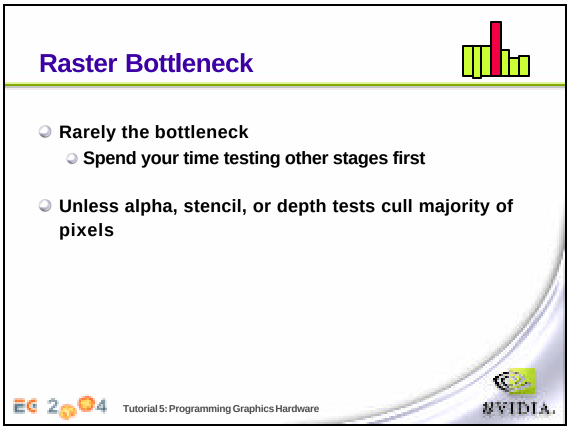
Reducing Vertex Load

- Simpler vertex shader
  - But still send all data to pixel shader
- Fewer triangles?
  - Also affects pixel shader, texture, frame buffer…
- Decrease AGP aperture?
  - Use NVPerfHUD to verify not AGP texturing



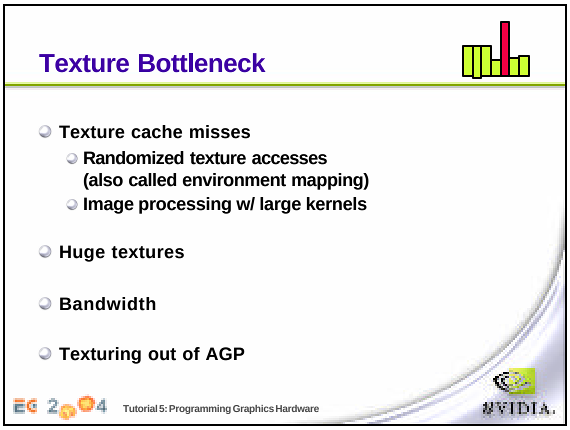
Vertex Optimizations

- Transferring vertices
  - Sort vertex buffer to be as linear-access as possible
  - Make vertex size smallest multiple of 32
    - 56 byte vertex slower than 64 byte vertex
  - Single stream vertices
- Minimize vertex shader
  - Move constant operations to CPU
- Maximize postTnL cache hits
  - nvTriStrip, ID3DXMesh::Optimize()



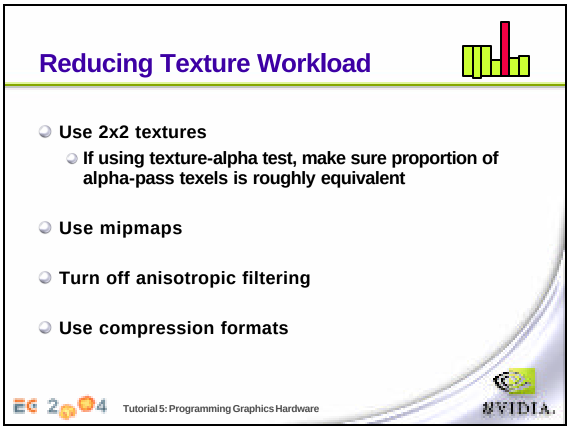
Raster Bottleneck

- Rarely the bottleneck
  - Spend your time testing other stages first
- Unless alpha, stencil, or depth tests cull majority of pixels



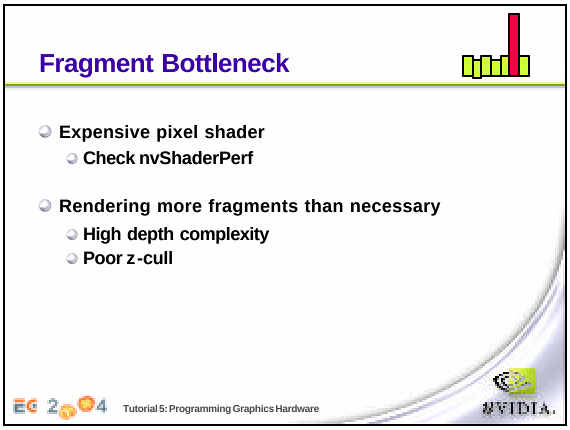
Texture Bottleneck

- Texture cache misses
  - Randomized texture accesses
    (also called environment mapping)
  - Image processing w/ large kernels
- Huge textures
- Bandwidth
- Texturing out of AGP



