GDC Tutorial:
Advanced OpenGL Game Development

Maximizing OpenGL Performance for GPUs

March 8, 2000

John F. Spitzer
john.spitzer@nvidia.com
Developer Relations Engineer
NVIDIA Corporation

Potential Bottlenecks

They mirror the OpenGL pipeline

- Data transfer from application to GPU
- Vertex lighting
- Texture coordinate generation (TexGen), other per-vertex or per-triangle operations
- Texture mapping
- Other per-fragment operations

What Else Can Slow You Down?

Pixel operations

- glDrawPixels, glReadPixels, glCopyPixels
- Texture image downloads

Other stuff

- Inefficient context management
- Inefficient state management

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 on graphics subsystem
- Typically very fast

Vertex Arrays

GAMEDevelopers CONFERENCE 2000

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 Extension

GAMEDevelopers CONFERENCE 2000

Eliminates the double copy

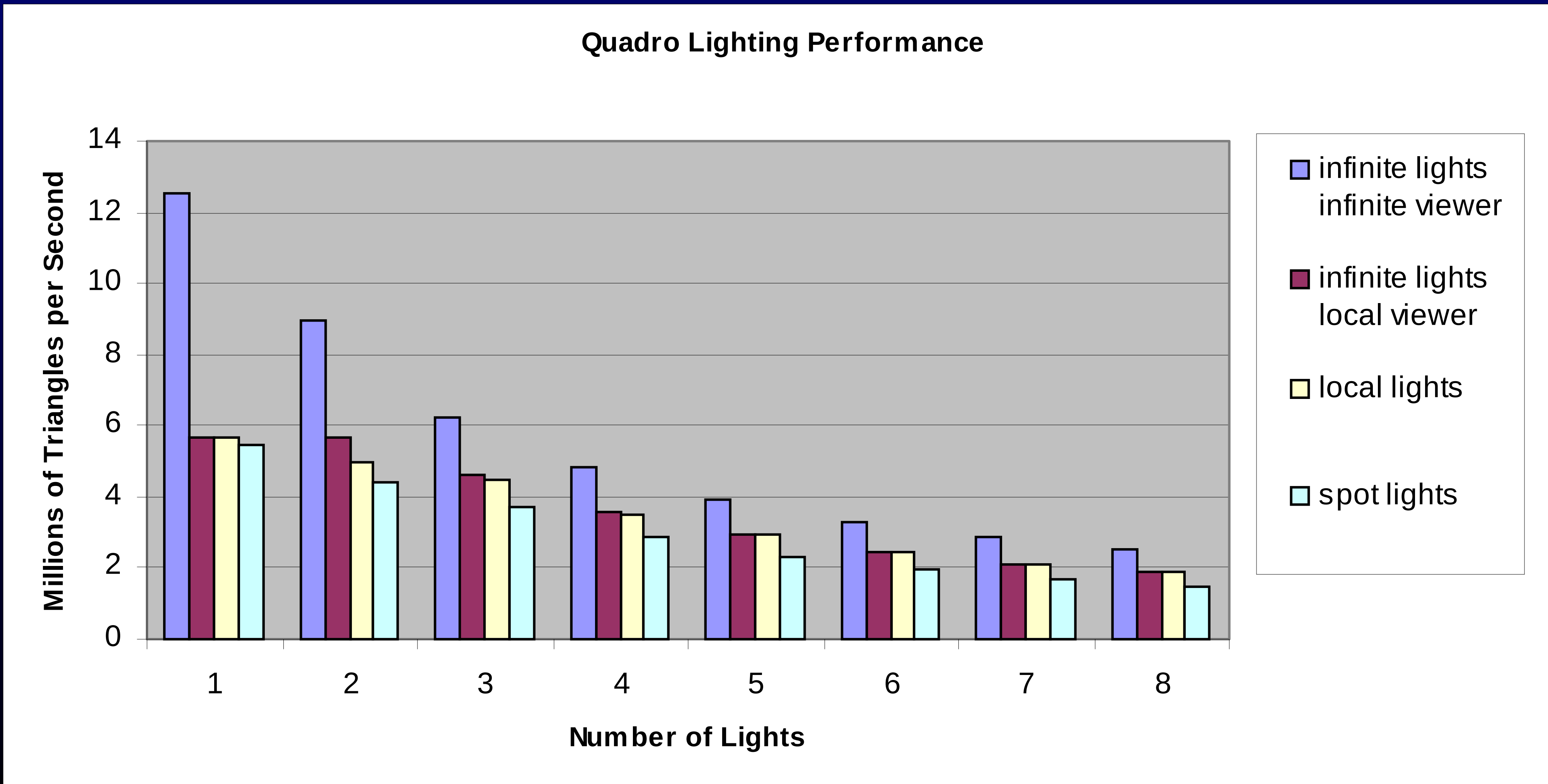
- Analogous to Direct3D vertex buffers
- wglAllocateMemoryNV returns a chunk of AGP or video memory depending upon the user's needs
- Application manages AGP/video memory itself
- Video memory is fastest, but most restrictive
- AGP is often just as fast, but must be used with care
- AGP memory is uncached – write to it sequentially to maximize write combining (and, thus, memory bandwidth)

