GameDevelopers Conference

EVOLVE GDC>04

GPGPU: Beyond Graphics

Mark Harris, NVIDIA

NVIDIA.

What is GPGPU?

- **G**eneral-**P**urpose Computation on **GPU**s
  - GPU designed as a special-purpose coprocessor
  - Useful as a general-purpose coprocessor
- The GPU is no longer just for graphics
  - It is a massively parallel stream processor
  - 32-bit float support
  - Flexible programming model
  - Huge memory bandwidth

What is GPGPU?

- Much academic research in this area
  - Cellular automata, fluid dynamics
  - Cloth / hair simulation, soft bodies
  - Particle systems, collision detection
  - Global illumination, computer vision
  - Computational Geometry
  - www.GPGPU.org

Outline

- **Motivation: Why GPUs?**
- Mapping computational concepts to GPUs
- Tricks of the trade: Branching Techniques
- Current Limitations
- New OpenGL Functionality
- The Future

Why GPUs?

Why GPUs? Economics, Really.

- Graphics is "embarrassingly parallel"
  - Data-parallel computation: vertices + pixels
- Millions of GPUs ship every month
  - Largely thanks to multi-billion [$,£,€,¥] game industry
- Result
  - GPUs are inexpensive, but powerful
  - Low cost per GFLOP

Compound Performance Growth Rates

	Measured	Period	CAGR Tri / sec	CAGR Frag / sec
SGI	Flat Color	84 – 96	1.8	1.3
NVIDIA	No AA	97 – 02	1.8	2.4
SGI	Depth Buf	84 – 96	2.2	2.2
NVIDIA	AA 32-bit	97 – 03	2.1	2.3

Significantly above Moore's Law

CAGR 2.0 → 1000x per decade

Slide courtesy of Kurt Akeley

Semiconductor Scaling Rates

From: *Digital Systems Engineering*, Dally and Poulton

Parameter	2001 Value	Yearly Factor	Years to Double (Half)
Moore's Law (grids on a die)**	1 B	1.49	1.75
Gate Delay	150 pS	0.87	(5)
Capability (grids / gate delay)		1.71	1.3
Device-length wire delay		1.00	
Die-length wire delay / gate delay		1.71	1.3
Pins per package	750	1.11	7
Aggregate off-chip bandwidth		1.28	3

** Ignores multi-layer metal, 8-layers in 2001

Slide courtesy of Kurt Akeley

Communication is the Key to Performance

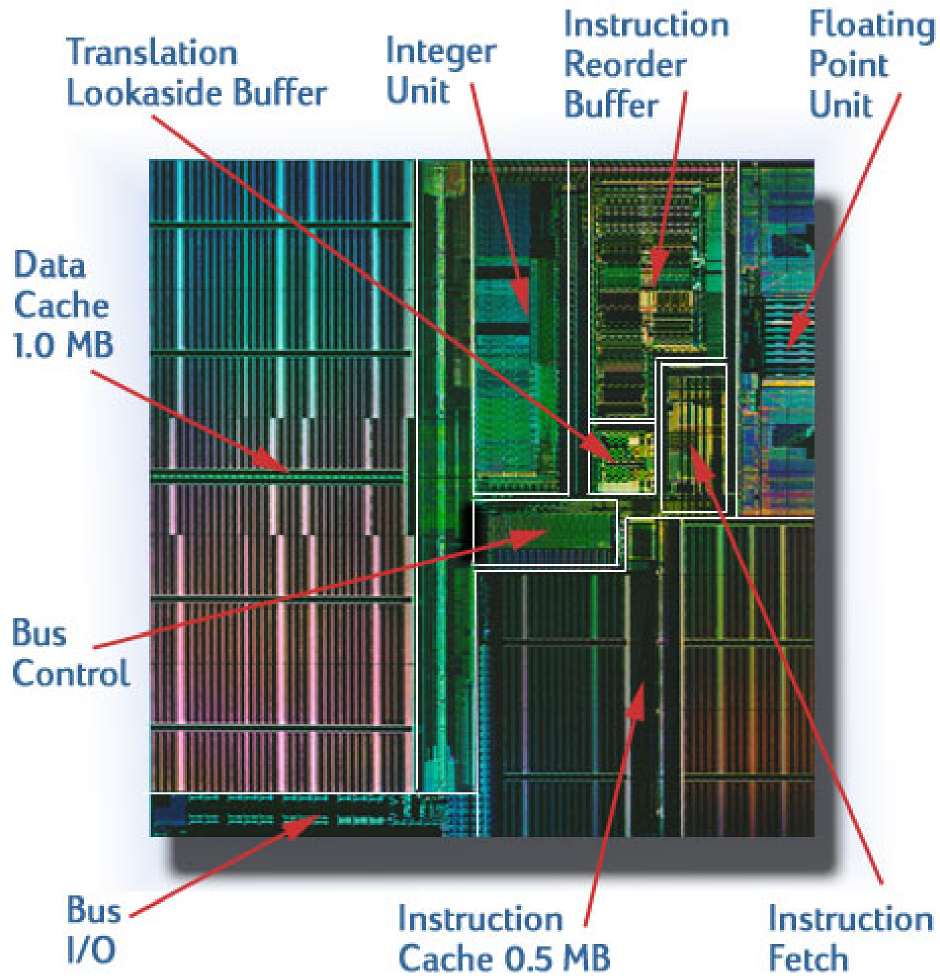
- **Move data faster (optimize speed)**
  - **Point-to-point wiring**
  - **Advanced protocols (e.g. clock in data)**
  - **Wide interfaces (256-bit GPUs)**
- **Move data less (optimize locality)**
  - **Algorithm**
  - **Architecture (e.g. pipeline GPU)**
  - **Cache data**

