



**GDC Europe 2005**

***Feeding the Beast:***  
**How to Satisfy Your GeForce**  
**While Differentiating Your Game**

**Lars M. Bishop**

**NVIDIA Embedded Developer Technology**



# Agenda

- GoForce 3D capabilities
  - Strengths and weaknesses
  - What that means to application developers
- Common performance bottlenecks
  - From a hardware vendor's viewpoint
- Techniques to optimize performance
  - While simultaneously maximizing quality



# What is GoForce 3D?

- Licensable 3D core for mobile devices
- Discrete solutions: GoForce 3D 4500/4800
- OpenGL ES compliant
- Low power
- Integrated SRAM
- Up to VGA resolution



# GoForce 3D 4800 Features

- Geometry engine
- 16-bit color w/ 16-bit Z (40-bit color internal)
- Fully perspective correct
- Sub-pixel accuracy
- Per-pixel fog, alpha blending, alpha-test

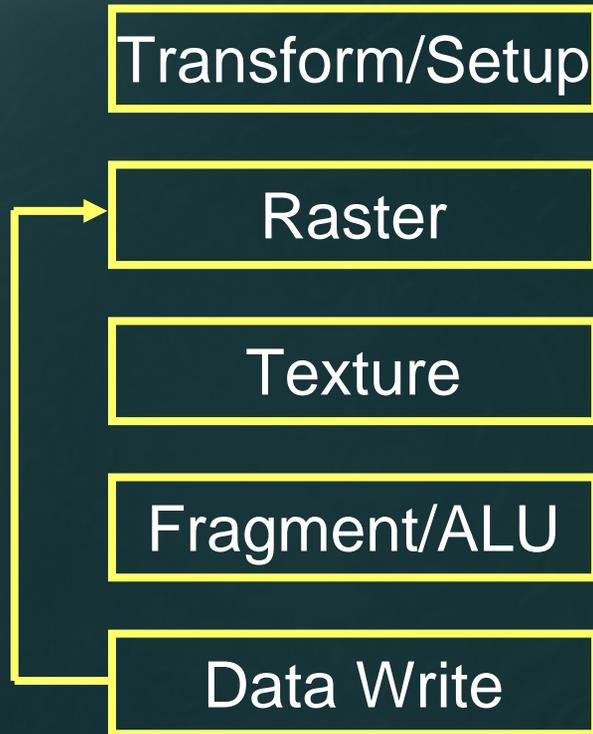


## GoForce 3D 4800 Features cont.

- Multi-texturing w/ up to 4 textures
  - In OpenGL ES 1.0!!
- Flexible texture formats (DXT1/4-bit/8-bit)
- Bilinear / trilinear filtering



# GoForce 3D Pipeline



(~50 pipe stages)

- Flexible Fragment ALU
- Raster – fragment generation and loop management
- Scalable
- Stages only trigger on activity
- Low power
- <50mW per 100M pixel/sec, actual game play

# Common Performance Bottlenecks



## ● CPU

- Low clock speeds
- HW floating point functionality *not* standard

## ● System memory bandwidth (bus)

- This is not your PCI Express x16

## ● GPU

- Powerful, but features are not “free”
- Need to pay attention to the features you enable



# Performance Bottleneck Causes

- CPU-bound
  - App-level work
  - Non-accelerated driver work
- Bus-bound
  - Geometry issues
  - Texture issues
- GPU-bound
  - Raster feature usage / multipass rendering



# App-level CPU Bottlenecks

- Simplify physics, collision, visibility
- Avoid non-coherent algorithms
  - Small caches: data incoherency hurts performance
- Avoid floating point *or* integer division
- Use
  - Fixed point if CPU does not support floating point
  - Floating point otherwise



# Render-related CPU Bottlenecks

- Batch geometry
  - Minimize number of state changes
  - i.e. sort geometry into buckets of common state
- Avoid examining every triangle every frame
  - e.g. per-triangle app-level culling
- Avoid multi pass
  - Use more complex fragment shaders instead
- Avoid CPU-assisted driver paths...

