Module 4: Beyond Static Scalar Fields

Dynamic Volume Computation and Visualization on the GPU

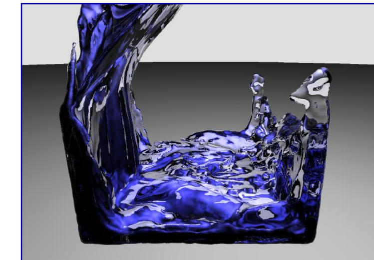
Aaron Lefohn

Visualization and Computer Graphics Group
University of California, Davis

Scientific Computing and Imaging Institute
University of Utah

Overview

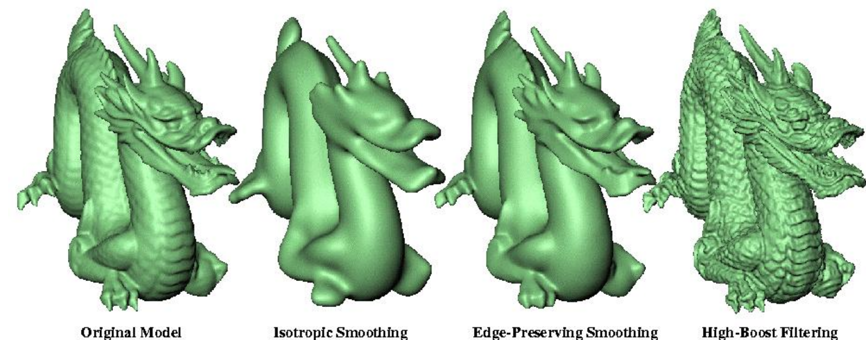
- Motivation and applications
- Challenges
  - Memory layout
    - Computation
    - Rendering
  - Load balancing
    - Available resources
    - Usage considerations
- GPU feature requests
- Conclusions



Why Dynamic Volumes?

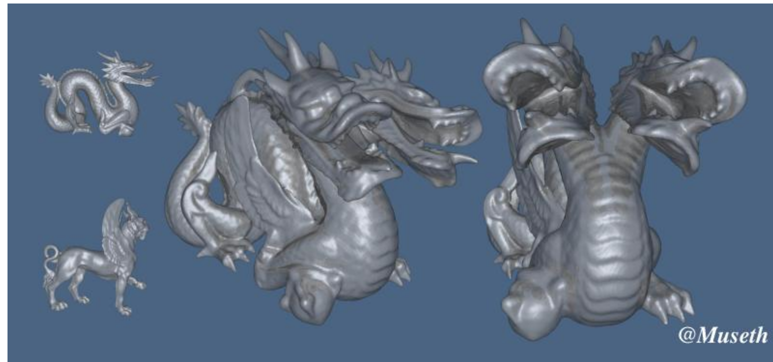
- Combine computation and visualization
  - Accelerate computation on GPU
  - Interactive visualization
  - Computational steering
  - Visual debugging
- Applications
  - 3D image and surface processing
  - Physical simulation
  - Implicit surface modeling

Application: Surface Processing



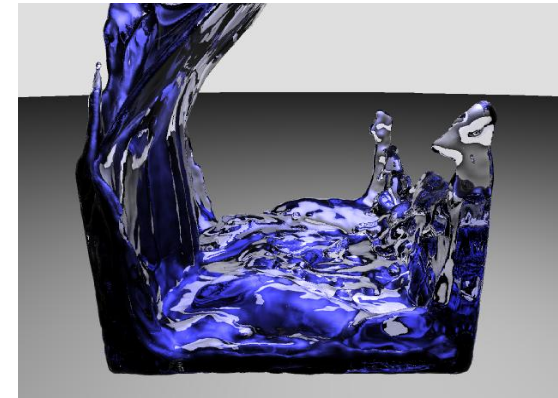
Tasdizen et. al., IEEE Visualization 2002

Application: Modeling



Museth et. al., SIGGRAPH 2002

Application: Physical Simulation

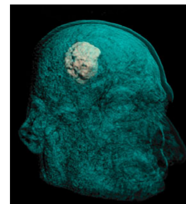


Premoze et. al., Eurographics 2003

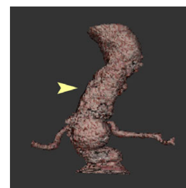
Interactive Visualization of Volumetric Data on Consumer PC Hardware:
Dynamic Volumes
Aaron Lefohn