Slide courtesy of Kurt Akeley

Microprocessors Are All Cache!

Locality optimized
using cache memory

	CAGR	Growth in Decade
CPU →	1.5	58
	1.75	270
	2.0	1024
GPU →	2.25	3325
	2.5	9537



PA-8500 microprocessor

Slide courtesy of Kurt Akeley

What does this mean for games?

- CPU bound unless you balance the load!
- Start planning uses for GPU power now!
  - Obvious: more graphics detail
  - Not-so obvious:
    - Physics simulation,
    - global illumination
    - AI path finding?
    - Procedural animation

[James 2001], [Elder Scrolls III: Morrowind]

Goal: Harness GPU Power

- The cost of continued performance growth
  - Specialization allows constraints
  - Constraints enable optimization, but
  - Makes generalization non-trivial
- GPU not as easy to program as a CPU
  - Sometimes mappings are not obvious
  - I'll talk about specific techniques, building blocks, and examples

Outline

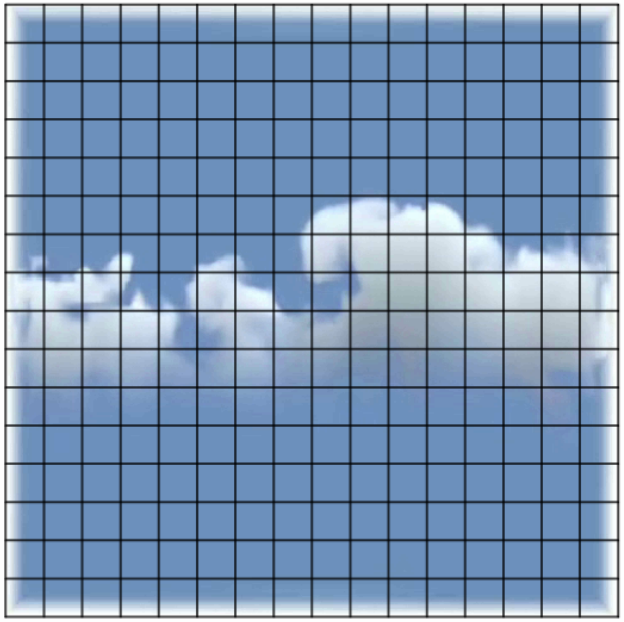
- Motivation: Why GPUs?
- **Mapping computational concepts to GPUs**
- Tricks of the trade: Branching Techniques
- Current Limitations
- New OpenGL Functionality
- The Future

Main Computational Resources

- Programmable parallel processors
  - Vertex & Fragment pipelines
- Rasterizer
  - Mostly useful for interpolating addresses (texture coordinates) and per-vertex constants
- Texture unit
  - Read-only memory interface
- Render to texture
  - Write-only memory interface

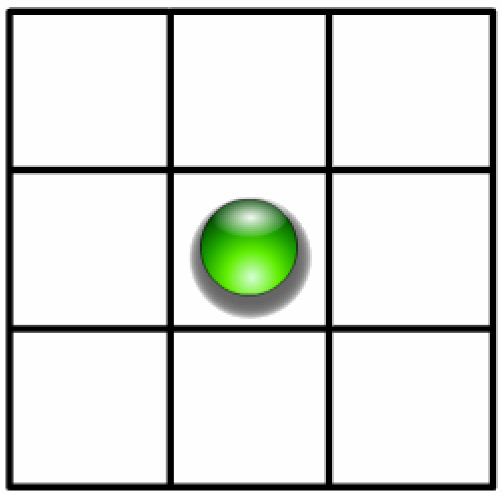
Array/Grid Computation

- Common GPGPU computation style
  - Textures represent arrays
- Many computations map well to grids
  - Matrix algebra
  - Image & Volume processing
  - Physical simulation
  - Global Illumination
    - ray tracing, photon mapping, radiosity
- Non-grid computations can often be mapped to grids



