OpenGL Performance

John Spitzer

GAME Developers CONFERENCE 2001

OpenGL Performance

John Spitzer

Manager, OpenGL Applications Engineering

jspitzer@nvidia.com

Possible Performance Bottlenecks

They mirror the OpenGL pipeline

- **High order surface evaluation**
- **Data transfer from application to GPU**
- **Vertex operations (fixed function or vertex program)**
- **Texture mapping**
- **Other per-fragment operations**

High Order Surface Evaluation

Each patch requires an amount of driver setup

- **CPU dependent for coarse tessellation**
- **Will improve as driver optimizations are completed**
- **Integer tessellation factors faster than fractional**
- **Driver may cache display-listed patches, so that setup is amortized over many frames for static patches**

Evaluating Higher Order Surfaces

- Use surfaces if:
  - You tessellate to more than

    Degree_U * Degree_V * 3
- Driver data is smaller than equivalent triangles
- Cost of increasing the tessellation is small
  - Cost is roughly Sqrt(N) up to 512 triangles.

Transferring Geometric Data from App to GPU

So many ways to do it

- **Immediate mode**
- **Display lists**
- **Vertex arrays**
- **Compiled vertex arrays**
- **Vertex array range extension**

Immediate Mode

The old stand-by

- **Has the most flexibility**
- **Makes the most calls**
- **Has the highest CPU overhead**
- **Varies in performance depending on CPU speed**
- **Not the most efficient**

Display Lists

Fast, but limited

- **Immutable**
- **Requires driver to allocate memory to hold data**
- **Allows large amount of driver optimization**
- **Can sometimes be cached in fast memory**
- **Typically very fast**

Vertex Arrays

Best of both worlds

- **Data can be changed as often as you like**
- **Data can be interleaved or in separate arrays**
- **Can use straight lists or indices**
- **Reduces number of API calls vs. immediate mode**
- **Little room for driver optimization, since data referenced by pointers can change at any time**

Compiled Vertex Arrays

Solve part of the problem

- Allow user to lock portions of vertex array
- In turn, gives driver more optimization opportunities:
  - Shared vertices can be detected, allowing driver to eliminate superfluous operations
  - Locked data can be copied to higher bandwidth memory for more efficient transfer to the GPU
- Still requires transferring data twice

Vertex Array Range (VAR) Extension

VAR allows the GPU to pull vertex data directly

- Eliminates double copy
- Analogous to Direct3D vertex buffers
- VAR memory must be specially allocated from AGP or video memory
- Facilitates post-T&L vertex caching
- Introduces synchronization issues between CPU and GPU

VAR Memory Allocation and Issues

Video memory has faster GPU access, but precious

- Same memory used for framebuffer and textures
- Could cause texture thrashing
- On systems without Fast Writes, writes will be slow

AGP memory typically easier to use

- Write sequentially to take advantage of CPU's write combiners (and maximize memory bandwidth)
- Usually as fast as Video memory

Best to use either one large chunk of AGP or one large chunk of Video; switching is expensive

NEVER read from either type of memory

Facilitates Post-T&L Vertex Caching

All NVIDIA GPUs have post-T&L caches

- Replacement policy is FIFO-based
- 16 elements on GeForce 256 and GeForce2 GPUs
- 24 elements on GeForce3
- Effective sizes are smaller due to pipelining

Only activated when used in conjunction with:

- Compiled vertex arrays or VAR
- glDrawElements or glDrawRangeElements

Vertex cache hits more important than primitive type, though strips are still faster than lists

Do not create degenerate triangle strips

VAR Synchronization Issues

CPU is writing to and the GPU reading from the same memory simultaneously

Must ensure the GPU is finished reading before the memory is overwritten by the CPU

Current synchronization methods are insufficient:

- glFinish and glFlushVertexArrayRangeNV are too heavy-handed; block CPU until GPU is all done
- glFlush is merely guaranteed to "complete finite time" – could be next week sometime

Need a mechanism to perform a "partial finish"