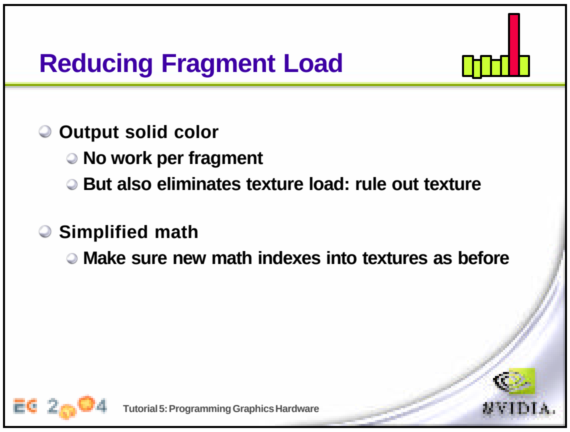
Reducing Texture Workload

- Use 2x2 textures
  - If using texture-alpha test, make sure proportion of alpha-pass texels is roughly equivalent
- Use mipmaps
- Turn off anisotropic filtering
- Use compression formats



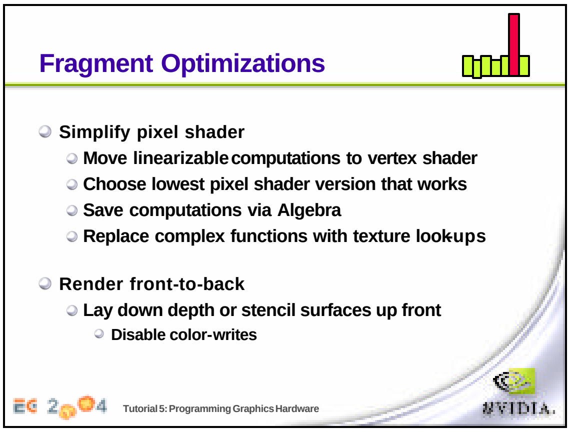
Fragment Bottleneck

- Expensive pixel shader
  - Check nvShaderPerf
- Rendering more fragments than necessary
  - High depth complexity
  - Poor z-cull



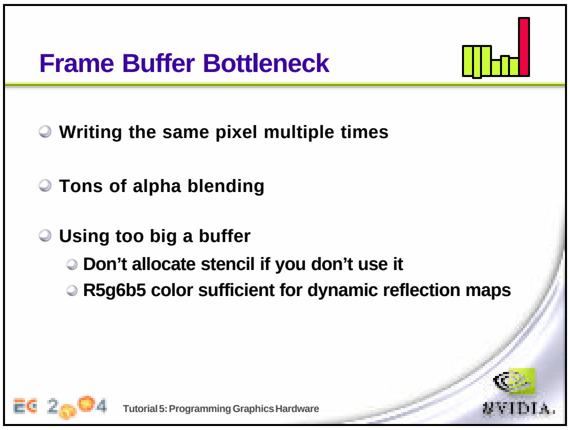
Reducing Fragment Load

- Output solid color
  - No work per fragment
  - But also eliminates texture load: rule out texture
- Simplified math
  - Make sure new math indexes into textures as before



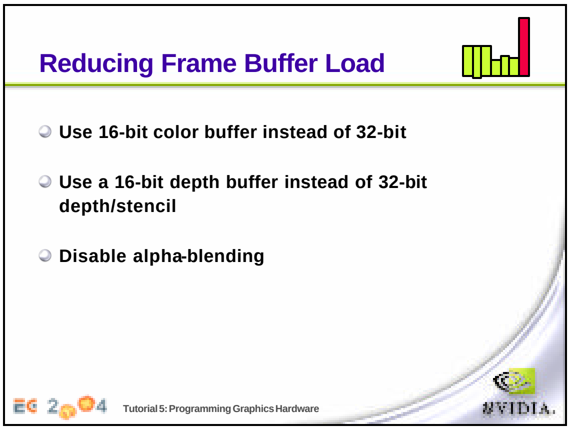
Fragment Optimizations

- Simplify pixel shader
  - Move linearizable computations to vertex shader
  - Choose lowest pixel shader version that works
  - Save computations via Algebra
  - Replace complex functions with texture look-ups
- Render front-to-back
  - Lay down depth or stencil surfaces up front
    - Disable color-writes



