Beyond Programmable Shading: In Action

SIGGRAPH 2008
5/19/2008

Course Organizers:
Aaron Lefohn, Intel
Mike Houston, AMD

Course Speakers:
Aaron Lefohn    Intel
Mike Houston    AMD
David Luebke    NVIDIA
Jon Olick       Id Software
Fabio Pellacini Dartmouth College

Speaker Contact Info:
Aaron Lefohn    Intel
2700 156th Ave NE, Suite 300
Bellevue, WA 98007
aaron.lefohn@intel.com

Mike Houston    AMD
4555 Great America Parkway
Suite 501
Santa Clara, CA 95054
Michael.Houston@ amd.com

Dave Luebke    NVIDIA
1912 Lynchburg Dr.
Charlottesville, VA 22903
dlebke@nvidia.com

Jon Olick       Id Software
3819 Towne Crossing #222
Mesquite, TX 75150
zelexi@gmail.com

Fabio Pellacini Dartmouth College
6211 Sudikoff Lab
Hanover, NH 03755
fabio@cs.dartmouth.edu
Beyond Programmable Shading: In Action

Course Description:
This second course in a series will demonstrate case studies of combining traditional rendering API usage with advanced parallel computation from game developers, researchers, and graphics hardware vendors. There are strong indications that the future of interactive graphics programming is a model more flexible than today’s OpenGL/Direct3D pipelines. As such, graphics developers need to have a basic understanding of how to combine emerging parallel programming techniques and more flexible graphics processors with the traditional interactive rendering pipeline. Each case study includes a live demo and discusses the mix of parallel programming constructs used, details of the graphics algorithm, and how the rendering pipeline and computation interact to achieve the technical goals.

Intended Audience:
We are targeting researchers and engineers interested in investigating advanced graphics techniques using parallel programming techniques on many-core GPU and CPU architectures, as well as graphics and game developers interested in integrating these techniques into their applications.

Prerequisites:
Attendees are expected to have experience with a modern graphics API (OpenGL or Direct3D), including basic experience with shaders, textures, and framebuffers and/or background with parallel programming languages. Some background with parallel programming on CPUs or GPUs is useful but not required as an overview will be provided in the course. Attendees are strongly encouraged to attend the first course in this series, “Beyond Programmable Shading: Fundamentals.”

Level of difficulty: Advanced

Special presentation requirements:
Several speakers will bring their own demo machines for use in the course.

Speakers:
- Aaron Lefohn, Intel
- Mike Houston, AMD
- David Luebke, NVIDIA
- Jon Olick, Id Software
- Fabio Pellacini, Dartmouth College
“Beyond Programmable Shading: In Action”

- Introduction: Mike Houston: 15 min
  o Brief recap of course 1
    ▪ From programmable shading to fully programmable graphics
    ▪ List of programming environment possibilities
    ▪ Recap of benefits over using the traditional graphics pipeline
    ▪ Teaser pictures for case studies

  (NOTE: All case studies will discuss the graphics algorithms, the mix of parallel computation being used to achieve the effect(s), as well as performance and of course pretty pictures)

- Research Case Study
  o Interactive Cinematic Lighting: Fabio Pellacini: 30 min
    ▪ The fake: pre-computing your way to interactive global illumination on a GPU
    ▪ The real deal: no-pre-computation global illumination in 5 seconds with CPUs + GPU
    ▪ What it will take to do this 500 times faster

- Games Case Studies
  o Current Generation Parallelism in Games: Jon Olick (Sony/Id): 30 min
    ▪ Offloading work from the GPU to many CPU cores
    ▪ Triangle culling, progressive meshes, displacement maps
    ▪ Geometry compression / decompression: indices, vertices
    ▪ Tightly-coupled CPU-GPU synchronization techniques
  o Next Generation Parallelism in Games: Jon Olick (Id): 30 min
    ▪ Why ray casting? What are the advantages over rasterization?
    ▪ Does it add end-user value? Risks?
    ▪ Efficient implementations, control flow, data structures

- Break, 15 minutes

- Graphics Hardware Vendor Case Studies
  o Each of the hardware vendor talks will give a case study covering several advanced graphics algorithms that require a mix of data- and/or task-parallel computation as well as the traditional graphics pipeline. Example topics may include, but are not limited to:
    ▪ Interactive global illumination
    ▪ Building/using irregular data structures during interactive rendering
    ▪ Combining real-time physics and rendering
    ▪ Combining ray tracing and rasterization
  o Each case study will describe the graphics algorithms, the data structures used/built, the parallel algorithms and programming tools used, as well as give live demos.
The details of the case studies cannot be revealed at this time due to intellectual property limitations; however, each of the vendors and speakers has a strong reputation of delivering cutting-edge content and algorithm design to the SIGGRAPH community.

- NVIDIA case study: 30 min
- AMD case study: 30 min
- Intel case study: 30 min

Conclusions: Lefohn + All Speakers: 15 min
- Summary of state-of-the-world
- Guesses at the future
- Open Q & A with all speakers

Speaker Biographies

Aaron Lefohn, Ph.D.

Aaron Lefohn is a Senior Graphics Architect at Intel on the Larrabee project. Previously, he designed parallel programming models for graphics as a Principal Engineer at Neoptica, a computer graphics startup that was acquired by Intel in October 2007. Aaron's Ph.D. in Computer Science from the University of California Davis focused on data structure abstractions for graphics processors and data-parallel algorithms for rendering. From 2003 - 2006, he was a researcher and graphics software engineer at Pixar Animation Studios, focusing on interactive rendering tools for artists and GPU acceleration of RenderMan. Aaron was formerly a theoretical chemist and was an NSF graduate fellow in computer science.

Aaron Lefohn
2700 156th Ave NE, Suite 300
Bellevue, WA  98007
425-881-4891
aaron.lefohn@intel.com

Mike Houston, Ph.D.

