Last Five Years

• Rise of programmable GPUs
  – 100s/GFLOPS compute
  – 10s/GB/s bandwidth

• Architecture characteristics have deeply influenced s/w and algorithm development

• Revolution in interactive graphics as software has exploited new hardware
Next Five Years: A Second Revolution

- Continued GPU innovation
- CPUs finally providing FLOPS as well
  - Don’t require nearly as much parallelism
- Shared memory/high bandwidth interconnect enable flexible computation model
- Future role of today's graphics APIs is unclear
Overview

• The transition to programmable GPUs and the importance of computation in interactive rendering
• New heterogeneous architectures
• Implications for graphics software
  – Implementation challenges and opportunities
  – Increasing importance of software to drive complex hardware
• Longer-term trends and convergence?
Offline Rendering 5 Years Ago

Shrek (PDI/Dreamworks)
Interactive 5 Years Ago

Quake 3 (id Software)
Modern Offline Rendering

Madagascar (PDI/Dreamworks)
Modern Interactive Rendering

Project Gotham Racing

neoptica
Modern Offline Rendering

Starship Troopers 2 (Tippett Studio)
Modern Interactive Rendering

I-8 (Insomniac Games)
What’s Happened In The Last 5 Years?

• GPUs have taken advantage of semiconductor trends to deliver performance
• GPU strengths/weaknesses have sparked innovation in algorithms and software
  – Interactive graphics is about computation
• Interactive is delivering near-offline quality 1,000,000x faster
GPU Architecture Has Led To Algorithmic Innovation

- GPU performance cliffs are large
  - Must stay on fast path
  - Easier to achieve good GPU utilization than good CPU utilization?
- Benefits from staying on fast path are enormous
- Everyone has a GPU
  - Many more developers working in this space
  - Millions of GPUs in PCs: incentive to use them efficiently
Three Phases Of Hardware-Accelerated Graphics

• Configurable fixed-function graphics
  – Register combiners, multipass

• Programmable shading
  – Vertex and fragment shaders, texture composition, pattern generation, lighting models

• Programmable graphics
  – Shaders implement graphics algorithms using complex data structures
Example: Displacement Mapping Redefined

• Classic technique from offline rendering
• Texture map defines offset from base surface
• Offline approach:
  – Finely tessellate to pixel-sized triangles
  – Move triangle vertices appropriately
  – Discard triangles when done with them
Displacement Mapping Redefined

- Offline approach not suited to current hardware
  - GPU is balanced for ~8 pixel big triangles
  - Not enough vertex processing power for many small triangles
  - Small triangles not good for rasterizer/fragment processor
- Therefore, developers must invent new techniques better suited to the hardware
Displacement Mapping Redefined

- Draw bigger triangles, do work in fragment processor
  - Better match to GPU’s strengths and weaknesses

- Representative approach: Donnelly’s distance map-based ray tracing
  - Small 3D table stores representation of empty space above displaced surface
  - Fragment shader marches through space until surface intersection is found
Distance Map Sphere Tracing
Bump Mapping

Will Donnelly
Displacement Mapping

Will Donnelly
Data Structures Are Central To Interactive Rendering

The most important value of GPU programmability is not procedural texture generation, but the ability to use data structures in interactive rendering.
Data Structures For Interactive Rendering

• Programmable graphics >> programmable shading
• Examples:
  – Distance map for displacement mapping
  – Quadtree for adaptive shadow map refinement
  – Hierarchical disk tree for dynamic ambient occlusion
• GPGPU mindset is needed to use the GPU this way
  – How do I make this data-parallel processor run data-parallel algorithms to achieve a desired result?
  – This expertise is not widely held…
One Little Problem…

• What if the displacement texture is computed by the GPU?
  – e.g. for a deforming displaced surface
• Need to also compute a new data structure for efficient sphere tracing
• No problem to do that on the powerful GPU… right?
Dynamic Data Structures Are A Big Problem

- CPU is good at traversing data structures, very bad at building them
- Data structure construction generally has little available parallelism
  - The GPU requires a lot of parallelism for performance
- There isn’t enough bandwidth for the CPU to build them dynamically
Dynamic Data Structures Are A Big Problem

• Sorting, reductions, etc., to build data structures on the GPU are possible
  – Getting good performance is very difficult
  – A lot of bandwidth is burned along the way
• Only experts have the necessary knowledge
  – Deep understanding of the hardware is needed
• Amdahl’s law: inefficient data structure construction will increasingly be the bottleneck
What’s New In Hardware Architectures?