Frame Buffer Bottleneck

- Writing the same pixel multiple times
- Tons of alpha blending
- Using too big a buffer
  - Don’t allocate stencil if you don’t use it
  - R5g6b5 color sufficient for dynamic reflection maps



Reducing Frame Buffer Load

- Use 16-bit color buffer instead of 32-bit
- Use a 16-bit depth buffer instead of 32-bit depth/stencil
- Disable alpha-blending

Enough Theory, Let's Talk Tools

- Any questions on
  - Bottleneck identification?
  - Optimizations?

Tools Overview

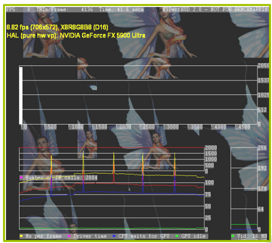
- nvPerfHUD
  - Registered NVIDIA Developer website https://nvdeveloper.nvidia.com/login.asp
- nvShaderPerf
  - http://developer.nvidia.com/object/nvshaderperf_home.html
  - Integrated into FX Composer http://developer.nvidia.com/object/fx_composer_home.html

More Tools

- CPU Profiler
  - E.g., AMD's CodeAnalyst
- Under-clocking utilities
  - BIOS
    - For CPU clock, FSB clock, AGP speed
  - NVIDIA control panel
    - For GPU core- and memory-clocks

NVPerfHud

- Free!
- Batches
- GPU idle
- Total time
- Time CPU waits for GPU
- Driver time
- Solid color pixel shaders
- 2x2 textures



FPS: 8 TRIs/Frame: 4136 Time: 41.6 secs NVPerfHUD 2.0 – NOT FOR BENCHMARKING

8.82 fps (706x572), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra

Practice

- Sample problems
  - Can you find what the problem is?
  - How would you fix it?
- Using NVPerfHUD to help

Practice: Clean the Machine!

- Is your profiling machine equivalent to target?
  - Using your 3GHz CPU for profiling application supposed to run well on a 2GHz CPU is pointless
  - Latest drivers of everything?
  - No control panel anisotropic filtering or anti-aliasing
  - Make sure v-sync is off
- Use the DirectX Release runtime
  - Debug runtime good for errors and warnings check
- Use release/optimized build of application

Example 1

- A seemingly simple scene runs horribly slow
  - Narrow in on the bottleneck

19.54 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra

Example 1 Code

- Uses a dynamic vertex buffer
  - Bad creation flags

```
HRESULT hr = pd3dDevice->CreateVertexBuffer(
                         6* sizeof( PARTICLE_VERT ),
                         0,      // declares as static&read&write
                         PARTICLE_VERT::FVF,
                         D3DPOOL_DEFAULT,
                         &m_pVB,
                         NULL );
```

Set Proper Creation Flags

- Tell runtime and driver as much as possible

```
HRESULT hr = pd3dDevice->CreateVertexBuffer(
                         6* sizeof( PARTICLE_VERT ),
                         D3DUSAGE_DYNAMIC |
                         D3DUSAGE_WRITEONLY,
                         PARTICLE_VERT::FVF,
                         D3DPOOL_DEFAULT,
                         &m_pVB,
                         NULL );
```

Locking Flags?

```
m_pVB->Lock(0, 0,(void**)&quadTris, 0);
```

- No flags at all? That can't be good…
- Means you will read…
- And write
  - Potentially anywhere on the buffer
- Driver must copy the buffer for you
  - Potentially wait for GPU to finish using it first
  - Synchronizes CPU and GPU

Set Proper Locking Flags

```
m_pVB->Lock(0, 0,(void**)&quadTris,
D3DLOCK_NOSYSLOCK | D3DLOCK_DISCARD);
```

- Use D3DLOCK_DISCARD first time you lock a vertex buffer each frame
  - And again when that buffer is full
  - Otherwise use NOSYSLOCK | NOOVERWRITE

Example 2: Another Slow Scene

9.55 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra

Texture Bandwidth Overkill

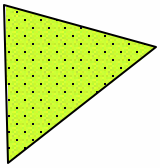
- Use mipmaps
- Use dxt1 if possible
  - Some cards store compressed data in cache
- Use smaller textures when possible
  - Do grass blades really need 1024x1024 textures?

