NVIDIA Graphics, Cg, and Transparency

Mark Kilgard
Graphics Software Engineer
NVIDIA Corporation
Outline

• NVIDIA graphics hardware
  – seven years for GeForce + the future

• Cg—C for Graphics
  – the cross-platform GPU programming language

• Depth peeling
  – out-of-order transparency now practical
# Seven Years of GeForce

<table>
<thead>
<tr>
<th>Year</th>
<th>Product</th>
<th>New Features</th>
<th>OpenGL Version</th>
<th>Direct3D Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>GeForce 256</td>
<td>Hardware transform &amp; lighting, configurable fixed-point shading, cube maps, texture compression, anisotropic texture filtering</td>
<td>1.3</td>
<td>DX7</td>
</tr>
<tr>
<td>2001</td>
<td>GeForce3</td>
<td>Programmable vertex transformation, 4 texture units, dependent textures, 3D textures, shadow maps, multisampling, occlusion queries</td>
<td>1.4</td>
<td>DX8</td>
</tr>
<tr>
<td>2002</td>
<td>GeForce4 Ti 4600</td>
<td>Early Z culling, dual-monitor</td>
<td>1.4</td>
<td>DX8.1</td>
</tr>
<tr>
<td>2003</td>
<td>GeForce FX</td>
<td>Vertex program branching, floating-point fragment programs, 16 texture units, limited floating-point textures, color &amp; depth compression</td>
<td>1.5</td>
<td>DX9</td>
</tr>
<tr>
<td>2004</td>
<td>GeForce 6800 Ultra</td>
<td>Vertex textures, structured fragment branching, non-power-of-two textures, generalized floating-point textures, floating-point texture filtering and blending, dual-GPU</td>
<td>2.0</td>
<td>DX9c</td>
</tr>
<tr>
<td>2005</td>
<td>GeForce 7800 GTX</td>
<td>Transparency antialiasing, quad-GPU</td>
<td>2.0</td>
<td>DX9c</td>
</tr>
<tr>
<td>2006</td>
<td>GeForce 7900 GTX</td>
<td>Single-board dual-GPU, process efficiency</td>
<td>2.1</td>
<td>DX9c</td>
</tr>
</tbody>
</table>
2006: the GeForce 7900 GTX board

- 512MB/256-bit GDDR3
- 1600 MHz effective
- 8 pieces of 8Mx32

Features:
- SLI Connector
- DVI x 2
- sVideo
- TV Out

- 16x PCI-Express

- 512MB/256-bit GDDR3
  - 1600 MHz effective
  - 8 pieces of 8Mx32
2006: the GeForce 7900 GTX GPU

- 278 million transistors
- 650 MHz core clock
- 1,600 MHz GDDR3 effective memory clock
- 256-bit memory interface

Notable Functionality
- Non-power-of-two textures with mipmaps
- Floating-point (fp16) blending and filtering
- sRGB color space texture filtering and frame buffer blending
- Vertex textures
- 16x anisotropic texture filtering
- Dynamic vertex and fragment branching
- Double-rate depth/stencil-only rendering
- Early depth/stencil culling
- Transparency antialiasing
2006: GeForce 7950 GX2, SLI-on-a-card

Two GeForce 7 Series GPUs
500 Mhz core

1 GB video memory
512 MB per GPU
1,200 Mhz effective

Effective 512-bit memory interface!

DVI x 2
sVideo
TV Out

Sandwich of two printed circuit boards

16x PCI-Express
GeForce Peak
Vertex Processing Trends

Assumes Alternate Frame Rendering (AFR) SLI Mode

rate for trivial 4x4 vertex transform
exceeds peak setup rates—allows excess vertex processing

Vertex units: 1, 1, 2, 3, 6, 8, 8, 2x8

Millions of vertices per second: 0, 600, 1,000, 1,600, 2,000, 2,500

Vertex Transformation Trends
Assumes Alternate Frame Rendering (AFR) SLI Mode
GeForce Peak
Memory Bandwidth Trends

Two physical 256-bit memory interfaces

Gigabytes per second

128-bit interface  256-bit interface

Raw bandwidth

Effective raw bandwidth with compression

Expon.
(Effective raw bandwidth with compression)

Expon.
(Raw bandwidth)
Effective GPU Memory Bandwidth

- **Compression schemes**
  - Lossless depth and color (when multisampling) compression
  - Lossy texture compression (S3TC / DXTC)
  - Typically assumes 4:1 compression