# CPU-Assisted Vertex Computations

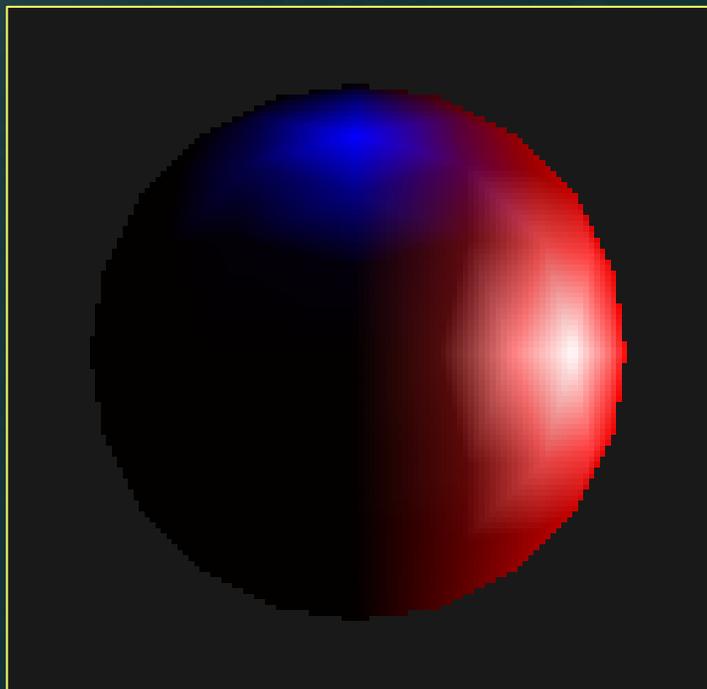


- Vertex lighting is NOT GPU accelerated
- Non-identity texture matrices NOT accelerated
- But do you *really* want per vertex lighting?
  - Per vertex specular is bad, unless highly tessellated
- Use per pixel computations instead
  - Per pixel tricks
  - Per pixel lighting

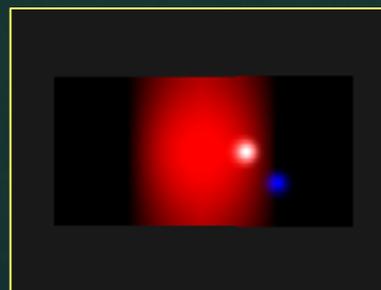
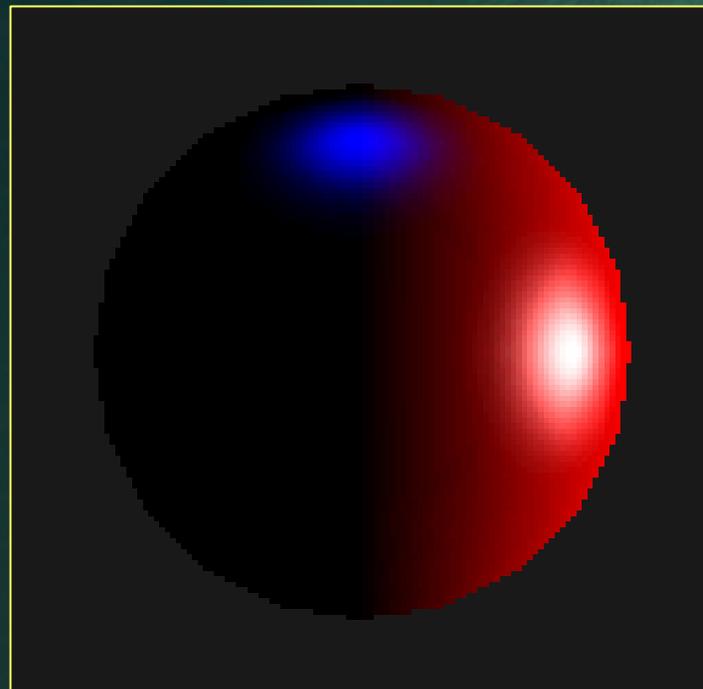