Mike Houston is a System Architect in the Advanced Technology Development group at AMD in Santa Clara working in architecture design and programming models for parallel architectures. He received his Ph.D. in Computer Science from Stanford University in 2008 focusing on research in programming models, algorithms, and runtime systems for parallel architectures including GPUs, Cell, multi-core, and clusters. His dissertation includes the Sequoia runtime system, a system for programming hierarchical memory machines. He received his BS in
Computer Science from UCSD in 2001 and is a recipient of the Intel Graduate Fellowship.

Mike Houston
4555 Great America Parkway
Suite 501
Santa Clara, CA 95054
408-572-6010
Michael.Houston@amd.com.

David Luebke, Ph.D.

David Luebke is a Research Scientist at NVIDIA Corporation, which he joined after eight years on the faculty of the University of Virginia. He has a Ph.D. in Computer Science from the University of North Carolina and a B.S. in Chemistry from the Colorado College. Luebke's research interests are GPU computing and realistic real-time computer graphics. Recent projects include advanced reflectance and illumination models for real-time rendering, image-based acquisition of real-world environments, temperature-aware graphics architecture, and scientific computation on GPUs. Past projects include leading the book "Level of Detail for 3D Graphics" and the Virtual Monticello museum exhibit at the New Orleans Museum of Art.

David Luebke
1912 Lynchburg Dr.
Charlottesville, VA  22903
434-409-1892
dlebke@nvidia.com

Jon Olick

Jon Olick has been working on games and games technology professionally for the past 9 years, helping to create multi-million selling games such as Medal of Honor: Allied Assault. Currently, Jon is a Programmer and Engineer at id Software. Previously, on the ICE team (a technology group based at Naughty Dog) he played key architecture roles in the design and development of many Sony first and third party tools and technologies for PlayStation 3. This work has already been used in game titles such as Uncharted: Drakes Fortune, Resistance: Fall of Man, Heavenly Sword, NBA 07, Warhawk, MotorStorm and MLB 2007. He is a published author in the book GPU Gems 2.

Jon Olick
3819 Towne Crossing #222
Mesquite, TX 75150
Fabio Pellacini, Ph.D.

Fabio Pellacini is an assistant professor in computer science at Dartmouth College. His research focuses on algorithms for interactive, high-quality rendering of complex environments and for artist-friendly material and lighting design to support more effective content creation. Prior to joining academia, Pellacini worked at Pixar Animation Studios on lighting algorithms, where he received credits on various movie productions. Pellacini received his Laurea degree in physics from the University of Parma (Italy), and his M.S. and Ph.D. in computer science from Cornell University.

Fabio Pellacini
6211 Sudikoff Lab
Hanover, NH  03755
603-646-8710
fabio@cs.dartmouth.edu
Disclaimer about these Course Notes

• The material in this course is bleeding edge
  - Unfortunately, that means we can’t share most of the details with you until SIGGRAPH 2008
  - Several talks are missing from the submitted notes
  - The talks that are included may change substantially

• To address this inconvenience
  - We will post all course notes/slides on a permanent web page, available the first day of SIGGRAPH 2008
Future interactive rendering techniques will be an inseparable mix of data- and task-parallel algorithms and graphics pipelines.
How do we write new interactive 3D rendering algorithms?
Fixed-Function Graphics Pipeline

• Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...

  - Examples
    • Shadow volumes
    • Hidden line removal
    • ...

Beyond Programmable Shading: In Action
Programmable Shading

• Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...
  - Plus: Writing shaders

- Examples
  • Parallax mapping
  • Shadow-mapped spot light
  • ...
Beyond Programmable Shading

- Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...
  - Plus: Writing shaders
    - Plus: Writing data- and task-parallel algorithms
      - Analyze results of rendering pipeline
      - Create data structures used in rendering pipeline
  - Examples
    - Dynamic summed area table
    - Dynamic quadtree adaptive shadow map
    - Dynamic ambient occlusion
    - ...

Beyond Programmable Shading: In Action
“Fast Summed-Area Table Generation and its Applications,”
Hensley et al., Eurographics 2005

“Resolution Matched Shadow Maps,”
Lefohn et al., ACM Transactions on Graphics 2007

“Dynamic Ambient Occlusion and Indirect Lighting,”
Bunnell, GPU Gems II, 2005
Beyond Programmable Shading

• Writing new rendering algorithms means
  - Tricks with stencil buffer, depth buffer, blending, ...
  - Plus: Writing shaders
  - Plus: Writing data- and task-parallel algorithms
    • Analyze results of rendering pipeline
    • Create data structures used in rendering pipeline
  - Plus: Extending, modifying, or creating graphics pipelines

• Examples
  • PlayStation 3 developers creating hybrid Cell/GPU graphics pipelines
    - See upcoming talk from Jon Olick (Id Software)

• Active area of research
Why “Beyond Programmable Shading?”

• **Short answer:**
  - The parallel processors in your desktop machine or game console are now flexible and powerful enough to execute both
    • User-defined parallel programs and
    • Graphics pipelines
  
  • ...All within 1/30th of a second
This Course

- Case studies from game developers, academics, and industry
This Course

• Show-casing new interactive rendering algorithms that result in more realistic imagery than is possible using only the pre-defined DX/OpenGL graphics pipeline by

  • Combining task-, data-, and/or graphics pipeline parallelism,
  • Analyzing intermediate data produced by graphics pipeline,
  • Building and using complex data structures every frame, or
  • Modifying/extending the graphics pipelines
Speakers (in order of appearance)

• Mike Houston, AMD
• Fabio Pellacini, Dartmouth College
• Jon Olick, Id Software
• Dave Luebke, NVIDIA
• Aaron Lefohn, Intel
## Schedule