The Getaway 3 (SCEE)
Next-Generation Heterogeneous Architectures

• We are again in a time of architectural change
  – CPUs are going parallel, starting to offer FLOPS
• High bandwidth interconnects let CPU and GPU work together
  – Heterogeneous compute resources are available
• New architectures address GPU weaknesses
  – Equally revolutionary implications for interactive graphics
The PC of 2006

- 2 core CPU
  - 30 GFLOPS
- GPU
  - 200 GFLOPS
- Interconnects
  - 1 GB/s CPU to GPU
  - 8 GB/s CPU to system memory
  - 30 GB/s GPU to graphics memory
What’s Wrong With Today’s PC?

- CPU still doesn’t have spare FLOPS for interactive graphics
- 1 GB/s on PCI-E prohibits CPU-GPU round trips
- The only processor with FLOPS, the GPU, requires much parallelism for performance
- GPU has limited memory writes
  - Improved somewhat by DX10
Game Console Architecture: XBox 360

- 3-core PPC CPU @ 3.2GHz
  - 115 GFLOPS
- Xenos GPU
  - 240 GFLOPS
  - EDRAM framebuffer
- Interconnects
  - 10GB/s CPU to shared memory
  - 20GB/s GPU to shared memory
Game Console Architecture: PS3

- **STI Cell @ 3.2GHz** (one PPU, seven SPUs)
  - ~200 GFLOPS
- **RSX GPU**
  - ~200 GFLOPS
- **Interconnects**
  - 25GB/s Cell to main memory
  - 20GB/s Cell to GPU, 15GB/s GPU to Cell
  - 22GB/s GPU to graphics memory
Heterogeneous Architecture Implications

- At 720p resolution, 60 frames per second...
  - 200 GFLOPS gives ~3000 FLOPS/pixel
  - 20GB/s allows 75 floats of communication/pixel
    - 150 halves
    - (Peak performance)

- Developer can choose: provide little parallelism and manage memory latency, or provide a lot and ignore it
Good News About
The New World Order

- GFLOPS are available on both the CPU and the GPU
  - Can have performance even with much less parallelism
  - Can consider algorithms not suited to GPU alone
- Bandwidth enables algorithm decomposition between CPU and GPU
  - While still maintaining interactivity
  - Goodbye to the one-way graphics pipeline
Good News About Dynamic Data Structures

• These architectures have the potential to build complex data structures at runtime:
  – GPU computes data
  – CPU builds data structure based on result
  – GPU uses data structure

• This can play to the strengths of both processors
Using The GPU More Efficiently With Help From the CPU

• The one-way PC graphics pipeline is a blunt hammer
  – CPU can use occlusion query, etc, to try to drive GPU more efficiently, but little information is available to it
  – And the GPU is unable to issue commands to itself
  – (DX10 geometry shaders do help here)

• Heterogeneous architectures can do much better
  – GPU does some work
  – CPU examines intermediate results
  – CPU gives GPU more work
Example: Efficient Data Structure Traversal

- The GPU can traverse tree data structures
  - E.g. ambient occlusion disk trees, kd-trees, lightcuts, shadow quad-trees, ...
- If a large collection of nearby fragments all traverse the same top level nodes, computation is wasted
- CPU can start traversal until divergence, then let GPU continue from there
  - Analogies to Reshetov et al Multi-Level Ray Tracing
Many Other Opportunities

• More efficient deferred shading
• Skip rendering cube map faces and shadow maps that are not needed
• Adaptive refinement
• Can do a smaller superset of the necessary computation for rendering an image than if GPU alone was doing the rendering
A Few Little Details…

- Parallel programming is a notorious quagmire
  - Heterogeneous processors don’t make it easier
  - Data synchronization and movement are tricky
- GPUs are the only type of parallel processor that has ever seen widespread success
  - …because developers generally don’t know they are parallel!
  - And if you want to do programmable graphics rather than programmable shading, cracks start to show
What Is The Right Programming Model?