- **Avoid useless work**
  - Early killing of fragments (Z cull)
  - Avoid useless blending and texture fetches

- **Very clever memory controller designs**
  - Combining memory accesses for improved coherency
  - Caches for texture fetches
NVIDIA Graphics Core and Memory Clock Rates

DDR memory transition—memory rates double physical clock rate
GeForce Peak
Texture Fetch Trends

assuming no texture cache misses

Texture units  2×4  2×4  2×4  2×4  16  24  24  2×24
GeForce Peak
Depth/Stencil-only Fill

assuming no read-modify-write
double speed depth-stencil only

Millions of depth/stencil pixel updates per second
GeForce Transistor Count and Semiconductor Process

More performance with fewer transistors: 
Architectural & process efficiency!

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>180</td>
<td>180</td>
<td>150</td>
<td>130</td>
<td>130</td>
<td>110</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

 Millions of transistors

600
500
400
300
200
100
0
GeForce 7900 GTX Parallelism

8 Vertex Engines

Z-Cull → Triangle Setup/Raster

Shader Instruction Dispatch

24 Fragment Shaders

Fragment Crossbar

16 Raster Operation Pipelines

Memory Partition

Memory Partition

Memory Partition

Memory Partition
<table>
<thead>
<tr>
<th>Hardware Unit</th>
<th>GeForce FX 5900</th>
<th>GeForce 6800 Ultra</th>
<th>GeForce 7900 GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>3</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Fragment</td>
<td>4+4</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>2nd Texture Fetch</td>
<td>4+4</td>
<td>16+16</td>
<td>16+16</td>
</tr>
<tr>
<td>Raster Color</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raster Depth</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2005: Comparison to CPU

Pentium Extreme Edition 840
- 3.2 GHz Dual Core
- 230M Transistors
- 90nm process
- 206 mm\(^2\)
- 2 x 1MB Cache
- 25.6 GFlops

GeForce 7800 GTX
- 430 MHz
- 302M Transistors
- 110nm process
- 326 mm\(^2\)
- 313 GFlops (shader)
- 1.3 TFlops (total)
2006: Comparison to CPU

Intel Core 2 Extreme X6800
- 2.93 GHz Dual Core
- 291M Transistors
- 65nm process
- 143 mm$^2$
- 4MB Cache
- 23.2 GFlops

GeForce 7900 GTX
- 650 MHz
- 278M Transistors
- 90nm process
- 196 mm$^2$
- 477 GFlops (shader)
- 2.1 TFlops (total)
Giga Flops Imbalance

Theoretical programmable
IEEE 754 single-precision
Giga Flops

Intel Core 2 Extreme X6800

GeForce 7900 GTX
Future NVIDIA GPU directions

• DirectX 10 feature set
  – Massive graphics functionality upgrade

• Language and tool support
  – Performance tuning and content development

• Improved GPGPU
  – Harness the bandwidth & Gflops for non-graphics

• Multi-GPU systems innovation
  – Next-generation SLI
DirectX 10-class GPU functionality

- Generalized programmability, including
  - Integer instructions
  - Efficient branching
  - Texture size queries, unfiltered texel fetches, & offset fetches
  - Shadow cube maps for omni-directional shadowing
  - Sourcing constants from bind-able buffer objects

- Per-primitive programmable processing
  - Emits zero or more strips of triangles/points/lines
  - New line and triangle adjacency primitives
  - Output to multiple viewports and buffers
Per-primitive processing example:
Automatic silhouette edge rendering

New triangle adjacency primitive = 3 conventional vertices + 3 vertices for adjacent triangles
More DirectX 10-class GPU functionality

• Better blending
  – Improved blending control for multiple draw buffers
  – sRGB and 32-bit floating-point framebuffer blending
• Streamed output of vertex processing to buffers
  – Render to vertex array
• Texture improvements
  – Indexing into an “array” of 2D textures
  – Improved render-to-texture
  – Luminance-alpha compressed formats
  – Compact High Dynamic Range texture formats
  – Integer texture formats
  – 32-bit floating-point texture filtering
Uses of DirectX 10 functionality

- Deep Waves
- Sparkling Sprites
- GPU Fluid Simulation
- GPU Marching Cubes
- Table-free Noise
- Styled Line Drawing
- Deformable Collisions
- GPU Cloth
DirectX 10-class functionality parity