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
<th>Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intro</strong></td>
<td>1:45 – 2:00</td>
<td>Houston</td>
</tr>
<tr>
<td><strong>Research Case Study</strong></td>
<td>2:00 – 2:30</td>
<td>Pellacini</td>
</tr>
<tr>
<td>- Interactive Cinematic Lighting</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Games Case Studies</strong></td>
<td>2:30 – 3:00</td>
<td>Olick</td>
</tr>
<tr>
<td>- Current Generation Parallelism in Games</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Next Generation Parallelism in Games</td>
<td>3:00 – 3:30</td>
<td>Olick</td>
</tr>
<tr>
<td><strong>&lt;Break&gt;</strong></td>
<td>3:30 – 3:45</td>
<td></td>
</tr>
<tr>
<td><strong>Graphics Hardware Vendor Case Studies</strong></td>
<td>3:45 – 4:15</td>
<td>Luebke</td>
</tr>
<tr>
<td>- NVIDIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- AMD</td>
<td>4:15 – 4:45</td>
<td>Houston</td>
</tr>
<tr>
<td>- Intel</td>
<td>4:45 – 5:15</td>
<td>Lefohn</td>
</tr>
<tr>
<td><strong>Q &amp; A</strong></td>
<td>5:15+</td>
<td></td>
</tr>
</tbody>
</table>
interactive cinematic lighting

fabiopellacini

joint work with M. Hasan (Cornell), K. Bala (Cornell), K. Vidimce et al. (Pixar)
disclaimer: will skip details

ask questions if interested
digital lighting design

user repeatedly set lighting parameters

feedback: run the rendering algorithm
digital lighting design

user repeatedly set lighting parameters

feedback: run the rendering algorithm

goal: interactive feedback
digital lighting design

user repeatedly set lighting parameters

feedback: run the rendering algorithm

goal: intuitive interfaces
“I have started to work in this field more than 15 years ago. I do not understand why, while computers get faster, it takes us more time than before to make an image”

- jc kalache, dp, pixar
cinematic rendering

• high geometric complexity
  \(10^5\) smooth primitives

• high shading complexity
  \(10^3\) shaders with \(10^5\) ops with 5 GBs textures

• high quality antialiasing
  no artifacts with depth-of-field, motion blur

• complex lighting
  indirect illumination, subsurface scattering, environment lighting, ambient occlusion
cinematic rendering

- high geometric complexity
  - $10^5$ smooth primitives
- high shading complexity
  - $10^3$ shaders with $10^5$ ops with 5 GBs textures
- high quality antialiasing
  - no artifacts with depth-of-field, motion blur
- complex lighting
  - indirect illumination, subsurface scattering, environment lighting, ambient occlusion

not interactive
goal: interactive feedback

in complex environments
with high accuracy
cinematic rendering

- high geometric complexity
  - $10^5$ smooth primitives
- high shading complexity
  - $10^3$ shaders with $10^5$ ops with 5 GBs textures
- high quality antialiasing
  - no artifacts with depth-of-field, motion blur
- complex lighting
  - indirect illumination, subsurface scattering, environment lighting, ambient occlusion
\[ L(x \rightarrow \omega_o) = L_e(x \rightarrow \omega_o) + \int_{\Omega} L(x \leftarrow \omega_i) \rho(x, \omega_o \leftarrow \omega_i) d\omega_i \]
\[ L(\mathbf{x} \rightarrow \omega_o) = L_e(\mathbf{x} \rightarrow \omega_o) + \int_{\Omega} L(\mathbf{x} \leftarrow \omega_i) \rho(\mathbf{x}, \omega_o \leftarrow \omega_i) d\omega_i \]

\[ L_i = \sum_j T_{ij} I_j \]
complex lighting as many-lights

lighting simulation converted to “virtual” point lights

[image courtesy of B. Walter]
\[ L = \sum_j T_j I_j \]
\[ L = T \cdot I \]

- ~100k lights
- ~1M pixels
$L = T \cdot I$

$size(T) \approx 10^{11}$
computing one column

lpics images

~ 30 m

~ 0.1 s

[SIGGRAPH 2005a] [SIGGRAPH 2005b]
method:

lpics
images
cache
approx.
data-parallel
hardware
results: artist at work

lpics
images

one column at a time
motion blur and depth of field

lightspeed images

[Regan-Kelley et al., SIGGRAPH 2007]
impact: interactivity for cinematic lighting changes artists workflow
lessons learned

• interactive cinematic lighting is possible
• interactive algorithms mostly match offline
• hard to match: shader as content is bad!
• render-dependent look definition
\[ L = \sum_{j} T_j I_j \]

brute force too slow

10 min 13 min 20 min
cache \[ T' \approx T \]

indirect illumination

direct on virtual lights

[SIGGRAPH 2006]
method:

precompute (sparsely) matrix
compress by signal processing
data-parallel sparse multiply
\[ L = T \cdot I \]
\[ \text{size}(T) \approx 10^{11} \]

\[ L' = T' \cdot I \approx L \]
\[ T' \approx T \rightarrow L' = T' \cdot I \approx L \]
coherence $\rightarrow$ sparse light vector

method: *impose* a wavelet basis, by clustering, then lossily compress
coherence $\rightarrow$ sparse matrix

method: impose same wavelet basis for rows
construct sparse images for each columns
results

11.4 - 18.7 fps

precomputation: 1.6 h  107k polys
results

8.5 - 25.8 fps

precomputation: 2.5 h  
2.1M polys
impact: highest quality interactive illumination for lighting design to date
lessons learned

• interactive global illumination is possible
• interactive algorithms *do not* match offline
• operations: *rasterize, shade, other kernels*
• other kernels are CPU or (painfully) GPU
required improvement:

dynamic environments with no precomputation
insight: $T \approx \text{low rank}$
intuition: coherence $\rightarrow$ low rank
\[ L = T \cdot I \]

\[ T = \text{?} \quad \text{size}(T) \approx 10^{11} \]

\[ L' = T' \cdot I \approx L \]

\[ (T' \cdot I) \approx (T \cdot I) < T' \approx T \]

\[ T \approx \text{low rank} \]
render representatives \hspace{1cm} \text{final image as weighted sum}