And Another One

9.41 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra

Expensive Pixel Shader

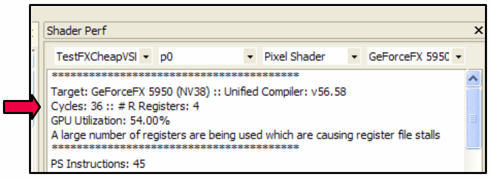
- Only 3 verts, but maybe a million pixels
  - That's only 1024x1024



Look at all those pixels!!

nvShaderPerf

- 36 cycles!

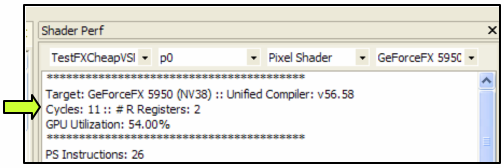


```
*********************************************
Target: GeForceFX 5950 (NV38) :: Unified Compiler: v56.58
Cycles: 36 :: # R Registers: 4
GPU Utilization: 54.00%
A large number of registers are being used which are causing register file stalls
*********************************************
PS Instructions: 45
```

Optimizing the Pixel Shader

- Move math that is constant across triangle into vertex shader
- Use 'half' instead of 'float'
- Get rid of unnecessary normalize()s
  - See also Normalization Heuristics http://developer.nvidia.com/object/normalization_heuristics.html

11 Cycles Is Better!



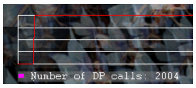
```
*********************************************
Target: GeForceFX 5950 (NV38) :: Unified Compiler: v56.58
Cycles: 11 :: # R Registers: 2
GPU Utilization: 54.00%
*********************************************
PS Instructions: 26
```

Last Example

47.26 fps (1280x948), X8R8G8B8 (D16)
HAL (pure hw vp): NVIDIA GeForce FX 5900 Ultra

Too Many Batches

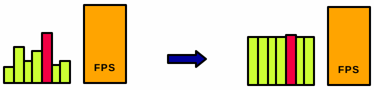
- Every quad uses its own Draw() call
- Pack all quads into one big vertex buffer
  - Send with one Draw() call



- What if quads use different textures?
  - Pack textures into atlases
  - Change texture coordinates on quads accordingly
  - See NVIDIA SDK 7, Atlas Comparison Viewer

Balancing the Pipeline

- Once satisfied with performance
  - Balance pipeline:
    - make more use of non-bottlenecked stages
  - Careful not to make too much use of them



Summary

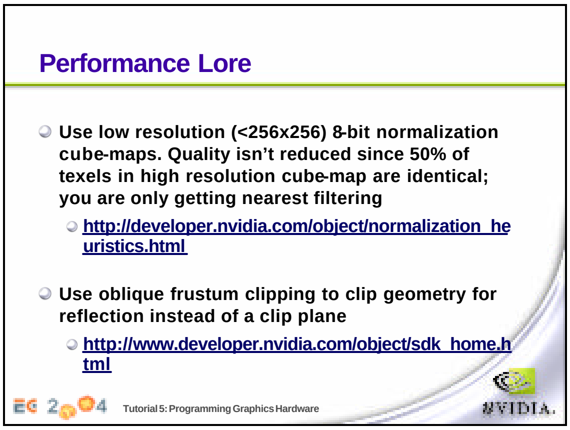
- Graphics is a multi-processor pipeline
- Bottlenecks rule pipeline architectures
- Don’t waste time optimizing stages needlessly
- Identify bottlenecks with quick tests
- Use NVPerfHUD to analyze your pipeline
- Use Fxcomposer to help tune your shaders
- Check your performance early and often
  - Don’t wait until a week before ship!