Application: Segmentation

- "Interactive Deformation and Visualization of Level-Set Surfaces Using Graphics Hardware"
  - Lefohn, Kniss, Hansen, Whitaker
  - Wednesday at 10:30am



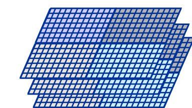
- "Fast Volume Segmentation and Simultaneous Visualization using Programmable Graphics Hardware"
  - Sherbondy, Houston, Napel
  - Wednesday at 3:15pm



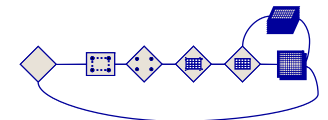
Interactive Visualization of Volumetric Data on Consumer PC Hardware:
Dynamic Volumes
Aaron Lefohn

Dynamic Volume Challenges

- Memory layout
  - Full volume
  - Sparse volume



- Load balancing
  - CPU
  - AGP bus
  - GPU vertex processor
  - GPU rasterizer
  - GPU fragment processor



Interactive Visualization of Volumetric Data on Consumer PC Hardware:
Dynamic Volumes
Aaron Lefohn

Sparse Volume: Computation

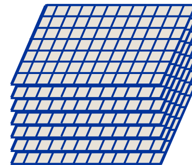
- Full volume
  - Computation performed on *all* voxels each computation step
  - Examples
    - Navier Stokes
    - Volume light transport
- Sparse volume
  - Computation performed only on *subset* of voxels
  - Examples
    - Surface embedding

Memory Layout: Computation

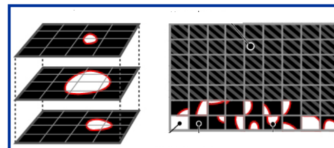
- How to update volume?
- GPUs only output 2D
  - Copy-to-texture from frame buffer
  - Render-to-texture with pbuffer
  - Render-to-texture with Uber-buffer
- Update one slice per render pass
- Multiple render targets (MRT)
  - Write four RGBA outputs per pass
  - ATI Radeon 9600/9700/9800

Memory Layout: Computation

- Full volume
  - Stack of 2D slices
  - Render-to-3D texture
    - Render each slice separately
    - Uber-buffers



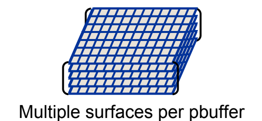
- Sparse volume
  - Active research topic
  - Two papers at Vis03
    - Lefohn et al. and Sherbondy et al.
  - Store full volume: Depth culling
  - Store only sparse domain



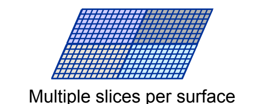
Full Volume: Computation

- Pbuffer layout
  - Assume RGBA
  - Changing pbuffers is slow, changing surfaces is fast

Layout option	Saves memory	Reduces context switches	Minimize program complexity
Pack data into RGBA	Yes	Yes	No
Multiple surfaces per pbuffer	No	Yes	Yes
Multiple slices per surface	No	Yes	No



Multiple surfaces per pbuffer



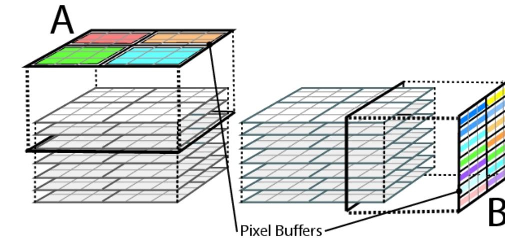
Multiple slices per surface

Full Volume: Computation

- Uber-buffers layout
  - No context switches
  - Obviates need for many memory tricks
  - Render-to-3D texture
  - See “OpenGL Extensions -- Siggraph 2003“
    - Rob Mace and James Percy
    - http://www.ati.com/developer/techpapers.html

Full Volume: Volume Rendering

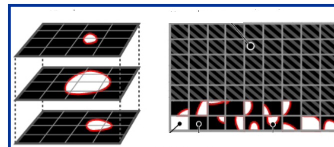
- Pbuffers
  - 2D texture-based volume rendering
    - Case A: Render slices
    - Case B: Reconstruct slice using lines from each slice
      - See Lefohn/Kniss et al. TVCG pre-print