T \approx \text{low rank}
method:
cluster similar columns
render representatives
computed sum
method:
render a few rows, cluster reduced columns
cluster metric

\[
\min_{C_c} \sum_c^k \left[ \sum_{i,j \in C_c} w_i w_j \| \hat{T}_i - \hat{T}_j \| \right]
\]

cost of all clusters

sum of column pairs

norms of reduced columns

dist of reduced columns

clustering as unbiased importance sampling
results

300 rows / 900 cols: ~ 17 s

reference: ~ 17 m

2.1 M polys / 100 k lights
results

complex geometry and arbitrary materials/lights
comparison to state of the art

> 30x for same quality

insight: data parallel queries
impact: fastest algorithm for previewing accurate lighting
next: geometric scalability
(cities, forests, crowds in secs)
take home message

- shaders are not content (renderer lock)
- complex lighting does not map to GL/DX
- GL/DX are great for other applications
- for us, GL/DX complexity is not useful
- for us, GL/DX constraints’ stifle innovation
questions?
Current Generation Parallelism
In Games

Jon Olick
id Software

Beyond Programmable Shading: In Action
Brief History of Parallelism

• 1 Processor
  - The good old days.
  - Why parallelize? Just wait a little and your programs will get faster.
Brief History of Parallelism

• 2 to 3 Processors
  - Logical splitting of game process into pipelined pieces.
    • Game
    • Rendering
    • Sound
    • Loading/Decompression
Brief History of Parallelism

• About 6 to 8 Processors
  - The transition to a job scheduling type architecture
  - 1st order parallelism
    • Game
    • Rendering
    • Sound
    • Physics
    • Collision
    • Loading/Decompression
    • Etc...
Brief History of Parallelism

• About 8 to 16 Processors
  - End of CPU history.
  - Enter 1998 in GPU history.
    • Approx # of processors as average parallel scalar operations.
  - 2nd order parallelism
  - Jobs which create and manage the resources of other jobs.
    • GPU Command Processor (DMA engine)
Brief History of Parallelism

• About 16+ processors
  - 3rd order parallelism
  - Jobs which create and manage the resources of other jobs which create and manage the resources of other jobs
    • GPU Vertex Processors
Beyond Programmable Shading: In Action
Current State of Parallelism

- **Desktop Processors**
  - Intel Core 2 Quad - 4 processors - 3.2 ghz - 51.2 Gflops (102.4 Gflops theoretical)
    - Soon to be 8 core?
- **Multimedia Processors**
  - Cell Processor - 8 processors - 3.2 ghz - 134.4 Gflops (192.0 Gflops theoretical)
    - 1 main, 7 co-processors
- **Graphics Accelerators**
  - 8800 GTX - 1.35 ghz - 345.6 Gflops (518.43 Gflops theoretical)
    - 128 stream processors
THE CELL PROCESSOR
PLAYSTATION®3 Cell Processor Overview

- Game
- Animation
- Geometry Processing
- Post Processing
- Occlusion Rasterization
- Sorting
- Collision Detection
- Fourier Transform
- (De)Compression
- Not going to cover all of these...
PLAYSTATION®3 Cell Processor Overview

- Parallelize ordinarily sequential CPU processing
- Assist in what is typically considered GPU processing
PLAYSTATION®3 Cell Processor Overview

- Fitting code and data in the 256k local co-processor memory
- Best solutions are ones that don't treat the 256k local store as a typical on demand caching architecture
  - Scattered reads bad, sequential reads good
- Software Pipelining
- Only 16 bytes aligned reads/writes
- Synchronization
MD6 Animation Processing

Beyond Programmable Shading: In Action
MD6 Animation Processing

Beyond Programmable Shading: In Action
Beyond Programmable Shading: In Action

MD6 Animation Processing

- Game Logic
- Blending Tree Generation
- Low Level Operation List Generation
Beyond Programmable Shading: In Action

MD6 Animation Processing

Game Logic

Blending Tree Generation

Low Level Operation List Generation

Low Level Operation Execution
Beyond Programmable Shading: In Action
• Additive Blending
• Subtractive Blending
• Animation Algebra
  - Blend Equations
    • Animation blending trees in the form of an equation.
    • Example equation:
      - \((\text{animA} + \text{animB}) - \text{animC}\)
Partial Animation Blending

- Generalized play an animation only on the face, torso, etc...
- One weight per joint per animation
- Compute alpha for slerp via following equation:
  - For each joint
    - Let $w_0 =$ weight of joint in animation A
    - Let $w_1 =$ weight of joint in animation B
    - If($w_1 > w_0$)
      - Let $\alpha = \frac{(\alpha \times w_1)}{w_0}$
    - Else
      - Let $\alpha = \frac{((w_1 - w_0) + \alpha \times w_0)}{w_1}$
Varying parameter treatment
Varying parameter treatment
Varying parameter treatment

Beyond Programmable Shading: In Action
MD6 Animation Webs

- Separates Thinking from Representation
  - Game Object says what it wants to look like.
  - Animation Webs take care of the rest.

- Unstructured graph
  - Each node has a blend tree

- Designed with simplicity in mind
  - Animators should animate, not fiddle with nodes.
  - Extract as much information as possible directly from the animation data.
MD6 Animation Webs

Stand

Beyond Programmable Shading: In Action
MD6 Animation Webs

Stand = “play standAnim”
• Desired State = Stand
• Blend Equation = “play standAnim”
MD6 Animation Webs

- Desired State = Stand
- Blend Equation = “play standAnim”
MD6 Animation Webs

- Desired State = Stand
- Blend Equation = “play standAnim”
MD6 Animation Webs

- Desired State = **Walk**
- Blend Equation = “play standAnim”
MD6 Animation Webs

• Desired State = Walk
• Blend Equation = “blend standAnim and walkAnim”
MD6 Animation Webs