Scatter vs. Gather

- Grid communication
  - Grid cells share information



Vertex Processor

- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex
  - Cannot read info from other vertices
  - Can only read a small constant memory
- Future hardware enables gather!
  - Vertex textures

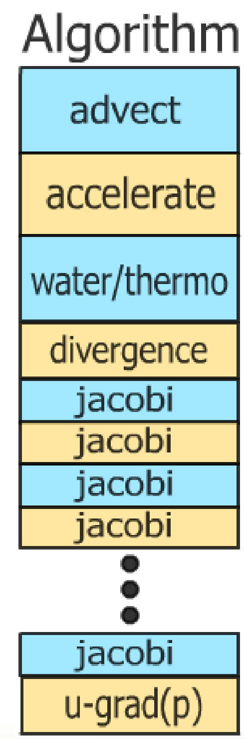
Fragment Processor

- Fully programmable (SIMD)
- Processes 4-vectors (RGBA / XYZW)
- Random access memory read (textures)
- Capable of gather but not scatter
  - No random access memory writes
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
  - More fragment pipelines than vertex pipelines
  - RAM read
  - Direct output

CPU-GPU Analogies

GPU Simulation Overview

- Analogies lead to implementation
  - Algorithm steps are fragment programs
    - Computational “kernels”
  - Current state variables accessed from textures
  - Feedback via Render to texture



Invoking Computation

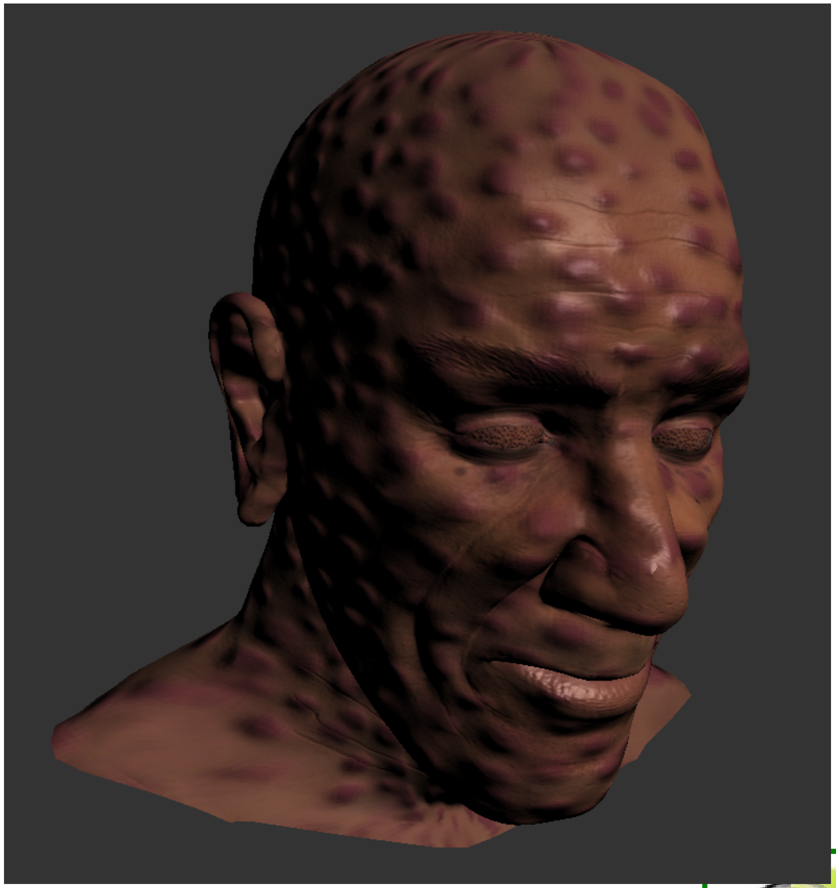
- Must invoke computation at each pixel
  - Just draw geometry!
  - Most common GPGPU invocation is a full-screen quad