# Specular Lighting Screenshots

Per Vertex



Per Pixel





# Alternative Lighting Strategies

- Fake Phong highlights using multi-texture
- Pre-compute vertex lighting



Stuntcar Extreme  
(Images Courtesy of Fathammer)

# System Memory Bandwidth: Vertices



- Avoid transferring large number of vertices
  - Cull high-level geometry in app: not per triangle
  - Use LODs
  - Use impostors and billboards
- Minimize per vertex size
  - Don't specify z or w if unused (e.g. screen-space)
  - Use packed ARGB (not floating point)
- Optimize vertex order for the vertex cache

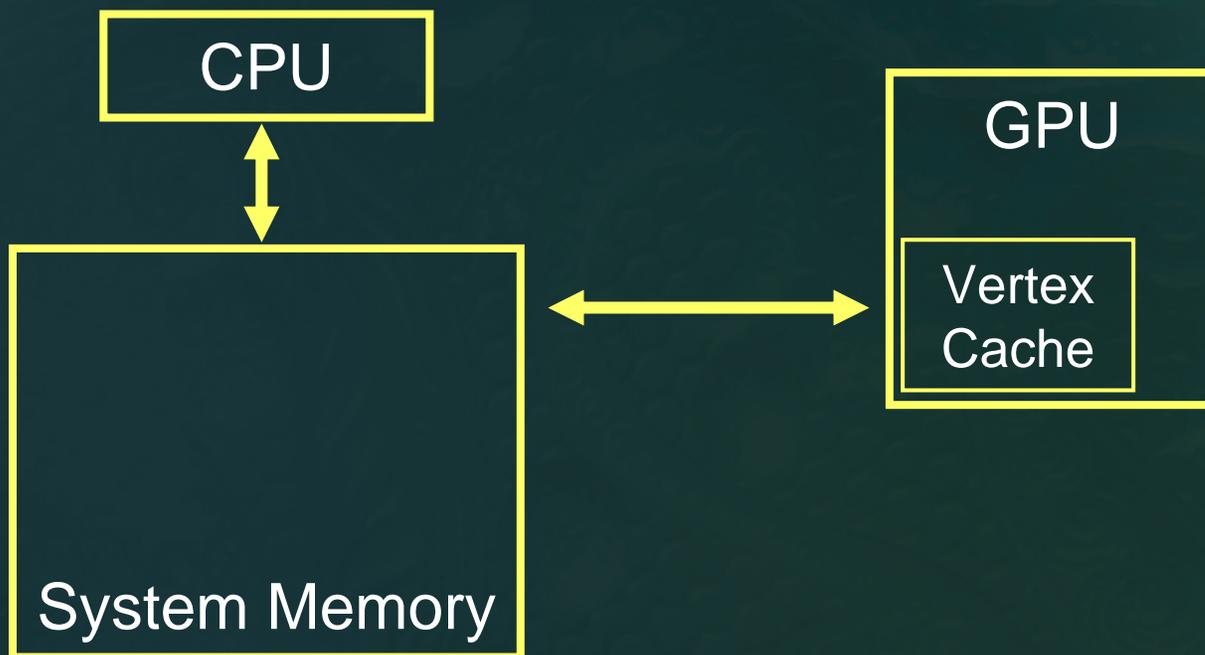


# Vertex Cache Optimization

- What is a vertex cache?
  - Place for the GPU to store vertex data
  - If vertex found in cache, GPU uses cached copy
- How to utilize this?
  - Use longest possible triangle strip or fan, or
  - Use indexed triangle lists, LRU cache
- Optimize for 32- or 16-entry LRU vertex cache
  - 32 8-word verts *or*
  - 16 16-word verts