- Desired State = Walk
- Blend Equation = “play walkAnim”
MD6 Animation Webs

- Desired State = Stand
- Blend Equation = “play standAnim”
MD6 Animation Webs

- Desired State = Stand
- Blend Equation = “play standAnim”
MD6 Animation Webs

- Desired Traversal = Stand to Walk
- Desired Direction = ~5 degrees
MD6 Animation Webs

- Desired Traversal = Stand to Walk
- Desired Direction = ~175 degrees
MD6 Animation Webs

- Desired State = Run
MD6 Animation Webs

- Desired Traversal = Run to Cover
- *Mechanical Behavior* - Not believable
MD6 Animation Webs

- Desired State = Run

Beyond Programmable Shading: In Action
MD6 Animation Webs

- Desired State = Run
MD6 Animation Webs

- Automatically picks best animation for search criteria
- Time warps animation to perfectly match criteria
- Works in parallel with game code
Beyond Programmable Shading: In Action

GEOMETRY PROCESSING
Two modes of usage

• Primary mode
  - Use offline tools
  - Partition into vertex sets
  - Use indexed triangles
  - All features of pipeline can be used
Two modes of usage (cont)

- **Secondary mode**
  - Data generated by other tools
  - Formats other than indexed triangles
  - Non-partitioned objects
  - Subset of pipeline features can be used
Beyond Programmable Shading: In Action

SPU Geometry Pipeline Stages

- Vertex Decompress
- Index Decompress
- Blend Shapes
- Skinning
- Progressive Mesh
- Triangle Culling
- Compression
- Output
Vertex Decompression

- Beyond Programmable Shading: In Action
Vertex Attributes

Unique Vertex
Array 0

Instance Vertex
Array 1
Vertex Decompression

Beyond Programmable Shading: In Action
24bit Unit Vector

• Smallest 2 compression
  - Two smallest components with 10 bits each
    • Encoded from $-\sqrt{2}/2$ to $+\sqrt{2}/2$
  - Largest component reconstructed via
    • Largest = $\sqrt{1 - \text{smallestA}^2 - \text{smallestB}^2}$
    • One additional bit for sign of largest component.
24bit Unit Vector

- **Smallest 2 compression**
  - Two smallest components with 10 bits each
    - Encoded from $-\sqrt{2}/2$ to $+\sqrt{2}/2$
  - Largest component reconstructed via
    - Largest = $\sqrt{1 - \text{smallestA}^2 - \text{smallestB}^2}$
    - One additional bit for sign of largest component.

- **One more bit to represent W as +1 or -1**
  - For constructing bi-normal from normal and tangent.
N-bit Fixed Point with integer offsets

- Simple n.x fixed point values
  - Per-segment integer offset
- Bit count may vary from attribute to attribute
Index Decompression

SPU Pipeline
- Vertex Decompress
- Index Decompress
- Blend Shapes
- Skinning
- Progressive Mesh
- Triangle Culling
- Compression
- Output

Beyond Programmable Shading: In Action
Index Table Construction

- Index table is created by a vertex cache optimizer
  - Based on K-cache algorithm
- Number of vertex program outputs affects Vertex Cache size.
- Four vertex mini cache most important optimization factor
Strip Example

3 new vertices
Strip Example

1 new vertex
Strip Example

1 new vertex
Strip Example

1 new vertex...

2 vertices + 1 per triangle in total
Free Form Example

3 new vertices
Free Form Example

1 new vertex
Free Form Example

1 new vertex
Free Form Example

1 new vertex
Free Form Example

1 new vertex
Free Form Example

0 new vertices!
Free Form Example

1 new vertex
Free Form Example

1 new vertex
Beyond Programmable Shading: In Action

Free Form Example

0 new vertices...

2 vertices + 3 per 4 triangles in total
Index Cache Optimizer

- Our vertex cache optimizer produces very regular index data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>
## Index Decompression

- Provided vertex cache optimizer produces very regular index data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>
Index Decompression

Triangle Indexes

0 1 2

Beyond Programmable Shading: In Action
Index Decompression

Triangle Indexes

2 0 1
### Index Decompression

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>
## Index Decompression

<table>
<thead>
<tr>
<th>00</th>
<th>PREVIOUS INDEX 0</th>
<th>PREVIOUS INDEX 2</th>
<th>NEW INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>PREVIOUS INDEX 2</td>
<td>PREVIOUS INDEX 1</td>
<td>NEW INDEX</td>
</tr>
<tr>
<td>10</td>
<td>PREVIOUS INDEX 1</td>
<td>PREVIOUS INDEX 0</td>
<td>NEW INDEX</td>
</tr>
<tr>
<td>11</td>
<td>NEW INDEX</td>
<td>NEW INDEX</td>
<td>NEW INDEX</td>
</tr>
</tbody>
</table>
Index Decompression

85% compression
6.5 : 1
Blend Shapes in MLB 08: The Show

Beyond Programmable Shading: In Action
Beyond Programmable Shading: In Action

Skinning

SPU Pipeline
- Vertex Decompress
- Index Decompress
- Blend Shapes
- **Skinning**
- Progressive Mesh
- Triangle Culling
- Compression
- Output
Skinning on SPUs

```c
void SkinVs(float4 inPosition : ATTR0, float4 weights : ATTR3,
            float4 matrixIndex : ATTR4,
            out float4 position : POSITION,
            uniform float4 joints[72], uniform float4x4 modelViewProj)
{
    position = 0;
    for (int i = 0; i < 4; i++)
    {
        float idx = matrixIndex[i];
        float3x4 joint = float3x4(joints[idx+0], joints[idx+1],
                                  joints[idx+2]);
        position += weights[i] * mul(joint, inPosition);
    }
    position = mul(modelViewProj, position);
}
```
Skinning on SPUs

30% Performance Improvement
Skinning on SPUs

30% Performance Improvement

Shadow map generation.... 70%!
Discrete Progressive Mesh

- Smoothly reduces the triangle count as a model moves into the distance
- With discrete progressive mesh, the LOD calculation is done once for an entire object
At an LOD there are two types of vertexes

LOD = 0.0

Parent Vertex
Child Vertex
As the LOD level decreases, the children “slide” towards their parents.
The children continue to move towards their parents

LOD = 0.7

Parent Vertex
Child Vertex
At the next integral LOD, all child vertexes disappear as do the triangles.