Vertex Lighting

Gaining speed and popularity

- Not terribly fast when performed on CPU
- Very fast when performed on GPU!
- Different types have different performance characteristics
- Some lighting modes cost more than others
- 8 simultaneous lights allowed, minimize for best performance

Vertex Lighting Performance



Light Types and Modes

Differing performance characteristics

- Infinite lights fastest with infinite viewer, since half angle vector need not be recomputed for every vertex
- Local lights are more computationally expensive, but often offer some features for “free”:
  - Local viewer
  - Attenuation
- Color material is typically not free
- Two-sided lighting is almost never free

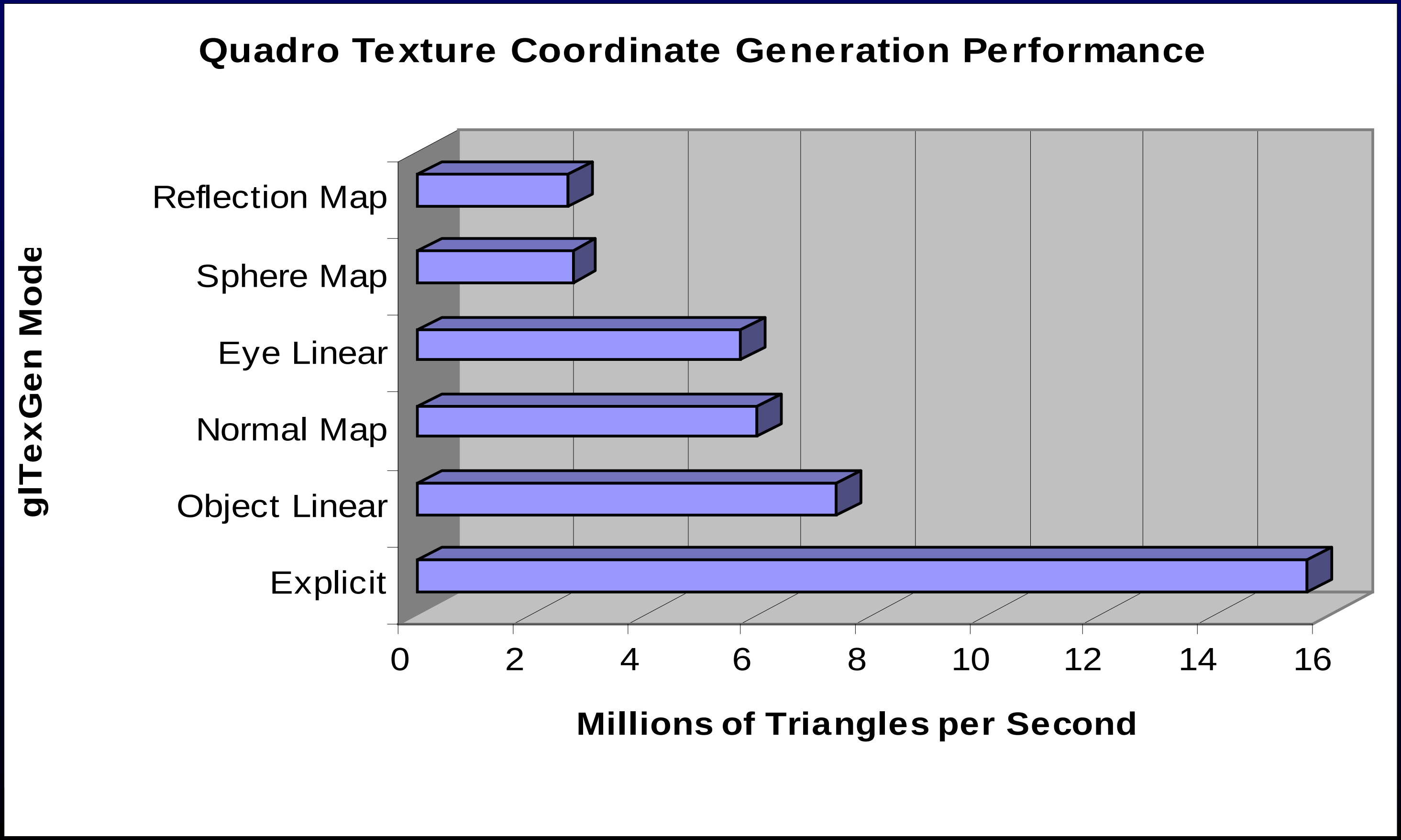
Number of Lights to Enable

Minimize lights to maximize performance

- More is not necessarily better
- Saturation often occurs with over four active lights
- Quickly calculate distance squared from each object to each local light to determine whether it should be enabled or not
- Dot products can be used to determine whether an object is in the cone of a spot light or not
- If an object has more than four lights, disable the furthest
- You might have to reduce the size of your objects