Standard “Grid” Computation

- Initialize “view” (so that pixels:texels::1:1)
  - ```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 1, 0, 1, 0, 1);
    glViewport(0, 0, outTexResX, outTexResY);
    ```
- For each algorithm step:
  - Activate render-to-texture
  - Setup input textures, fragment program
  - Draw a full-screen quad (1x1)
    ```

Example: “Disease”

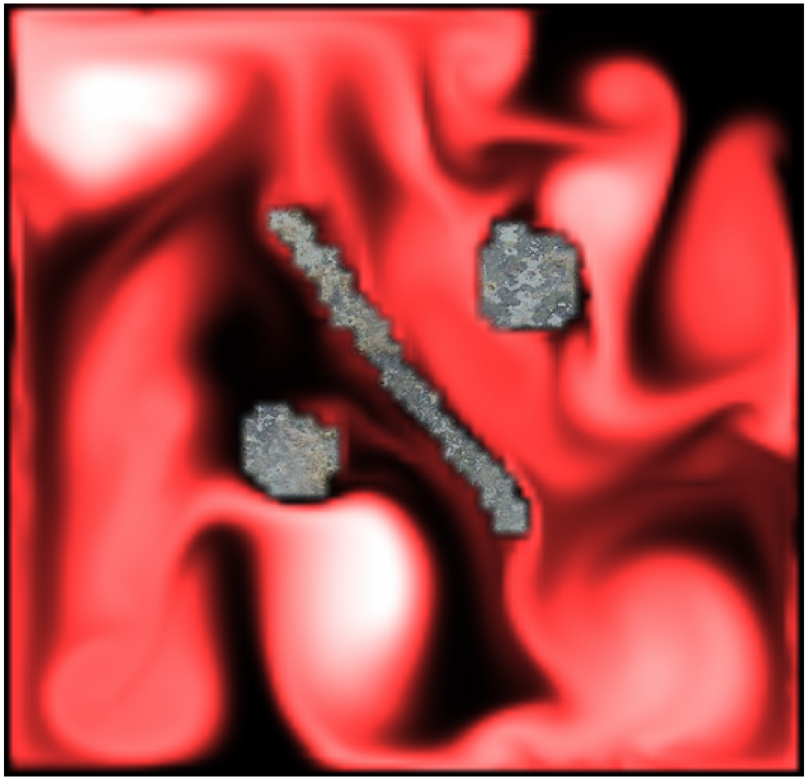
- Chemical reaction-diffusion simulation
  - Generate dynamic normal map from the result
- Add creepy effects to your characters!



[Harris & James, GDC 2003]

Example: Fluid Simulation

- Navier-Stokes fluid simulation on the GPU
- GPU Gems article:
  – "Fast Fluid Dynamics Simulation on the GPU"



Outline

- Motivation: Why GPUs?
- Mapping computational concepts to GPUs
- **Tricks of the trade: Branching Techniques**
- Current Limitations
- New OpenGL Functionality
- The Future

Branching Techniques

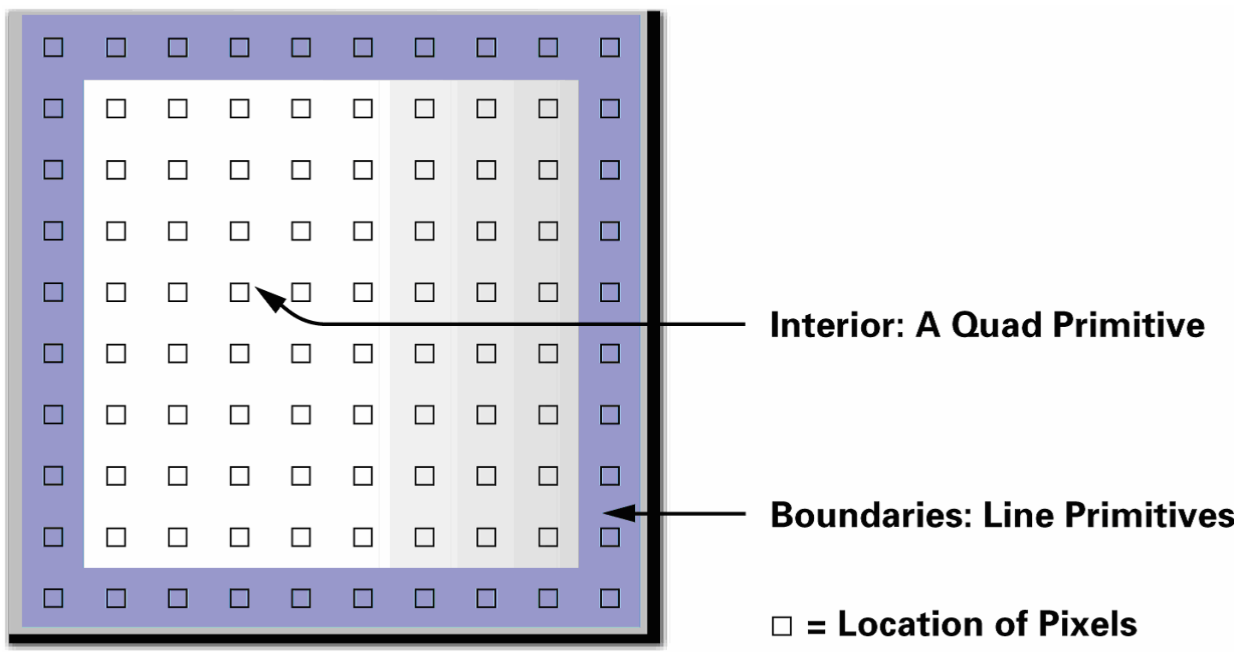
- Fragment program branches are costly
  - No true branching on NV3X & R3X0
  - Dynamic branches not cheap in near future
- Better to move decisions up the pipeline
  - Replace with math
  - Occlusion Query
  - Domain decomposition
  - Z-cull
  - Pre-computation

Branching with OQ

- Use it for iteration termination
  - Loop on CPU
    - Begin Query
    - Render with fragment program
      - In fragment program, discard fragments that match termination criteria
    - End Query
    - Terminate if query returns zero pixels
- Can be used for subdivision techniques
  - Demo later

Domain Decomposition

- Avoid branches where outcome is fixed
  - One region is always true, another false
  - Separate FPs for each region, no branches