# Vertex Cache Diagram





# Avoiding Vertex Computations

- Avoid multi-pass
  - Every vertex processed once per pass
  - *Much* more efficient to render in a single pass using a more complex fragment shader
- Minimize number of vertices
  - Use LODs: Remember your screen size!
  - Cull high-level geometry in app: not per triangle
- Use triangle strips or (optimized) indexed tri lists

# System Memory Bandwidth: Textures



- Avoid overflowing GPU texture memory
  - Memory contains frame buffer and textures
- GoForce 3D 4800: 1280K SRAM
  - Texture memory: 1280K – frame buffer
  - QVGA (320x240) 16-bit color double-buffered w/ z:
    - Frame buffer: 450K
    - Texture memory available: 830K

# Avoid Overflowing Texture Memory



- Avoid allocating full-screen buffers
- Use compact texture formats
  - EXT\_texture\_compression\_dxt1 (4-bit/texel)
  - Palette textures
- Use lowest texture resolution possible
  - See GPU Gems 2, Chapter 28:  
“Mipmap-Level Measurement”
- If you must exceed budget, sort by texture



# Palette Textures

- HW supports
  - 8bit palettes
  - 4bit palettes
- But only 256 entries total in a rendering pass
  - So only one 8bit palette texture per fragment shader
  - Or up to 16 4bit palettes
  - i.e. can't multi-texture from 2 different 8bit palettes



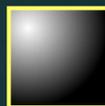
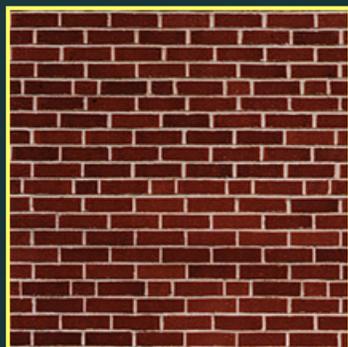
## Benefits of DXT1

- Bubble: scene uses 8 textures
  - 2 – 256x256 textures (mipmapped)
  - 6 – 256x256 textures (not mipmapped)
- Space:
  - RGB565 (16bits/texel) = 1.08MB
  - DXT1 (4bits/texel) = 0.27MB
- DXT1: high quality & 25% the size of RGB565
  - Actually, also faster to texture from





# Leverage Multi-Texture



Four Full Res (256 x 256) 4-bit = 128KB  
Full Res Base + four 1/4 res lightmaps = 40KB

- Store diffuse maps in lower resolution and use multi-texture to save memory



# Texture Size

- Avoid allocating alpha if unused
  - DXT1 supports RGBA and RGB
- Avoid allocating mipmaps if not useful
  - Don't bother skipping just 1x1 & 2x2 levels:
    - Very negligible space savings
  - Consider NO mipmaps:
    - Reasonable space savings
    - Saves ~50 clocks of triangle setup calculations
    - But not if image quality suffers (sparklies)



# GPU Bottleneck Overview

- Texture optimizations
  - Multi-texturing
  - Filtering
- Fragment optimization
  - Not all features “cost” the same
  - Avoiding framebuffer reads
  - Avoiding depth-buffer expense



# Texture Filtering

- Bilinear filtering is 'free'
- Trilinear filtering is half-speed
  - Compared to bilinear
- Use trilinear filtering when needed
  - Don't turn it on by default



# Fragment Shading

- Do as much work as possible in each fragment
- Use Combiner Programs
  - NV\_combiner\_program extension from OpenGL ES
- Combiner Programs allow for powerful effects
  - DOT3 bump/normal mapping
  - Environment-mapped bump mapping
  - Image processing (blurring, edge detect, etc)
  - Up to four textures per pass



# Assembly Language Consists of

- Different types of registers
- Instructions
- Write masking and co-issue
- Argument modifiers