LOD = 1.0
Continuous Progressive Mesh

• Like discrete progressive mesh, child vertexes move smoothly toward their parents

• However, the LOD is calculated for each vertex instead of just once for the object
Vertex set about to undergo continuous progressive mesh

Parent Vertex
● Child Vertex, LOD 1
○ Child Vertex, LOD 0

Beyond Programmable Shading: In Action
A single vertex set can straddle several LOD ranges

- Parent Vertex
- Child Vertex, LOD 1
- Child Vertex, LOD 0
Vertexes move depending on their distance
Triangle Culling
Up to 70% of triangles do not contribute to final image.
Beyond Programmable Shading: In Action
Back Facing Triangles
Zero Area Triangles
Zero Area Triangles
No Pixel Triangles
Triangle Culling
Multisampling adds some complications...
Beyond Programmable Shading: In Action
Triangle Culling

10% to 20% Performance Improvement
Compression for Output

SPU Pipeline
- Vertex Decompress
- Index Decompress
- Blend Shapes
- Skinning
- Progressive Mesh
- Triangle Culling
- Compression
- Output

Beyond Programmable Shading: In Action
Beyond Programmable Shading: In Action

Float Tables

[Diagram of float tables with varying shades of blue and white]
When done, the vertex attributes are compressed into one output stream.
Output Buffering Schemes

- Vertex and index data constructed by the SPUs is output from SPU local store
- Holes in the RSX command buffer are patched with pointers to the vertex and index data as well as the draw commands
Double Buffer

- Each buffer stores vertex and index data for an entire frame
- SPUs atomically access a mutex which is used to allocate memory from a buffer
- Easy synchronization with the RSX™ once a frame
- Uses lots of memory
It is possible to completely fill a buffer

- Can use a callback to allocate new memory (which you may not have)
- Don’t draw geometry that doesn’t fit (difficult to pick which geometry not to draw)
Double buffering adds a frame of lag

<table>
<thead>
<tr>
<th>Build Jobs on PPU</th>
<th>Process Jobs on SPU</th>
<th>Render on RSX™</th>
<th>Scan Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Jobs on PPU</td>
<td>Process Jobs on SPU</td>
<td>Render on RSX™</td>
<td>Scan Out</td>
</tr>
<tr>
<td>Build Jobs on PPU</td>
<td>Process Jobs on SPU</td>
<td>Render on RSX™</td>
<td>Scan Out</td>
</tr>
</tbody>
</table>

Beyond Programmable Shading: In Action
Single Buffering

- Uses only half the memory!
- Still possible to completely fill the buffer
Single Buffering has a shorter pipeline

- Vertex and index data is created just-in-time for the RSX™
- Draw commands are inserted into the command buffer while the RSX™ is rendering
- Requires tight SPU↔RSX™ synchronization
### SPU ↔ RSX™ Synchronization Using Local Stalls

<table>
<thead>
<tr>
<th>Command Buffer</th>
<th>Draw 17</th>
<th>Local Stall</th>
<th>Local Stall</th>
<th>Local Stall</th>
<th>Local Stall</th>
<th>Local Stall</th>
<th>Local Stall</th>
<th>Other</th>
</tr>
</thead>
</table>

- Place local stalls in the command buffer where necessary
- RSX™ will stop processing at a local stall until it is overwritten by new commands
- SPUs will generally stay ahead of the RSX™, so stalls rarely occur
SPU will overwrite local stalls when it outputs a set of new commands.

No SPU→SPU synchronization required!
Ring Buffers

- Small memory footprint
- Will not run out of memory
- Can stall the SPUs if buffers become full
- Objects need to be processed in the same order the RSX™ renders them to prevent deadlock
RSX™ writes a semaphore once a chunk of data has been consumed

- A command to write a semaphore needs to be added to the command buffer after all commands that use the data
  - The value of the semaphore to be written is the new end of free area pointer
Each SPU has its own buffer
# Geometry Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles / Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Decompression</td>
<td>10.5</td>
</tr>
<tr>
<td>Index Decompression</td>
<td>12.3</td>
</tr>
<tr>
<td>Blend Shapes (Per Shape)</td>
<td>11.0</td>
</tr>
<tr>
<td>Vertex Transform + Triangle Culling</td>
<td>30.4</td>
</tr>
<tr>
<td>Matrix Palette Skinning</td>
<td>34.4</td>
</tr>
</tbody>
</table>

*Beyond Programmable Shading: In Action*
do
{
    m1  = in1;
in1 = si_lqx(pIn1, offset);
m2  = in2;
in2 = si_lqx(pIn2, offset);
m3  = in3;
in3 = si_lqx(pIn3, offset);
temp2 = si_selb(m3, m1, mask_0X00);
si_stqx(out1, pOut1, offset);
temp3 = si_selb(m2, m1, mask_00X0);
si_stqx(out2, pOut2, offset);
temp1 = si_selb(m1, m2, mask_0X00);
si_stqx(out3, pOut3, offset);
    offset = si_ai(offset, 0x30);
    out2 = si_shufb(m2, temp2, qs_bCaD);
    out1 = si_shufb(temp1, m3, mask_00X0);
    out3 = si_shufb(m3, temp3, qs_caBD);
} while(si_to_int(offset) != 0);
do
{
    m1  = in1;
in1 = si_lqx(pIn1, offset);
m2  = in2;
in2 = si_lqx(pIn2, offset);
m3  = in3;
in3 = si_lqx(pIn3, offset);
temp2 = si_selb(m3, m1, mask_0X00);
si_stqx(out1, pOut1, offset);
temp3 = si_selb(m2, m1, mask_00X0);
si_stqx(out2, pOut2, offset);
temp1 = si_selb(m1, m2, mask_0X00);
si_stqx(out3, pOut3, offset);
offset = si_ai(offset, 0x30);
out2 = si_shufb(m2, temp2, qs_bCaD);
out1 = si_selb(temp1, m3, mask_00X0);
out3 = si_shufb(m3, temp3, qs_caBD);
} while(si_to_int(offset) != 0);
Beyond Programmable Shading: In Action