- Example: boundaries



Z-Cull

- In early pass, modify depth buffer
  - Write depth=0 for pixels that should not be modified by later passes
  - Write depth=1 for rest
- Subsequent passes
  - Enable depth test (GL_LESS)
  - Draw full-screen quad at z=0.5
  - Only pixels with previous depth=1 will be processed
- Available in future GPUs
  - Depth replace disables Z-Cull on NV3X

Pre-computation

- Pre-compute anything that will not change every iteration!
- Example: arbitrary boundaries
  - When user draws boundaries, compute texture containing boundary info for cells
  - Reuse that texture until boundaries modified
  - Future hardware: combine with Z-cull for higher performance!

Outline

- Motivation: Why GPUs?
- Mapping computational concepts to GPUs
- Tricks of the trade: Branching Techniques
- **Current Limitations**
- New OpenGL Functionality
- The Future

Current GPGPU Limitations

- Programming is difficult
  - Limited memory interface
  - Usually “invert” algorithms (Scatter → Gather)
  - Not to mention that you have to use a graphics API…
- Limited bandwidth from GPU to CPU
  - PCI-Express will help
  - Frame buffer read can cause pipeline flush
  - Avoid large & frequent communication to CPU

Outline

- Motivation: Why GPUs?
- Mapping computational concepts to GPUs
- Tricks of the trade: Branching Techniques
- Current Limitations
- **New OpenGL Functionality**
- The Future

New Functionality Overview

- Vertex Programs
  - Vertex Textures: gather
  - MIMD processing: full-speed branching
- Fragment Programs
  - Looping, branching, subroutines, indexed input arrays, explicit texture LOD, facing register
- Multiple Render Targets