# Overall Program Limits

- Up to 4 stages
- Each stage consists of
  - Optional color interpolation
  - Optional texture look-up
  - Math instruction(s)

```
STAGE
color GL_DIFFUSE

texld v0.x, v0.y, TEX0
mul   r0, tex, col
```



# Register Types

- Temporary registers: r0, r1
  - read and write, persistent across stages
- Constant registers: c0-c3
  - read only
- Color register: col
  - read only
- Texture register: tex and tex.a
  - read only and read/write, respectively



# Instructions

- mov, mul, add, mad, lrp, min, max
- seq, sne, sge, sgt, sle, slt
- mmad:  $(a*b) + (c*d)$ 
  - mmul:  $r0 = a*b, r1 = c*d$
  - mmin:  $\min(a*b, c*d)$ ; mmax:  $\max(a*b, c*d)$
  - mseq, msne, msge, msgt, msle, mslt
- keq, kne, kge, kgt, kle, klt
  - mkeq, mkne, mkge, mkgt, mkle, mklt



# Write Masking and Co-Issue

- Optional write mask
  - Legal: .rgba, .rgb, .a
  - Defaults to .rgba
  - `add r0.a, r0, r1`  
`mul r0.rgb, c0, r1`
- Co-issue:
  - Alpha and rgb parts execute separately
  - Alpha result available before rgb computation