- Uber-buffers
  - 3D texture volume rendering

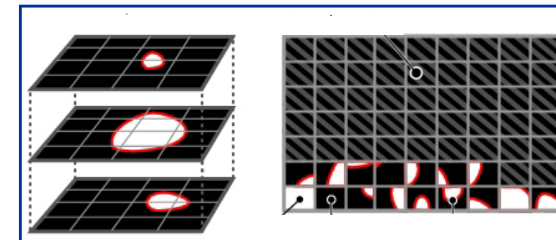
Sparse Volume: Computation

- Store full volume
  - Culling limits computational domain
    - Depth, mesh, stencil
  - Use full-volume storage techniques
  - See Sherbondy et al. talk on Wednesday at 3:15pm
    - Application of depth culling and uber-buffers
- Store sparse domain
  - Pack active voxels into 2D texture
  - Compute 3D computation on dynamic 2D representation
  - Lefohn et al. on Wednesday at 10:30am
    - Multi-surface pbuffers



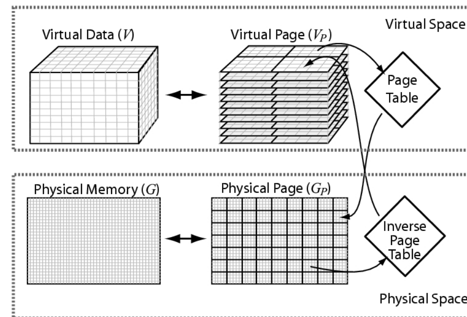
Sparse Volume: Computation

- Problem
  - 3D computation/visualization domain
  - 2D memory representation



Sparse Volume: Virtual Memory

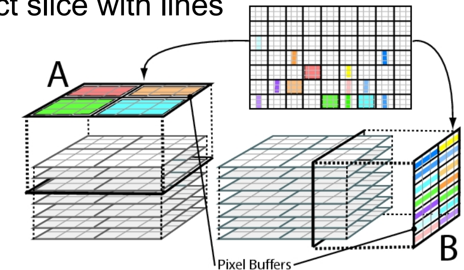
- Solution
  - Multi-dimensional virtual memory abstraction
    - 3D virtual memory space
    - 2D physical memory space



http://www.sci.utah.edu/~lefohn/work/rls/visLevelSet/lefohn_tvcg03.pdf

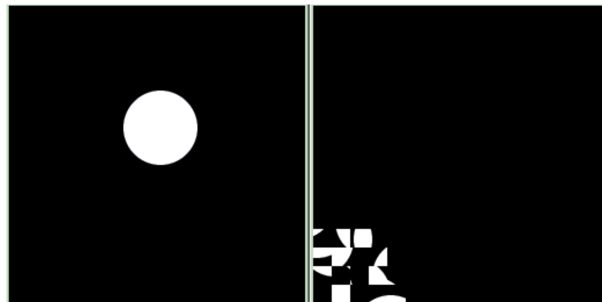
Sparse Volume: Rendering

- Store full volume / depth culling
  - Same rendering techniques as full volume
- Store sparse domain
  - Case A: Reconstruct slice with quadrilaterals
  - Case B: Reconstruct slice with lines



Sparse Volume: Rendering Example

- Sparse isosurface representation
- Reconstruction of a slice from packed data



Memory Layout Summary

- Full volume
  - Pbuffers
    - Pack scalar data into RGBA
    - Multi-surface pbuffers to avoid context switching
  - Uber-buffers
    - Render-to-3D texture
    - No context switching
- Sparse volume
  - Store full volume
    - Depth culling for sparse computation
  - Store sparse volume
    - Multidimensional virtual memory

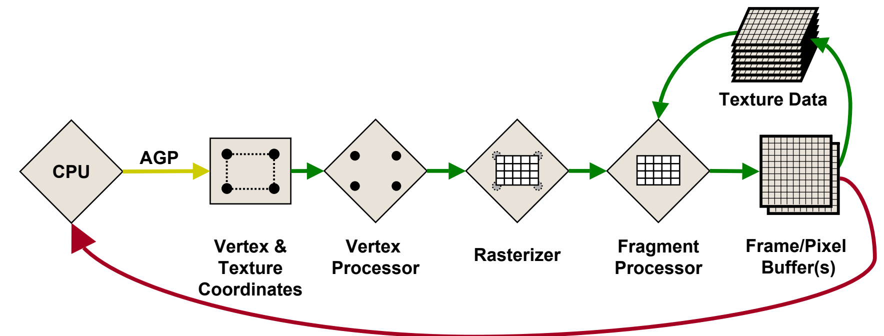
Load Balancing

- What are GPUs designed for?