- Introduce token into command stream
- Force the GPU to finish up to that token

Introducing: NV_fence

NV_fence provides *fine-grained* synchronization

- **A “fence” is a token that can be placed into the OpenGL command stream by glSetFenceNV**
- **Each fence has a condition that can be tested (only condition available is GL_ALL_COMPLETED_NV)**
- **glFinishFenceNV forces the app (i.e. CPU) to wait until a specific fence’s condition is satisfied**
- **glTestFenceNV queries whether a particular fence has been completed yet or not, without blocking**

How to Use VAR/fence Together

Combination of VAR and fences is very powerful

- VAR gives the best possible T&L performance
- With fences, apps can achieve very efficient pipelining of CPU and GPU

In memory-limited situations, VAR memory must be reused

- Fences can be placed in the command stream to determine when memory can be reclaimed
- Different strategies can be used for dynamic/static data arrays
- App must manage memory itself

VAR/fence Performance Hints

Effective memory management and synchronization is key

- **Delineate static/dynamic arrays**
- **Avoid redundant copies**
  - **Do this on a per-array bases, not per-object**
  - **Be clever in your use of memory**
- **Use fences to keep CPU and GPU working in parallel**

Other Vertex Array Range Issues

Separate vs. Interleaved Arrays

- If an object's arrays are a mix of static and dynamic, then interleaved writing will be inefficient
- Interleaved arrays may be slightly faster though

Only use VAR when HW T&L is used

- Not with vertex programs on NV1X
- Not with feedback/selection

VAR array can be of arbitrary size, but indices must be less than 1M (1,048,576) on GeForce3

Data arrays must be 8-byte aligned

Vertex Operations

Vertex Program or Fixed Function Pipeline?

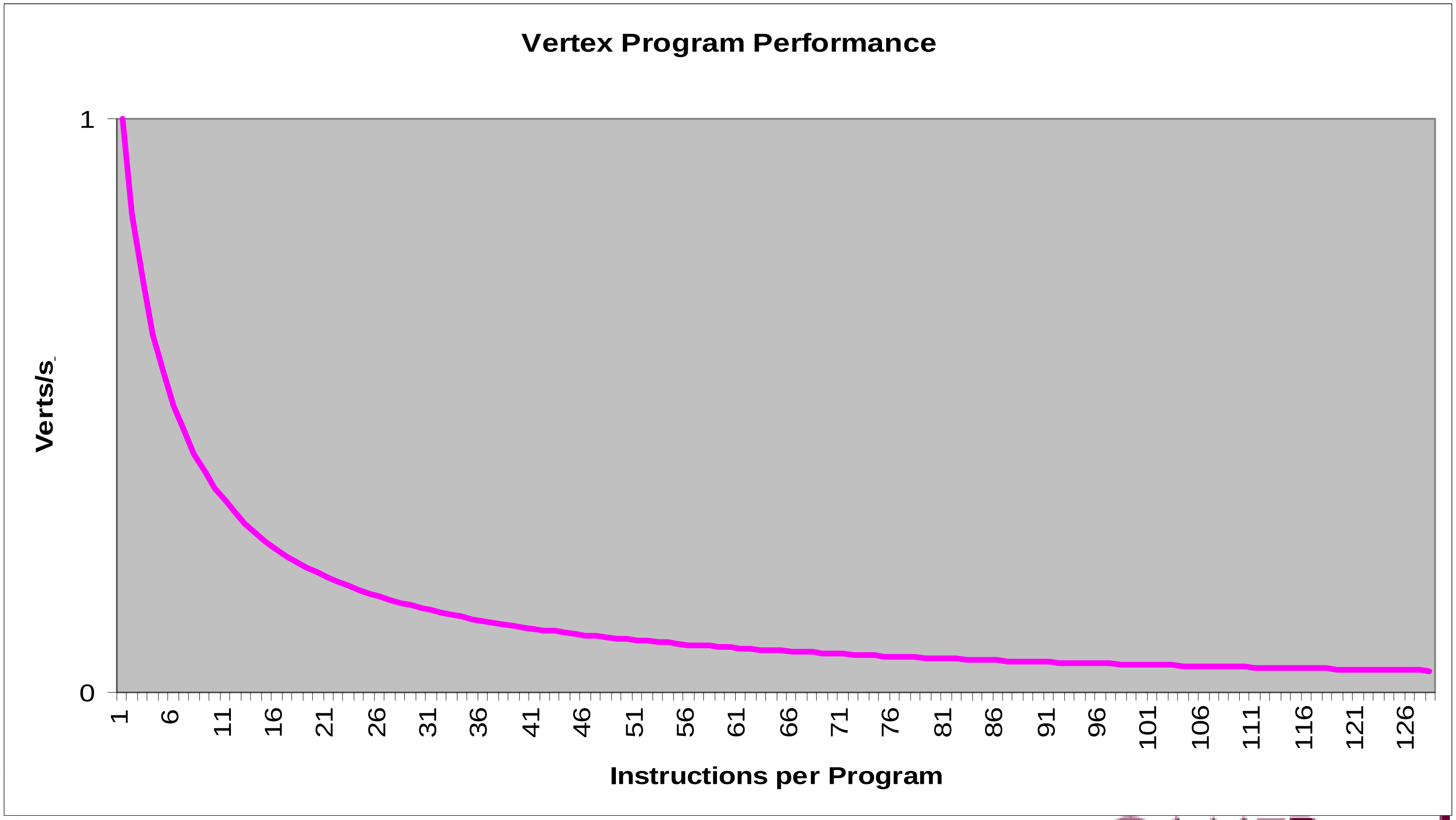
For operations that can be performed with the fixed function pipeline, that should be used, because:

- it will also be hardware accelerated on NV1X
- it will be faster on GeForce3, if lighting-heavy

All other operations should be performed with a vertex program

Vertex Program Performance

Vertices per second = ~ clock_rate/program_length



Fixed Function Pipeline Performance

GeForce3 fixed function pipeline performance will scale in relation to NV1X in most cases

Two sided lighting is an exception – it is entirely hardware accelerated in GeForce3

Performance penalty for two sided lighting vs. one sided is dependent upon number of lights, texgen, and other variables, but should be almost free for “CAD lighting” (1 infinite light/viewer)

Texturing

Hard to optimize

- **Speed vs. quality – make it a user settable option**
- **Pick the right filtering modes**
- **Pick the right texture formats**
- **Pick the right texture shader functions**
- **Pick the right texture blend functions**
- **Load the textures efficiently**
- **Manage your textures effectively**
- **Use multitexture to reduce number of passes**
- **Sort by texture to reduce state changes!**

Texture Filtering and Formats

Still use bilinear mipmapped filtering for best speed when multitexturing

LINEAR_MIPMAP_LINEAR is as fast as bilinear when only a single 2D texture is being used on GeForce3

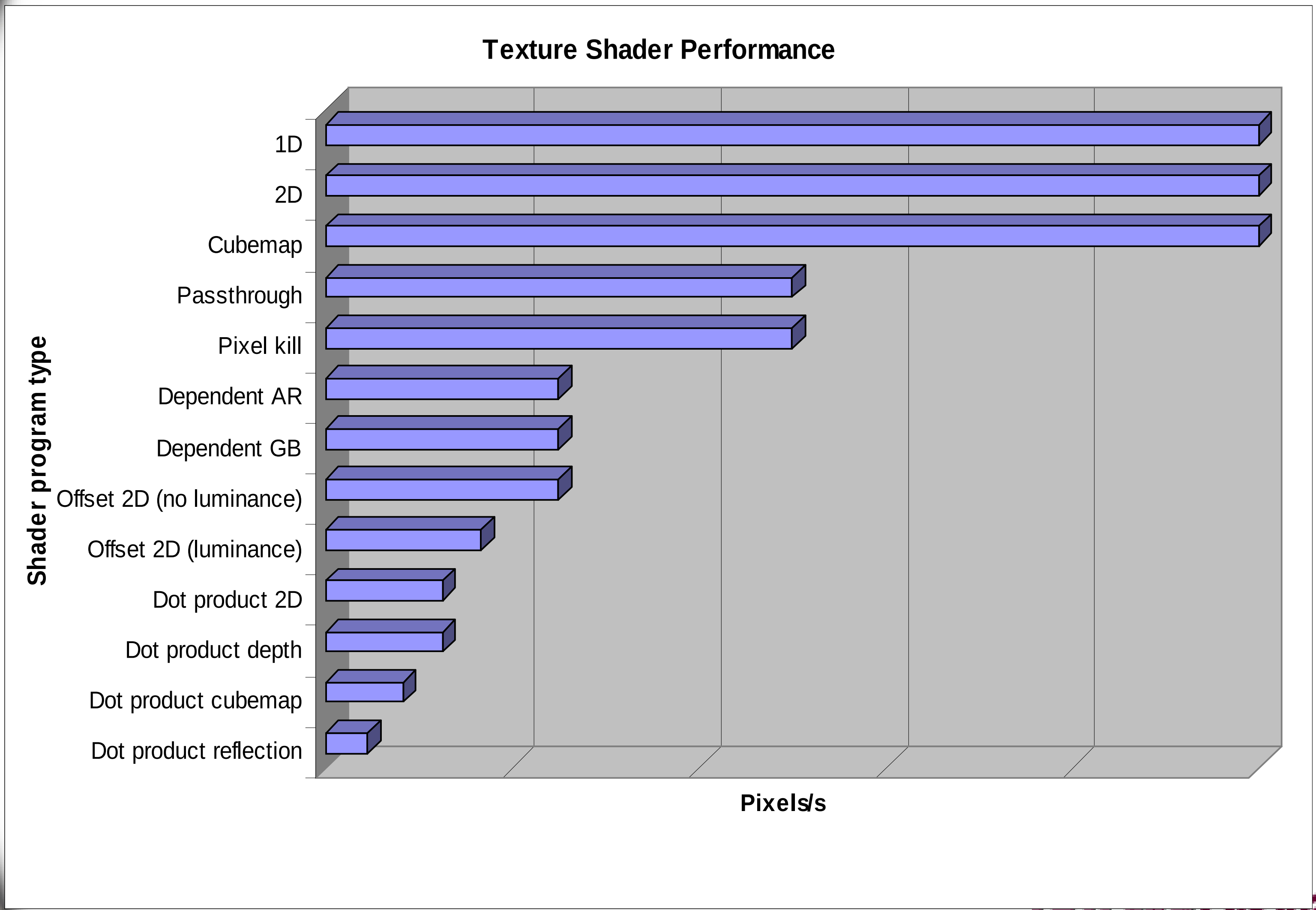
Use anisotropic filtering for best image fidelity

- Will not be free, but penalty only occurs when necessary, not on all pixels
- Higher levels will be slower

Use texture compression, if at all practical