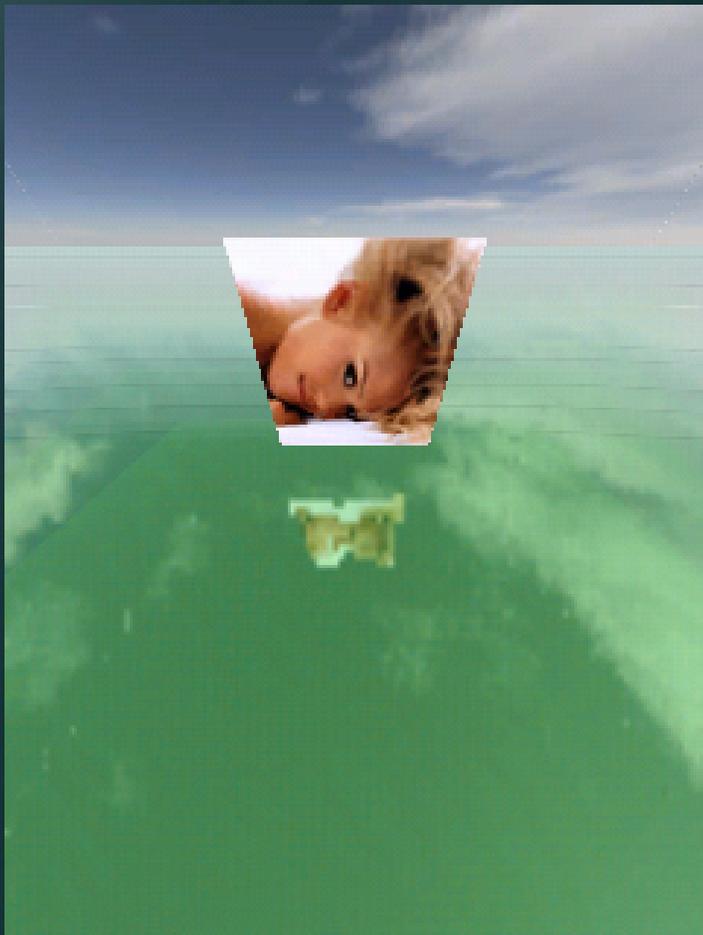


# Modifiers

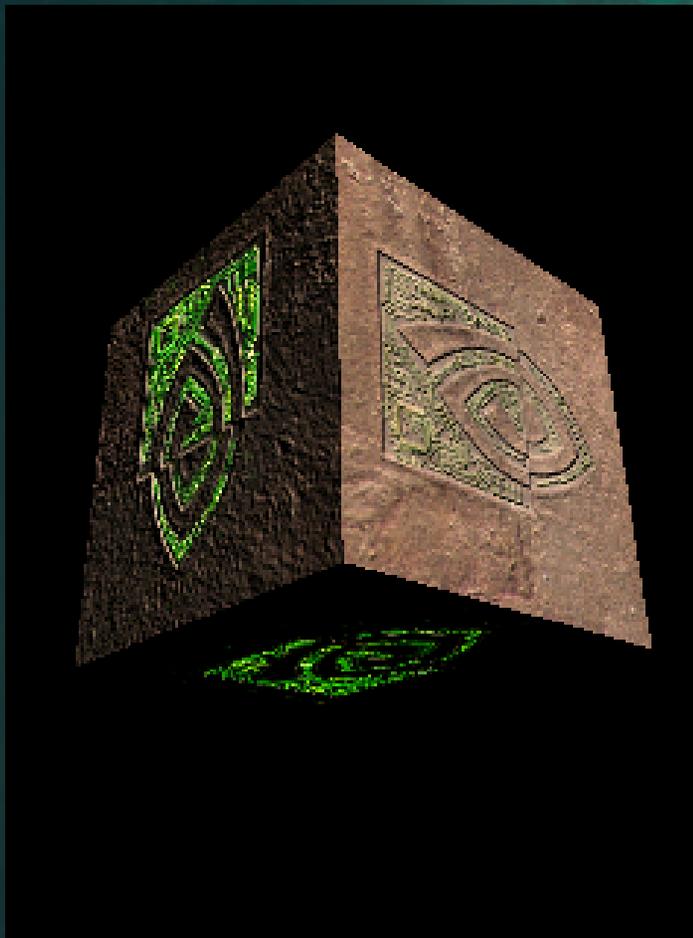
- Destination clamp
- Source negate
- Source swizzle (tex and color registers)
  - Not on temporaries or constants
  - “smear”: tex.a means tex.aaaa
  - “mux”: one of each r, g, b, a (e.g. tex.bgar)



# Combiner Program Examples



**Environment-mapped  
Bumpmapping**



**Glow + DOT3 Bumpmapping**



# Fragment Shader Considerations

- Pixel failing z costs same number of cycles
  - i.e., no z-cull, no early out
- But killed pixels do not load texels from cache
  - Slightly faster
  - Consumes less power
- May consider rough front-to-back rendering
  - Do not override state sorting!

# Fragment Shader Instruction Costs



- Single cycle
  - Single texture, color-interpolated, z-buffered
- Additional cycle for each:
  - Additional texture stage
  - Alpha test
  - Alpha blending
  - Fog
  - Color masking

# Raster-OP Readback: Turn Them Off



- Alpha blending consumes bandwidth
  - Do not assume  $1 * \text{src} - 0 * \text{dest}$  is optimal (it's not)
- Color masking actually reads framebuffer
  - More expensive than not masking
- Z comparison reads z-buffer
- If possible, turn *all three* above options *off*
  - You'll get an extra throughput boost in rasterization



## Other Advice

- Avoid framebuffer LogicOp (bitwise AND, etc)
  - Only rudimentary support (expensive)
- Don't read from framebuffer
  - `glReadPixels()`
- Avoid dynamic texture updates
  - Never per-frame on large textures



## Other Advice (Continued)

- There is (currently) no push buffer
  - Run CPU and GPU in parallel as much as possible
  - Going to get better soon
- Full screen apps can flip
  - Pointer swap versus blit (actual memory copy)



# Conclusion

- Offload CPU
  - Avoid CPU driver paths
  - Avoid multi-pass, prefer multi-texture
  - Avoid state changes
- Watch vertex computations and bandwidth
- Watch video memory usage
  - But video memory will continue to increase
- Shift load to fragment shaders