- **Feature parity**
  - DirectX 10-class features available via OpenGL
  - Cross API portability of programmable shading content through Cg

- **Performance parity**
  - 3D API agnostic performance parity on all Windows operating systems

- **System support parity**
  - Linux, Mac, FreeBSD, Solaris
  - Shared code base for drivers
Multi-GPU Support

• Original SLI was just the beginning
  – Quad-SLI
  – SLI support infuses all NVIDIA product design and development

• New SLI APIs for application-control of multiple GPUs

• SLI for notebooks
  – Better thermals and power
<table>
<thead>
<tr>
<th>Hardware Unit</th>
<th>GeForce 7900 GTX</th>
<th>GeForce 7900 GTX Quad SLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Cores</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Fragment Cores</td>
<td>24</td>
<td>96</td>
</tr>
<tr>
<td>Raster Color Cores</td>
<td>16+16</td>
<td>64+64</td>
</tr>
<tr>
<td>Raster Depth Cores</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cg: C for Graphics
Cg: C for Graphics

• Cg as it exists today
  – High-level, inspired mostly by C
  – Graphics focused
    • API-independent
      – GLSL tied to OpenGL; HLSL tied to Direct3D; Cg works for both
    • Platform-independent
      – Cg works on PlayStation 3, ATI, NVIDIA, Linux, Solaris, Mac OS X, Windows, etc.

• Production language and system
  – Cg 1.5 is part of 3D content creation tool chains
  – Portability of Cg shaders is important
Evolution of Cg

General-purpose languages

Graphics Application
Program Interfaces
Shading Languages

C (AT&T, 1970's)

C++ (AT&T, 1983)

Java (Sun, 1994)

IRIS GL (SGI, 1982)

OpenGL (ARB, 1992)

Reality Lab (RenderMorphics, 1994)

Direct3D (Microsoft, 1995)

RenderMan (Pixar, 1988)

PixelFlow Shading Language (UNC, 1998)

Real-Time Shading Language (Stanford, 2001)

Cg / HLSL (NVIDIA/Microsoft, 2002)
Cg 1.5

- Current release of Cg
  - Supports Windows, Linux, Mac (including x86 Macs) + now Solaris
  - Shader Model 3.0 profiles for Direct3D 9.0c
  - Matches Sony’s PlayStation 3 Cg support
  - Tool chain support: FX Composer 2.0

- New functionality
  - Procedural effects generation
  - Combined programs for multiple domains
  - New GLSL profiles to compile Cg to GLSL

- Improved compiler optimization
FX Composer for Cg shader authoring

- Shaders are assets
  - Portability matters
- So express shaders in a multi-platform, multi-API language
  - That’s Cg
Future: Modernizing Cg

• Opportunity to re-think the Cg language
  – Experience-driven
  – Shader writing was programming-in-the-small
    • But not anymore!
    • Provide better abstraction mechanisms
  – Must be backward compatible

• Challenge: Instead of inventing yet-another shading language-specific keyword, think how a C++ programmer express the feature
  – Think templates and classes
Cg Directions

• DirectX 10-class feature support
  – Primitive (geometry) programs
  – Constant buffers
  – Interpolation modes
  – Read-write index-able temporaries
  – New texture targets: texture arrays, shadow cube maps

• Incorporate established C++ features, examples:
  – Classes
  – Templates
  – Operator overloading
  – But not runtime features like new/delete, RTTI, or exceptions
Why C++?

- Already inspiration for much of Cg
  - Think of Cg’s first-class vectors simply as classes
- Functionality in C++ is well-understood and popular
- C++ is biased towards compile-time abstraction
  - Rather than more run-time focus of Java and C#
  - Compile-time abstraction is good since GPUs lack the run-time support for heaps, garbage collection, exceptions, and run-time polymorphism
Logical Programmable Graphics Pipeline

3D Application or Game

3D API: OpenGL or Direct3D Driver

3D API Commands

GPU Command & Data Stream

GPU Front End

Primitive Assembly

Programmable Vertex Processor

Programmable Fragment Processor

Rasterization & Interpolation

Transformed Vertices

Assembled Polygons, Lines, and Points

Rasterized Pre-transformed Fragments

Transformed Fragments

Pixel Location Stream

Pixel Updates

Framebuffer

Pre-transformed Vertices

Vertex Index Stream

Assembled Polygons, Lines, and Points

Transformed Polygons, Lines, and Points

CPU – GPU Boundary

Program vertex and fragment domains
Future Logical Programmable Graphics Pipeline