  - More outputs from a single shader
  - Fewer passes, side effects
  - “Deferred Computation”

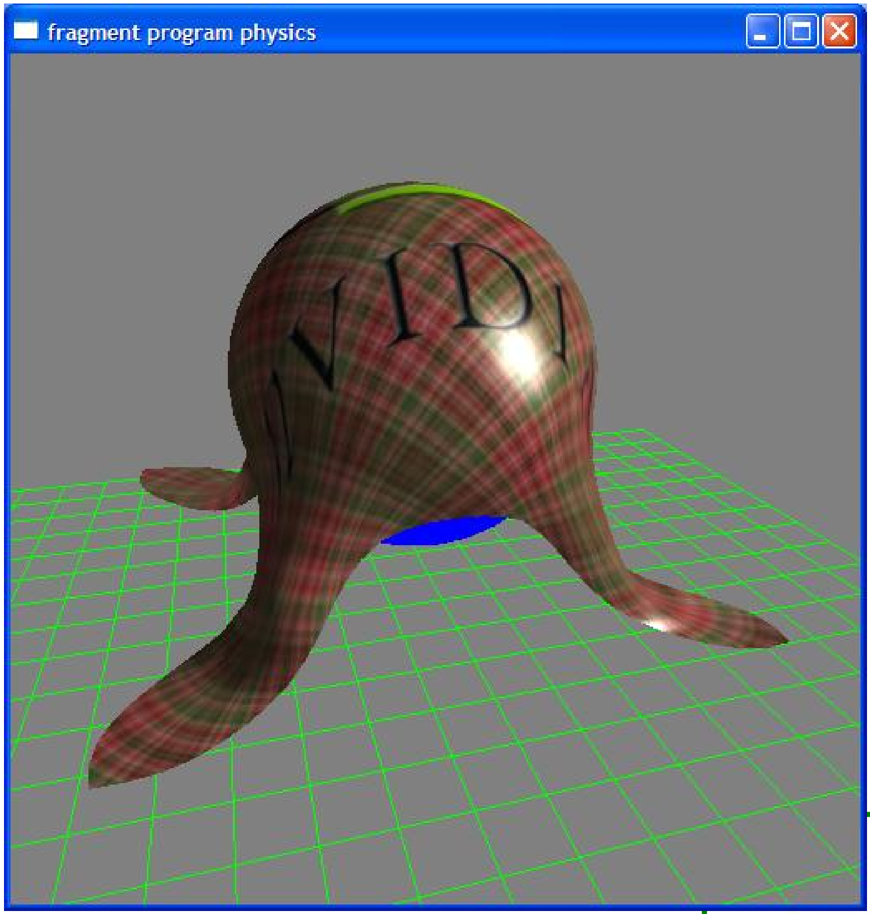
New Functionality Overview

- VBO / PBO & Superbuffers
  - Feedback texture to vertex input
  - Render simulation output as geometry
  - Not as flexible as vertex textures
    - No random access, no filtering
  - Demos
- PCI-Express
  - Faster data download from GPU to CPU

EXAMPLES

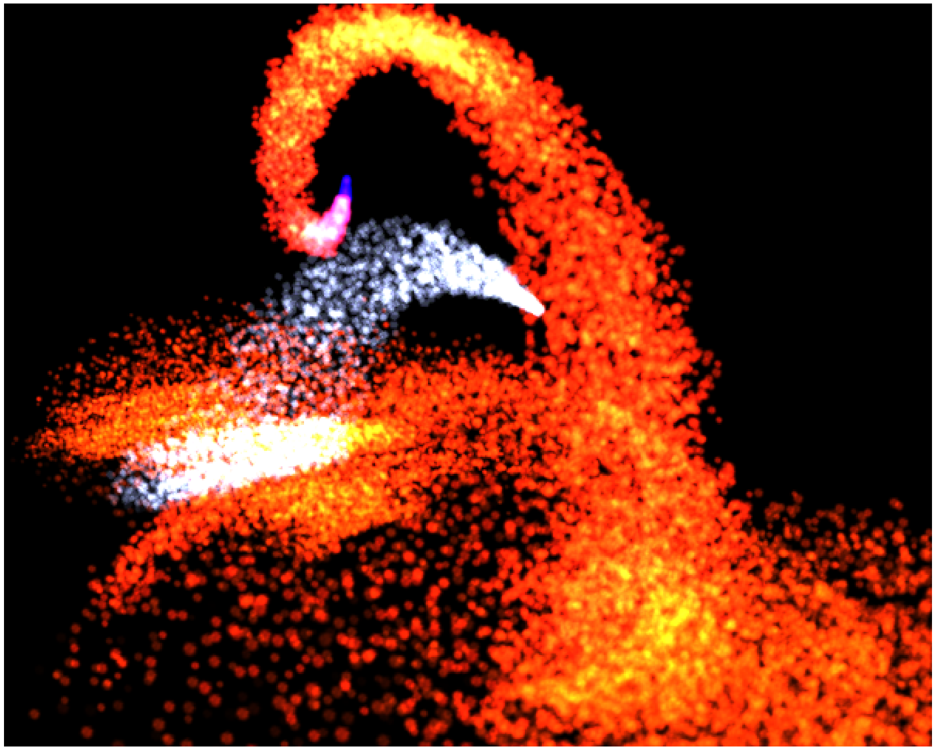
Example: Cloth Simulation

- Cloth Simulation
  - Simon Green
  - Simulation in fragment program
  - Use PBO/VBO to cast texture as vertex array for rendering

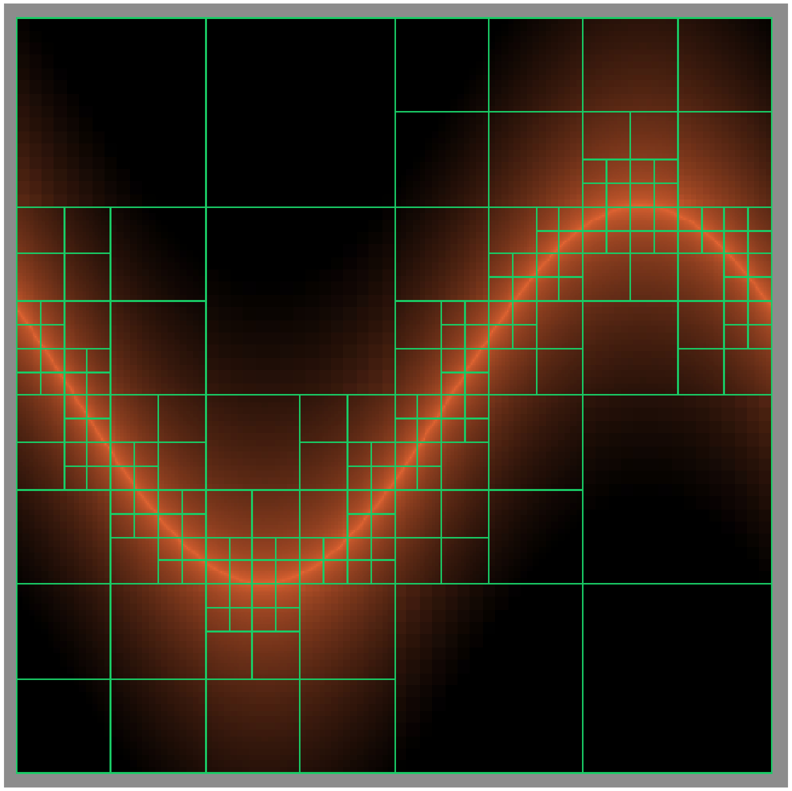


Example: Particle Simulation

- Lecture: "Building A Million Particle System"
  - By Lutz Latta, Wednesday at noon, GDC 2004



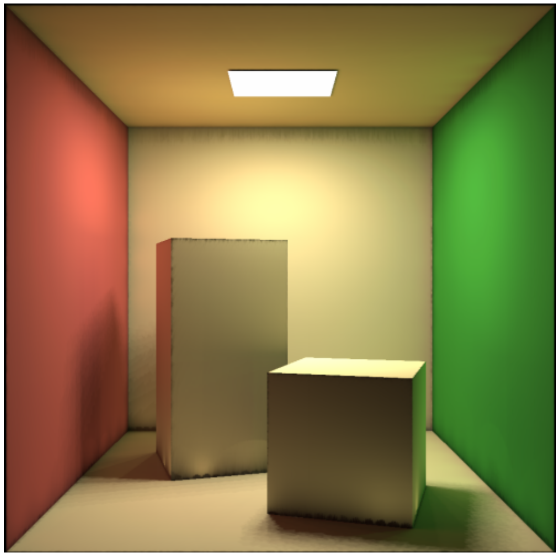
Example: OQ-based subdivision

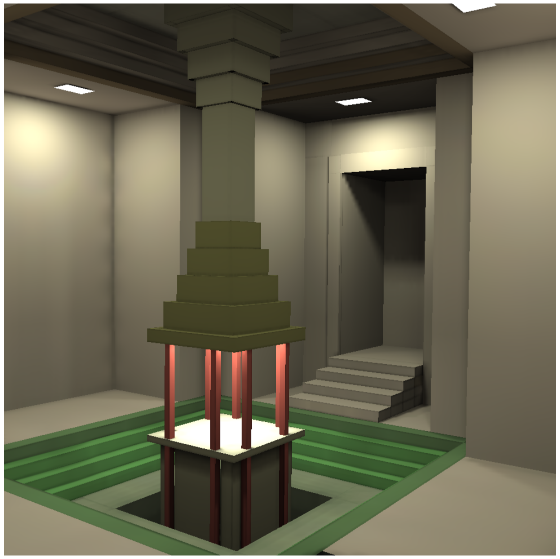


- Used in
  Coombe et al., “Radiosity on Graphics Hardware”

Example: GPU Radiosity

- Greg Coombe, UNC
- Progressive-refinement radiosity
- Uniform and adaptive solutions
- Hemisphere visibility (not hemicube)





The Future

- Increasing flexibility
  - Vertex textures (gather, feedback)
  - MRT (side effects)
  - Branching (especially in vertex programs)
- Easier programming
  - Non-graphics APIs and languages?
  - Brook for GPUs
    - http://graphics.stanford.edu/projects/brookgpu

The Future

- Increasing power
  - More vertex & fragment processors
  - GFLOPs, GFLOPs, GFLOPs!
    - Fast approaching TFLOPs!
    - Supercomputer on a chip
  - Start planning ways to use it!

- Massive multi-GPU Supercomputers?

More Information

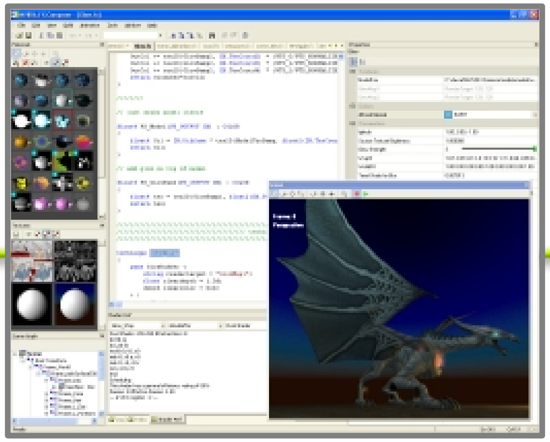