1 SPU
1 SPU

800,000+
Triangles Per Frame
at 60 Frames per Second
1 SPU

800,000+
Triangles Per Frame
at 60 Frames per Second

60% of which are culled!
Next Generation Parallelism
In Games
Jon Olick
id Software

SIGGRAPH 2008
Beyond Programmable Shading: In Action
GAME ENTITY PROCESSING
Game Entity Processing

- Current Generation
  - Serial Processing of entities in a giant for loop.

```cpp
for(int i = 0; i < numEntities; ++i) {
    entity[i]->Think();
}
```
Game Entity Processing

• Current Generation
  - Serial Processing of entities in a giant for loop.

• Next Generation
  - Parallelism via Double Buffering
Game Entity Processing

• Current Generation
  - Serial Processing of entities in a giant for loop.

• Next Generation
  - Parallelism via Double Buffering
  - Each entity can only read from previous frame’s results
Game Entity Processing

• Current Generation
  - Serial Processing of entities in a giant for loop.

• Next Generation
  - Parallelism via Double Buffering
  - Each entity can only read from previous frame’s results
  - Each entity can only write to itself
  - Every entity runs in parallel with each other with no dependency stalls.
Game Entity Processing

• Record the progress of the game and replay it to debug.

• Single thread and randomize processing of entities to help find bugs.

• Can protect memory so that bad accesses cause exceptions to enforce double buffering rules.
RAY CASTING
Why Ray Casting?

• A good question...
Beyond Programmable Shading: In Action
• Back in Quake 1
  - If you had to make a decision between an additional CPU and a Graphics Card which would you choose?
• Back in Quake 1
  - If you had to make a decision between an additional CPU and a Graphics Card which would you choose?
  - Why is this any different today?
• **Back in Quake 1**
  - If you had to make a decision between an additional CPU and a Graphics Card which would you choose?
  - Why is this any different today?
  - Its not.
Why Ray Casting?

• What value does it provide to game developers?
  - Scene complexity no longer significantly affects performance or memory
Why Ray Casting?

• What value does it provide to end users?
• Screen shot from RAGE goes here
• Screen shot from RAGE goes here
• (highlight flat regions)
Why Ray Casting?
Current State of Rasterization

- Command Buffer
- Vertex Processing
- Triangle Setup
- Fragment Processing

Beyond Programmable Shading: In Action
Future of Rasterization
Future of Rasterization

Beyond Programmable Shading: In Action
Future of Rasterization

Beyond Programmable Shading: In Action
Future of Rasterization

Beyond Programmable Shading: In Action
Future of Rasterization

Beyond Programmable Shading: In Action
Future of Rasterization

Beyond Programmable Shading: In Action
Beyond Programmable Shading: In Action
Future of Rasterization

Command Buffer

Vertex Processing

Triangle Sorting

Triangle Setup

Fragment Processing

Vertex Processing

Fragment Processing

Beyond Programmable Shading: In Action
Future of Rasterization

Command Buffer → Vertex Processing → Triangle Setup → Fragment Processing → Multiple Cores

Beyond Programmable Shading: In Action
Future of Rasterization x2

GPU Triangle Setup
## Future of Rasterization x2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

**GPU Triangle Setup**
## Future of Rasterization x2

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

**GPU Triangle Setup**
Future of Rasterization x2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td><strong>10</strong></td>
<td><strong>9</strong></td>
<td><strong>11</strong></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

GPU Triangle Setup
### Future of Rasterization x2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

**GPU Triangle Setup**
Future of Rasterization x2

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>19</td>
</tr>
</tbody>
</table>

GPU Triangle Setup
Why voxels, and not triangles?

- Unifying Primitive
- No Near Phase Collision
- Solves Two Problems in One
  - Unique Texturing & Unique Geometry
- Good for CUDA / Larabee
Beyond Programmable Shading: In Action

Why is the control flow efficient?
Why is the control flow efficient?
Why is the control flow efficient?
Why is the control flow efficient?
Voxel Mip Mapping - Thin Walls
### Voxel Mip Mapping - Thin Walls

<table>
<thead>
<tr>
<th>Red</th>
<th>Blue</th>
<th>Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Voxel Mip Mapping - Thin Walls**

Beyond Programmable Shading: In Action
Voxel Mip Mapping - Thin Walls

Beyond Programmable Shading: In Action
Caveats of Ray-Tracing?

- “Primary rays cache, secondary rays thrash”™
  - Importance sampling to the rescue!
- Ray Tracing != Ray Casting
Results May Vary
Sparse Voxel Oct-trees

- Oct-trees as collection of maximal blocks.
  - Related to run-length encoding.
- Possibly variable splitting planes in future.
  - Guided by artist placed hint planes.
Data Structure

• Disk Caching with Virtual and Physical Pages
  - Start out with a single virtual page.
  - Render some voxels into the tree until page capacity is reached.
  - Split page into 8 sub-pages and attempt to add the overflow voxel again.
  - Store out virtual pages to disk.
  - Load/Unload each page’s levels as necessary at runtime.
Data Structure

• Page capacity can be based on...
  - CUDA's shared memory size
  - Cell's SPU local store size
  - Optimum disk streaming performance
  - Minimum physical page memory
Data Structure - Page Fragmentation

- Traverse indexing oct-tree
  - Write out pages according to optimal layout (breadth first, depth first, etc...)
• Execution time proportional to number of nodes.
• Number of nodes can be reduced through translation.
• Translating by $2^n$ doesn’t affect any oct-tree level smaller than $2^n$. 
• Create scratch page with enlarged region
  - \( 2^{n+1} \times 2^{n+1} \times 2^{n+1} \)

• Apply successive translations of magnitude power of 2 in the x, y, & z directions and keep a count of the number of nodes required.