More Information

- http://developer.nvidia.com
  The Source for GPU Programming
- NVIDIA GPU Programming Guide
  http://developer.nvidia.com/object/gpu_programming_guide.html
- Matthias Wloka (mwloka@nvidia.com)

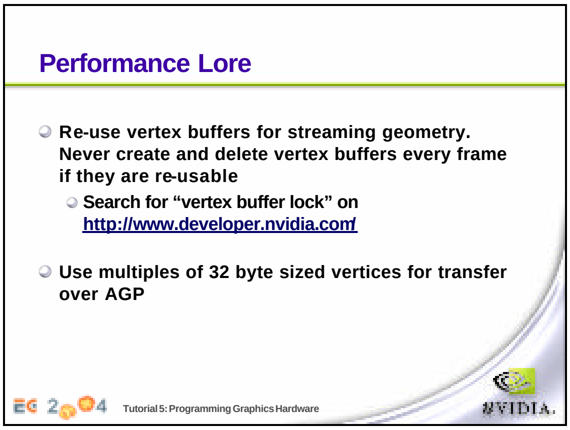
Performance Lore

- Collected advice from various developers
- So you don’t have to discover it the hard way



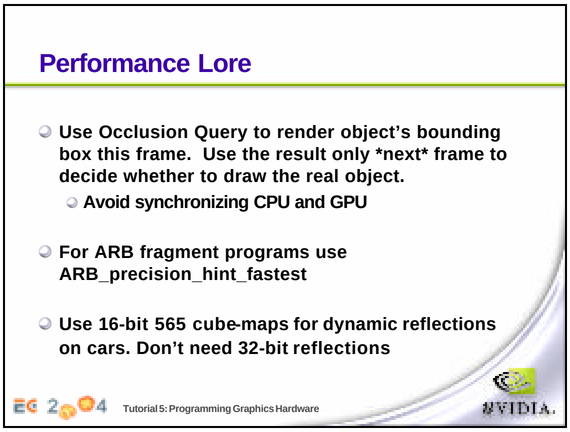
Performance Lore

- Use low resolution (<256x256) 8-bit normalization cube-maps. Quality isn't reduced since 50% of texels in high resolution cube-map are identical; you are only getting nearest filtering
  - http://developer.nvidia.com/object/normalization_heuristics.html
- Use oblique frustum clipping to clip geometry for reflection instead of a clip plane
  - http://www.developer.nvidia.com/object/sdk_home.html



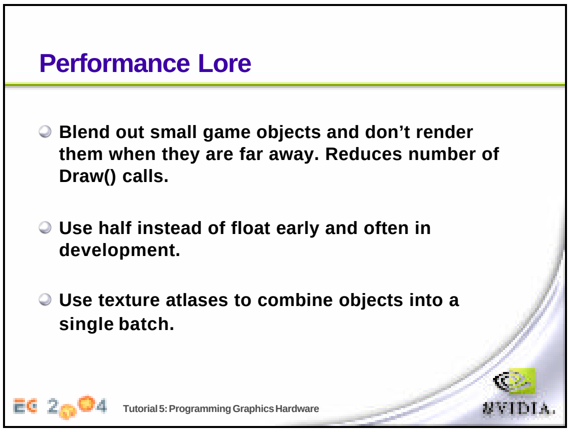
Performance Lore

- Re-use vertex buffers for streaming geometry. Never create and delete vertex buffers every frame if they are re-usable
  - Search for "vertex buffer lock" on http://www.developer.nvidia.com/
- Use multiples of 32 byte sized vertices for transfer over AGP



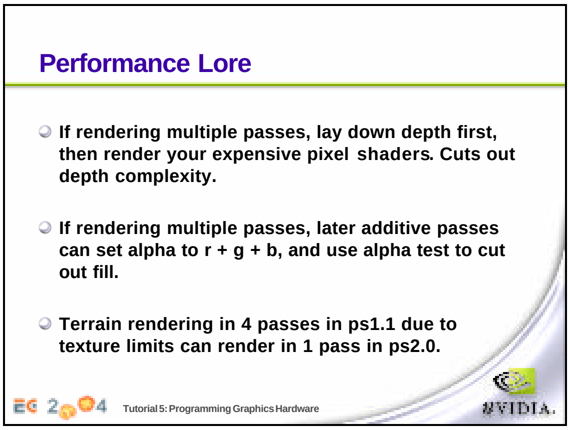
Performance Lore

- Use Occlusion Query to render object's bounding box this frame. Use the result only *next* frame to decide whether to draw the real object.
  - Avoid synchronizing CPU and GPU
- For ARB fragment programs use ARB_precision_hint_fastest
- Use 16-bit 565 cube-maps for dynamic reflections on cars. Don't need 32-bit reflections