- GPGPU news, research links and forums
  - www.GPGPU.org
- SIGGRAPH 2004 GPGPU Course
  - Wednesday, full-day
  - Building blocks, advanced techniques & case studies

- Questions?
  - mharris@nvidia.com

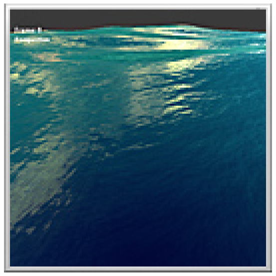
developer.nvidia.com

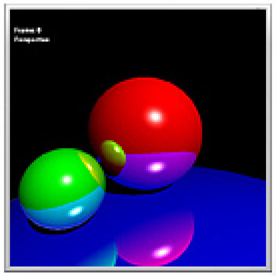
The Source for GPU Programming

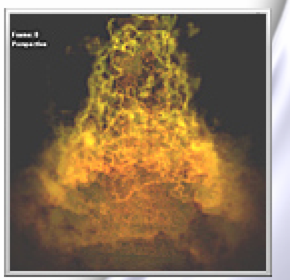
- Latest documentation
- SDKs
- Cutting-edge tools
  - Performance analysis tools
  - Content creation tools
- Hundreds of effects
- Video presentations and tutorials
- Libraries and utilities
- News and newsletter archives

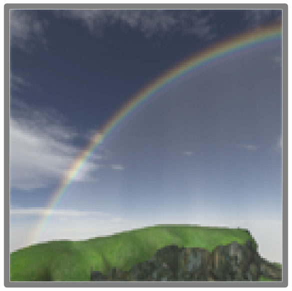


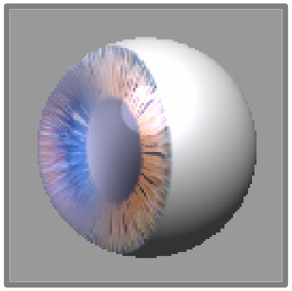
EverQuest® content courtesy Sony Online Entertainment Inc.



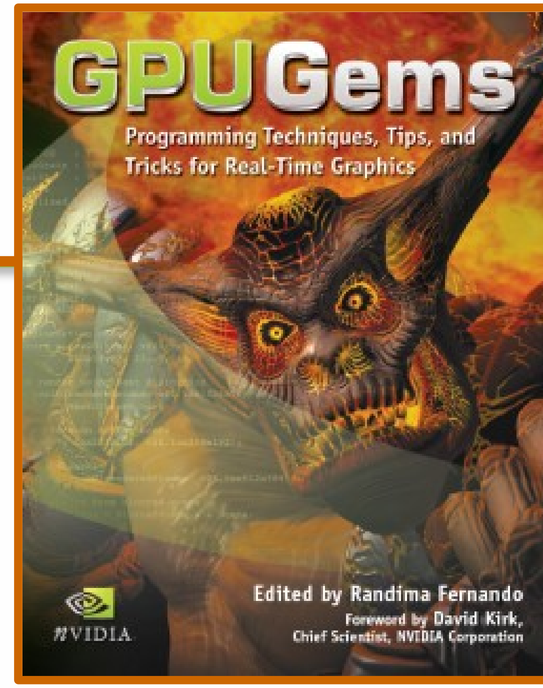








GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics



- **Practical real-time graphics techniques from experts at leading corporations and universities**
- **Great value:**
  - **Contributions from industry experts**
  - **Full color (300+ diagrams and screenshots)**
  - **Hard cover**
  - **816 pages**
  - **Available at GDC 2004**

For more, visit: http://developer.nvidia.com/GPUGems

"*GPU Gems* is a cool toolbox of advanced graphics techniques. Novice programmers and graphics gurus alike will find the gems practical, intriguing, and useful."

Tim Sweeney

Lead programmer of *Unreal* at Epic Games

"This collection of articles is particularly impressive for its depth and breadth. The book includes product-oriented case studies, previously unpublished state-of-the-art research, comprehensive tutorials, and extensive code samples and demos throughout."