• Store off total translation so that the ray casting can be adjusted appropriately.

• \( O(4n) \)
  - \( n \) is the number of levels in the oct-tree
Data Structure - Page Optimization
Data Structure - Page Optimization

• Minimize outside nodes for faster tracing
<table>
<thead>
<tr>
<th>Data Structure - Page Optimization</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Data Structure</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Page Optimization</th>
<th>Page Optimization</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>SIGGRAPH 2008</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>175</th>
</tr>
</thead>
</table>
Data Structure - Page Optimization

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value 1</td>
<td>Value 2</td>
<td>Value 3</td>
<td>Value 4</td>
</tr>
<tr>
<td>Value 5</td>
<td>Value 6</td>
<td>Value 7</td>
<td>Value 8</td>
</tr>
<tr>
<td>Value 9</td>
<td>Value 10</td>
<td>Value 11</td>
<td>Value 12</td>
</tr>
<tr>
<td>Value 13</td>
<td>Value 14</td>
<td>Value 15</td>
<td>Value 16</td>
</tr>
</tbody>
</table>

Note: The table and chart are placeholders for data representation.
Data Structure

- Different structures for runtime and storage.
Runtime Data Structure

- child pointers : 32
- diffuse rgb : 3
- specular scale/power : 1
- normal xy : 2
- total : 38 bytes per node
Storage Data Structure

- children bit mask : 1
- diffuse rgb : 3
- specular scale/power : 1
- normal xy : 2
- total : 7 bytes per node
Data Compression

- Compressing child bits
- Compressing Colors
Compressing Child Bits

• (diagram showing statistical distribution of child bytes)
Compressing Child Bits

• Split by oct-tree level.
• Arithmetic Compression
• (diagram showing statistical distribution of child bytes at each oct-tree level)
Compressing Color Data

• (Diagram showing each node of the tree as a delta between itself and its parent)
Compressing Color Data

- Split by oct-tree level.
- Quantization
- Arithmetic Compression
Data Structure Size

- 1.15 bits of positional data per voxel
- Cost savings improves as triangle size decreases.
- 72 bits equivalent per triangle in oct-tree
  - (for next generation)
- 160 bits per triangle in traditional format
  - x,z,y,s,t all 32-bits
  - 2.2:1 compression ratio
- 80 bits per triangle in compressed format
  - x,y,z,s,t all 16-bits
  - 1.1:1 compression ratio
Generating the data

- Every surface can enumerate into voxels.
- 3D Scan Conversion
- Volume Projection
- Subdivision
3D Scan Conversion

Beyond Programmable Shading: In Action
Beyond Programmable Shading: In Action
3D Scan Conversion

Beyond Programmable Shading: In Action
3D Scan Conversion

• Only 8 connectivity required.
• Flood fill world and remove unnecessary voxels.
Generating the data

- Generate geometry mip-maps
- Geometry mips consume 15% extra data
Generating the data

• Save off the un-lit data.
Generating the data

• Perform ray-tracing to light the voxels.
Using the data

• For each pixel on the screen
  - Shoot out a ray into the oct-tree and record the color (and depth?)
Need More Detail?

- Bottom up?
- Top Down?
- Etc...
Rendering Skinned Characters

- Ray cast against triangle mesh
- Transform to base pose
- Trace with local oct-tree
Abstracting Skinned Characters

• Moving world geometry without oct-tree re-organization cost.
• Enables instancing.
Hybrid Rendering

• Render a coarse hull of the geometry into a z-buffer.
  - Automatically calculate from voxel geometry.

• Post-process the depth-buffer to start the ray casting at the point specified by the depth-buffer instead of at the ray origin.
  - Possibly transforming it as well.
Hybrid Rendering

- Skips most of the traversal process.
  - 2x to 4x expected speed improvement
- Allows dynamic geometry
- Prevents second and third order ray tracing
  - No z-buffer pre-pass to transform ray start for dynamic geometry
Hybrid Rendering with Atmospheric Effects

• Atmospheric effects in combination with rasterization possible.
  - Rasterize coarse hull into depth buffer
  - Rasterize triangles into color and alternate depth buffer
  - At ray-collision time, keep track of atmospheric collisions along the way.
  - If hit rasterized depth,
    • stop and modify existing color
  - Else
    • modify voxel color.
Adaptive Sub-Sampling

• After rendering the scene, perform a Sobel edge filter over the frame buffer to figure out where additional rays would improve the quality of the image.

• Cast additional rays.

• Repeat until 16 ms.
Adaptive Sub-Sampling Problems

• Inherently always sampling the most divergent parts of the scene
• Can manage performance hit by sampling highly aliased to less aliased in chunks
Infinite Surface Detail

• Oct-tree node's recursively point back in on themselves to create an infinite amount of detail

• Create detail types octree sub-segments to simulate rough, smooth, porous, sharp edges, etc..
  - Recursive trees are entirely in cache, so very fast calculation.
  - Requires delta compressed colors to be in the runtime format for correct shadowing
    • Not a huge computation burden, but more bandwidth required
How much time to innovate?

- 1 year tools
- 3 months runtime
Expected Runtime Performance

- 33% of the time rendering characters / etc
- 66% of the time rendering world
- Ray-casting the world must complete in ~20ms for 30 FPS
- Theoretically possible on today's technology (GeForce 8800 Series)
  - Possible on 7800 with additional memory requirements.
How would this affect a platform launch?

- Generational skip in geometric complexity
- Next gen platforms 4 times better at least
- Completely plausible at 720p and 60 FPS w/ antialiasing
  - or 1080p and 60 FPS
Questions

Jon Olick (jon.ollipop@gmail.com)
id Software