Use single/dual component formats when possible

Texture Shader Programs



Texture Blending

Use register combiners for most flexibility

Final combiner is always “free” (i.e. full speed)

General combiners can slow things down:

- 1 or 2 can be enabled for free
- 3 or 4 can run at one half maximum speed
- 5 or 6 can run at one third maximum speed
- 7 or 8 can run at one fourth maximum speed

Maximum speed may not be possible due to choice and number of texture shaders (or other factors)

As such, number of enabled general combiners will likely not be the bottleneck in most cases

Texture Downloads and Copies

Texture downloads

- **Always use glTexSubImage2D rather than glTexImage2D, if possible**
- **Match external/internal formats**
- **Use texture compression, if it makes sense**
- **Use SGIS_generate_mipmap instead of gluBuild2DMipmaps**

Texture copies

- **Match internal format to framebuffer (e.g. 32-bit desktop to GL_RGBA8)**
- **Will likely have a facility to texture by reference soon, obviating the copy**

Other Fragment Operations

Render coarsely from front-to-back

- Minimizes memory bandwidth, which is often the performance bottleneck
- Particularly important on GeForce3, which has special early Z-culling hardware

Avoid blending

- Most modes drop fill rates
- Try to collapse multiple passes with multitexture

Avoid front-and-back rendering like the plague!

- Instead, render to back, then to front

Questions, comments, feedback

- **John Spitzer, jspitzer@nvidia.com**
- **www.nvidia.com/developer**