Eric Haines

Author of *Real-Time Rendering*

Extra Slides Begin Here

GL_NV_vertex_program3

- Vertex Textures (TEX, TXP)
  - Up to 4 on NV40
  - Mipmaps (TXB, TXL: bias or explicit LOD)
  - GL_NEAREST filtering
- Indexed arrays of input / output attributes
- One additional condition code (2 total)
- PUSHA / POPA instructions
  - For subroutine call / return
- NV40: MIMD – full-speed branching.

GL_NV_fragment_program2

- Data-dependent branching
  - Static / dynamic branching
  - Fixed-iteration-count loops
  - Conditional loop break (BRK)
- Subroutine calls
- Explicit LOD texture lookup (TXL)
- Indexed input arrays
- Facing register (front / back)

Multiple Render Targets

- Write multiple RGBA results in FPs
- Reduce # passes by writing side-effects
  - Avoid duplicate computation computation
- “Deferred computation”
  - Like deferred shading, but for GPGPU
- See GL_ATI_draw_buffers spec

VBO / PBO & Superbuffers

- Flexible video memory allocation
- Vertex buffers and pixel buffers
- Specify usage at allocation time
  - Driver can optimize location and format
- Multi-use buffers possible
  - Closes the loop between fragment and vertex units!

PCI-Express

- With AGP, GPU to CPU transfers *slow*
  – Asymmetric bandwidth
- PCI-Express is symmetric
  – CPU-GPU bandwidth = 1.5x AGP 8x
  – GPU-CPU bandwidth = 5x AGP 8x!
- May be feasible to return GPU results to CPU

Render To Vertex Array

- Render to texture, use as vertex array
  – Allows feedback to vertex unit without CPU read back.
- Useful for simulation
  – Simulate physics in fragment programs
  – Render output as vertex arrays
- Demos:
  – Cloth simulation
  – Particle simulation