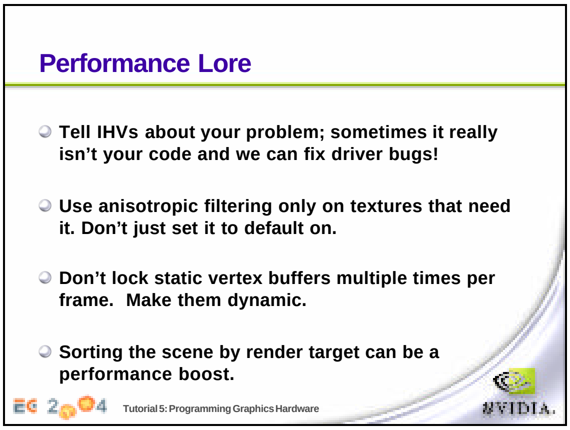
Performance Lore

- Blend out small game objects and don't render them when they are far away. Reduces number of Draw() calls.
- Use half instead of float early and often in development.
- Use texture atlases to combine objects into a single batch.



Performance Lore

- If rendering multiple passes, lay down depth first, then render your expensive pixel shaders. Cuts out depth complexity.
- If rendering multiple passes, later additive passes can set alpha to r + g + b, and use alpha test to cut out fill.
- Terrain rendering in 4 passes in ps1.1 due to texture limits can render in 1 pass in ps2.0.



Performance Lore

- Tell IHVs about your problem; sometimes it really isn't your code and we can fix driver bugs!
- Use anisotropic filtering only on textures that need it. Don't just set it to default on.
- Don't lock static vertex buffers multiple times per frame. Make them dynamic.
- Sorting the scene by render target can be a performance boost.

Performance Lore

- When locating the bottleneck, divide and conquer. Lower resolution first, cuts the problem almost in half. Rules out just about everything fill and pixel related.
- Use float4 to pack multiple float2 texture coordinates.
- Optimize your index and vertex buffers to take advantage of the cache.

Performance Lore

- Move per object calculations out of the vertex shader and onto the CPU.
- Move per triangle calculations out of the pixel shader and into the vertex shader.
- Use swizzles and masks in your vertex and pixel shaders: Value.xy = register.wz

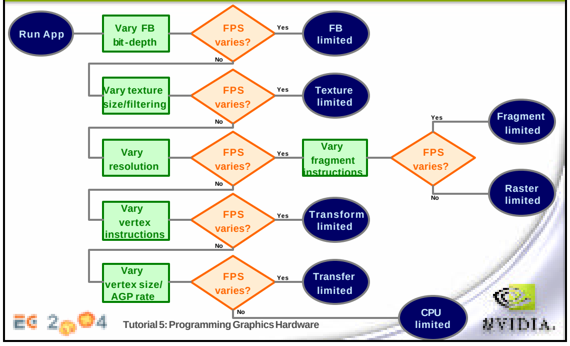
Performance Lore

- Use the API to clear the color and depth buffer.
- Don’t change the direction of your z test mid frame
  - Going from > ...to... >= ...to... = is fine
  - Don’t go from > ...to... <
- Don’t use polygon offset if something else works.
- Don’t write depth in your pixel shader if you don’t have to.

Performance Lore

- Use mipmaps. If they are too blurry for you, use anisotropic filtering: Better quality than LOD bias.
- Rarely is there a single bottleneck in a game. If you find a bottleneck and fix it, and performance doesn’t improve more than a few fps, don’t give up. You’ve helped yourself by making the real bottleneck apparent. Keep narrowing it down until you find it.

Bottleneck Identification Flowchart


Run App
Vary FB bit-depth
FPS varies?
Yes
FB limited
No
Vary texture size/filtering
FPS varies?
Yes
Texture limited
No
Vary resolution
FPS varies?
Yes
Vary fragment instructions
FPS varies?
Yes
Fragment limited
No
Raster limited
No
Vary vertex instructions
FPS varies?
Yes
Transform limited
No
Vary vertex size/ AGP rate
FPS varies?
Yes
Transfer limited
No
CPU limited