- GPU data-parallel languages (Cg, HLSL) do not map well to CPU model
- C/C++ don’t map well to GPU model
- Need new approaches and abstractions
  - Unified language that spans all processors?
  - Native code + glue?
  - Functional programming (this time at last)?
Future Graphics APIs

• Whither OpenGL 3.0 and DX11?
  – Are these APIs for controlling the GPU, or APIs for interactive graphics?
    • i.e. how do they handle heterogeneous architectures?
  – Developers and GPU vendors generally have opposite views on this question

• Currently closely tied to the one-way PC graphics pipeline
Future Graphics APIs

- What is the role of a graphics API if graphics is about computation and data structures?
  - If API does not embrace all processors and make it easier to use them for graphics, it will become irrelevant.

- Has the time come to kill the graphics API and expose the hardware instead?
Exposing A GPU

Hardware Abstraction

• Graphics drivers often are an impediment to using the GPU well
  – Details they abstract are increasingly important for developers to understand
  – Hide actual perf. characteristics of the h/w
  – Programmable graphics needs a more direct understanding of memory for performance
Exposing A GPU Hardware Abstraction

• Closer-to-the-metal APIs like ATI’s CTM?
  – Focus on exposing the GPU’s computational capabilities
  – Expose GPU memory model directly

• More burden on h/w vendors to have clean orthogonal designs, though

• Developers are unlikely to be happy with vendor-specific APIs
Future Hardware

• What is the future of GPUs?
  – Continually increasing parallelism requirements are a problem for programmable graphics
  – (Less so for programmable shading)

• What is the future workload?
  – Graphics continues to offer a lot of parallelism
  – But more and more irregular computation
  – If CPU offloads irregular parts of the computation, what is left for the GPU is more homogeneous
Future Hardware

• More reasons to build a single chip CPU and GPU than cost savings
  – Off-chip bandwidth will become more and more limited w.r.t. available computation

• Will hardware designers have a broad perspective that allows all processors to work well together for graphics?
  – Presumably a certain two of them will at least!
What Is The Right Future Architecture?

- Homogeneous vs heterogeneous?
  - 64 P4s or 8 P4s and 128 fragment processors?
  - More than a few CPUs are overkill for graphics and other parallelizable compute-intensive workloads
  - But the four processors on PS3 (PPU, SPU, vertex, fragment) are probably too many

- Sweet spot is probably a handful of CPUs and a lot of fragment processors/SPUs
  - (With a rasterizer and texture units)
SPUs vs. Fragment Processors

- Two very different models for FP performance

**SPU**
- Arbitrary writes (to local store and main memory)
- User-managed latency hiding
- Only need one thread per SPU

**Fragment Processor**
- Limited writes
- Memory latency hidden through parallelism
- Requires many threads
Memory Models I

- GPU: limited writes, efficient streaming reads, no user data synchronization
- Shared memory multi-core CPU: all up to the application or support library
Memory Models II: SPU

- Explicit DMA to/from main memory to local memory
- This is good discipline
  - Encourages operating on large chunks of data
  - Encourages data reuse
- Explicit transfers make communication clear
  - Useful information for low-level support libraries
- This style is necessary for perf. on CPUs anyway
Challenges In Building Interactive Renderers

- All increasing with new architectures
- Choosing the right algorithms
- Implementation complexity
  - Math is hard
  - Architectures are complex
  - Need to develop algorithms that span CPU and GPU
- Software must solve these problems for new hardware to have value
Summary

• New heterogeneous architectures have great promise for interactive graphics
  – Dynamic data structures
  – Efficient GPU utilization
  – An enabler for programmable graphics

• Challenges are significant
  – Programming model
  – Algorithm implementation
  – Designing the right hardware architectures
Thanks

• Neoptica: Craig Kolb, Aaron Lefohn, Paul Lalonde, Tim Foley, Geoff Berry
• Stanford: Mike Houston, Jeremy Sugarman, Daniel Horn, Kayvon Fatahalian, Pat Hanrahan
• John Owens
• Bill Mark
• Kiril Vidimce
• Doug Epps
• Eric Leven
Questions?