3D Application or Game

3D API: OpenGL or Direct3D Driver

3D API Commands

GPU Command & Data Stream

GPU Front End

Pre-transformed Vertices

Transformed Vertices

Input assembled Polygons, Lines, and Points

Programmable Vertex Processor

Programmable Primitive Processor

Primitive Assembly

Rasterization & Interpolation

Raster Operations

Output assembled Polygons, Lines, and Points

Transformed Pre-transformed Fragments

Rasterized Fragments

Programmable Fragment Processor

Pixel Location Stream

Transformed Fragments

Pixel Updates

Framebuffer

New per-primitive “geometry” programmable domain

CPU – GPU Boundary

SIGGRAPH 2006
Pass Through Geometry Program Example

flatColor initialized from constant buffer 6

BufferInit<float4,6> flatColor;

Primitive’s attributes arrive as “templated” attribute arrays

TRIANGLE void passthru(AttribArray<float4> position : POSITION,
AttribArray<float4> texCoord : TEXCOORD0)
{
    flatAttrib(flatColor:COLOR);
    for (int i=0; i<position.length; i++) {
        emitVertex(position[i], texCoord[i]);
    }
}

Length of attribute arrays depends on the input primitive mode, 3 for TRIANGLE

Makes sure flat attributes are associated with the proper provoking vertex convention

Bundles a vertex based on parameter values and semantics
Depth peeling

- Brute force order-independent transparency algorithm [Everitt 2001]
- Approach
  - Render transparent objects repeatedly
    - Each pass peels successive color layer using dual-depth buffers
    - Composite peeled layers in order
- Caveats
  - Typically makes “thin film” assumption
  - No refraction or scattering
Transparency: The Good vs. the Ugly

Correct ordered depth peeling

Wrong unordered blending
Peeled layer visualization

Fragment count per layer

Composite layers

SIGGRAPH 2006
Another example: The Good vs. the Ugly

Correct ordered depth peeling

Wrong unordered blending
Another peeled layer visualization

Fragment count per layer

composite layers
Real-time transparency demo
Depth peeling: How it works

- Conventional depth buffer after rendering
  - Color buffer has color of closest fragment
  - Depth buffer has depth of closest fragment
- Re-use the depth buffer!
  - Make depth buffer into a shadow map
  - Clear a 2\textsuperscript{nd} depth buffer
  - Discard fragments if fragment depth is closer than corresponding pixel’s depth in shadow map
  - Save color buffer for compositing
  - Repeat this with current depth buffer to peel another layer
- Prior depth buffer works as “back stop” for next pass
  - Discard fragments closer or as close as last pass for every pixel
Optimizations for real-time depth peeling

• Optimizations
  – Render-to-texture to ping-pong between 2 back stop depth buffers (no depth buffer copies)
  – Shadow mapping for 2nd read-only “back stop” depth buffer
  – Asynchronous occlusion queries to determine fragments still being peeled
  – Threshold to stop peeling
  – Smart front-to-back (“under”) compositing

• **Result:** 120+ fps depth peeling for peeling and composting up to 14 layers as needed
So is transparency a solved problem?

- Bounding the error
  - Assume a lower bound on opacity of objects
    - …and an upper bound on layers peeled
  - $\text{worstCaseError} = (1 - \text{minOpacity})^{\text{maxLayers}}$
    - **Example:** 20% min. opacity with 15 peeled layers
      - Remaining potential transparency could be off by just 3.5% if looking through 15 layers of 20% opacity (worst possible case)
  - Typical cases are much, much better than that
    - As occlusion query can provide a count of mis-ordered pixels
- Arguably **could be** for a certain class of transparency
  - Mostly opaque scenes with thin film transparency like windows
  - CAD models made of virtual Jell-O®
Conclusions

- NVIDIA GPUs
  - Expect more compute and bandwidth increases >> CPUs
  - DirectX 10 = large functionality upgrade for graphics
- Cg, the only cross-API, multi-platform language for programmable shading
  - Think shaders as content, not GPU programs trapped inside applications
- Depth peeling
  - Harnessing the GPU’s brute force for transparency