Half-Life 2, Valve Software

Load Balancing

- Distribute work to under-utilized resources
  - Lighten fragment processor load
- Avoid slow pathways
- Consider computational frequency



Load Balancing: General Concepts

- Basics
  - Resources operate in parallel
  - If single resource dominates, using the others is "free"
  - Find uses for under-utilized resources
  - Additional information
    - "Graphics Pipeline Performance"
    - http://developer.nvidia.com/object/GDC_2003_Presentations.html

- Remember why GPUs are fast…

Remember Why GPUs are Fast

- Raster graphics is embarrassingly parallel
- Independent processing of elements
- Pre-fetch and cache coherence
- Little or no branching
- Fragment programming is limited for a reason

- Think about the fixed-function OpenGL pipeline

Load Balancing: Fragment Processor

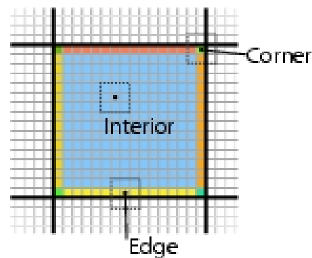
- Fragment processor most powerful
  - Often dominates computation and rendering time
  - Highest computational frequency
  - Use dependent texture reads sparingly
    - Balance lookup tables with computation
  - Can values be computed at lower frequency?
  - Avoid conditionals...

Load Balancing: Fragment Conditionals

- All branches of conditionals are executed
  - Polluted caches
  - Useless instructions
  - Useless memory reads
- *Might* change on future hardware

Load Balancing: Substreams

- Resolve fragment conditionals into cases
  - Draw each case as separate geometry batch
  - Fragment program optimized for each case
  - Example
    - Boundary conditions



  - References
    - Lefohn et al., IEEE Visualization 2003
    - Goodnight et al., Graphics Hardware 2003
    - James and Harris, Game Developers Conference 2003

Load Balancing: Vertex Processor

- Vertex processor often idle
  - Lift fragment ops when result can be linearly interpolated
  - Pass result to fragment processor as per-vertex-interpolant
  - Adds load to rasterizer
  - Lessens AGP and fragment load
- Example
  - Neighbor memory address computations

Load Balancing: AGP

- Minimize CPU → GPU traffic
  - Use vertex buffers objects if can spare GPU memory
  - Generate texture coordinates with vertex processor
- Minimize GPU → CPU traffic
  - Does CPU need all data from GPU?
  - Use GPU to produce concise message
    - Reduction using mipmapping or down-sampling shader
  - Use GPU computation to save AGP bandwidth
  - Specific application: Bit-code image
    - TVCG pre-print http://www.sci.utah.edu/~lefohn/work/rls/visLevelSet/lefohn_tvcg03.html

Load Balancing: CPU

- CPU strengths
  - Pre-computing GPU-invariant computation
  - Updating complex data structures
  - Complex logic
- Use for operations not suited-for or supported-by GPU
  - Communication must be minimized
  - Using CPU *may* accelerate solution faster than running entirely on GPU

Load Balancing Summary

- Fragment stage is often bottleneck
  - Statically resolve conditionals with substreams
- Compute texture addresses with vertex stage
  - Reduces AGP traffic and fragment instructions
- Minimize GPU → CPU communication
  - Use GPU computation to produce minimal message
- Leverage CPU if necessary
  - GPU-only solution not necessarily faster

GPU Feature Requests

- Uber buffers
  - Render-to-3D texture
  - Fast changing of render targets (no context switch)
  - Interchangeable depth/stencil/color buffers
- More interpolants
  - Compute texture coordinates with vertex processor
- Global accumulation registers
  - min, max, sum, norm, etc.
- Integer data type
  - Bitwise operations for compression, message passing

GPU Programming Tools Requests

- GPU profiling tools for OpenGL
- Vertex and fragment program debuggers
  - Fragment debugger: Tim Purcell and Pradeep Sen at Stanford

Dynamic Volume Summary

- Marriage of computation and visualization
  - "Putting a face on computation"
  - Computational steering
  - Important visualization application
- Memory layout
  - Full and sparse volume formats
  - Rendering considerations
- Load Balancing
  - Maximize resource utilization

Future Directions

- User interface design for interactive computation
  - Batch processing ⟶ Intuitive control
- Separate computation from memory layout
  - N-D Multidimensional virtual memory manager

Thank you

- Questions?