# Plan for the Future

- NVIDIA is committed to handheld 3D
  - Expect exciting advances in HW soon
- Prepare your content to scale up in all directions
- Expect future GoForce products to have:
  - More powerful/flexible shader capabilities
  - More VRAM
  - Increased vertex/triangle throughput

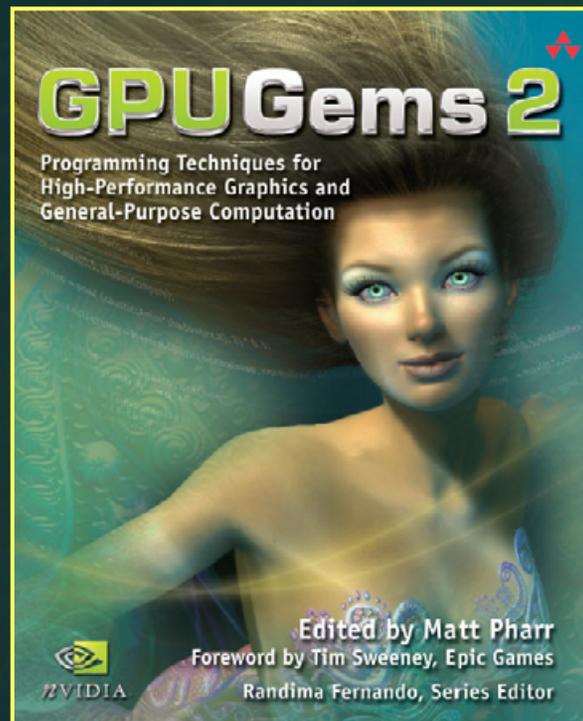
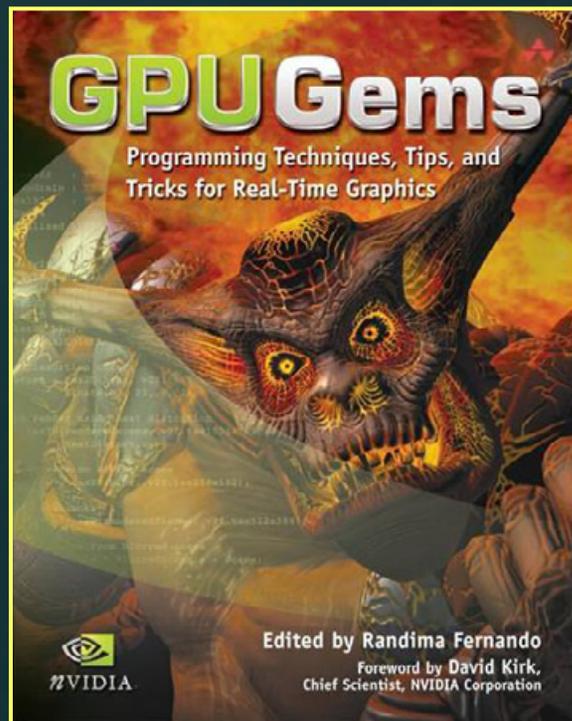
# Sooner Than You'd Think!





# GPU Gems Plug

- Not (yet) 100% applicable, but fast approaching!





# Questions?

- Register for NVIDIA Handheld Developer Program

<http://developer.nvidia.com>

- Email:

[handset-dev@nvidia.com](mailto:handset-dev@nvidia.com)

# The Source for GPU Programming

[developer.nvidia.com](http://developer.nvidia.com)

- Latest News
- Developer Events Calendar
- Technical Documentation
- Conference Presentations
- GPU Programming Guide
- Powerful Tools, SDKs and more ...

Join our FREE registered developer program for early access to NVIDIA drivers, cutting edge tools, online support forums, and more.



**NVIDIA**

[developer.nvidia.com](http://developer.nvidia.com)

©2004 NVIDIA Corporation. NVIDIA, and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation. Nalu is ©2004 NVIDIA Corporation. All rights reserved.