Texture Coordinate Generation

TexGen often hardware accelerated, but not free



Texture Coordinate Transformation

Use the texture matrix wisely

- Like TexGen, the texture matrix is often hardware accelerated, but not free
- If texture coordinates are not changed on a per-frame basis, it might be better to pre-multiply them
- For calculating projected textures, shadows and so forth, using the texture matrix is encouraged
- If one texture matrix is hardware accelerated, the other (in the case of multitexturing) usually is too

Other Vertex/Face Calculations

Clipping

- Performed efficiently on GPU, no need to do yourself
- Per-object view-frustum culling still strongly encouraged

Culling

- Backface culling can cut fillrate requirements in half
- Enable whenever feasible (i.e. on closed objects)

Polygon offset

- Very useful for hidden line, decals, appliqués
- Typically little or no performance overhead

Other Vertex/Face Calculations (continued)

GAMEDevelopers CONFERENCE 2000

Dual matrix vertex weighting

- Great for doing simple skinning
- Not free, but can be done much faster than on CPU

Fog calculations

- Not free, but probably faster than the CPU
- Different modes usually have the same performance
- Can calculate your own, if you want

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 functions
- Load the textures efficiently
- Manage your textures effectively
- Use multitexture

Texture Filtering/Formats

GAMEDevelopers CONFERENCE 2000

Highest quality, not necessarily highest speed

- Use GL_LINEAR_MIPMAP_LINEAR filtering
- Optionally use anisotropic filtering (only 10% hit on GeForce)
- Use 24-bit or 32-bit internal texture formats

Highest speed, not necessarily highest quality

- Use bilinear mipmapping (GL_LINEAR_MIPMAP_NEAREST)
- Use packed pixel 16-bit internal (and external) formats
- Use S3TC texture compression, if available
- Use single/dual component formats, if practical

Maximizing Texture Download Performance

Very important if you have a lot of textures

- Use glTexSubImage2D rather than glTexImage2D
- Match external/internal formats
- Use texture compression, if available
- If using copy_texture, match texture internal format to that of framebuffer (e.g. 32-bit desktop to GL_RGBA8)
- If using paletted textures, share the palette between multiple textures

Other Texture Tips

Texture Binds

- Minimize these, possibly by sorting objects by texture ID

Multitexture

- Collapse two passes into one by using multitexture
- Use register combiners extension to reduce number of passes
  - Allows much more flexibility than standard OpenGL modes
  - Permits separate RGB and Alpha processing
- Use only one general register combiner, if possible

Other Fragment Operations

Polygon stipple

- May be fast by itself, but not in conjunction with texturing

Specular color summation and fog application

- Free on many systems

Testing operations (scissor, alpha, stencil, depth)

- Testing for scissor/alpha usually free
- Depth/stencil can require a read/modify/write at some cost
- Render from front-to-back to minimize writing to depth buffer

Other Fragment Operations (continued)

Blending

- Most modes cut fill rates in half, because of read/modify/write
- Use only where necessary

Color logical operation (LogicOp)

- Can make system default to software rendering
- Avoid anything but default mode (GL_COPY)

Pixel Operations

Blitting between system and framebuffer memory

- Keep it simple – no weird formats or types
- No pixel maps, shifts, biases, or other operations
- On almost all systems, RGB/RGBA unsigned byte formats are somewhat optimized
- Other formats, which more closely match native framebuffer configuration, may be faster
- Avoid reading/writing depth buffer – instead, use GL_KTX_buffer_region extension for incremental updates

Context Switching

Do it wisely, or it will cost you

- Context switching is expensive, keep it to a minimum
- Try “faking” multiple windows by setting the viewport and scissor rectangle to restrict drawing to that “sub-window”
- If multiple windows are necessary, try re-using a single context by binding it to separate windows

General Performance Concerns

State management

- Try to avoid setting redundant state (this is common)
- Minimize state changes by sorting in order of attributes, if possible (starting with most expensive to change)

Antialiasing

- Be sure that the system can support it in hardware
- Test at run-time to determine if it's fast enough, and disable if it's not

Identifying Bottlenecks

Start with your application

- Use a profiling tool, like Intel's VTUNE, to identify parts of your code where the most time is being spent
- Expect a graphics-intensive application (like a game) to spend a good amount of time in glBegin, glEnd, glFinish, etc.

Graphics bottlenecks

- Make the window smaller
- Assuming you don't have a dynamic LOD selector, your performance will go up if raster bound, not if geometry bound

Know What's Fast

GAMEDevelopers CONFERENCE 2000

Before you start coding

- Use a performance benchmark, like SPECglperf, or a custom-written benchmark
- Investigate your target platform(s)
- Determine which modes are fast, and which aren't

At runtime

- Build in a mini-benchmark to test performance
- Select rendering paths depending upon performance
- Allows scalability across many platforms

Questions?