Volume Ray Casting Techniques and Applications using General Purpose Computations on Graphics Processing Units

Michael Romero

M.S. Thesis Defense
Department of Computer Engineering, RIT
June 16, 2009

Thesis Goals

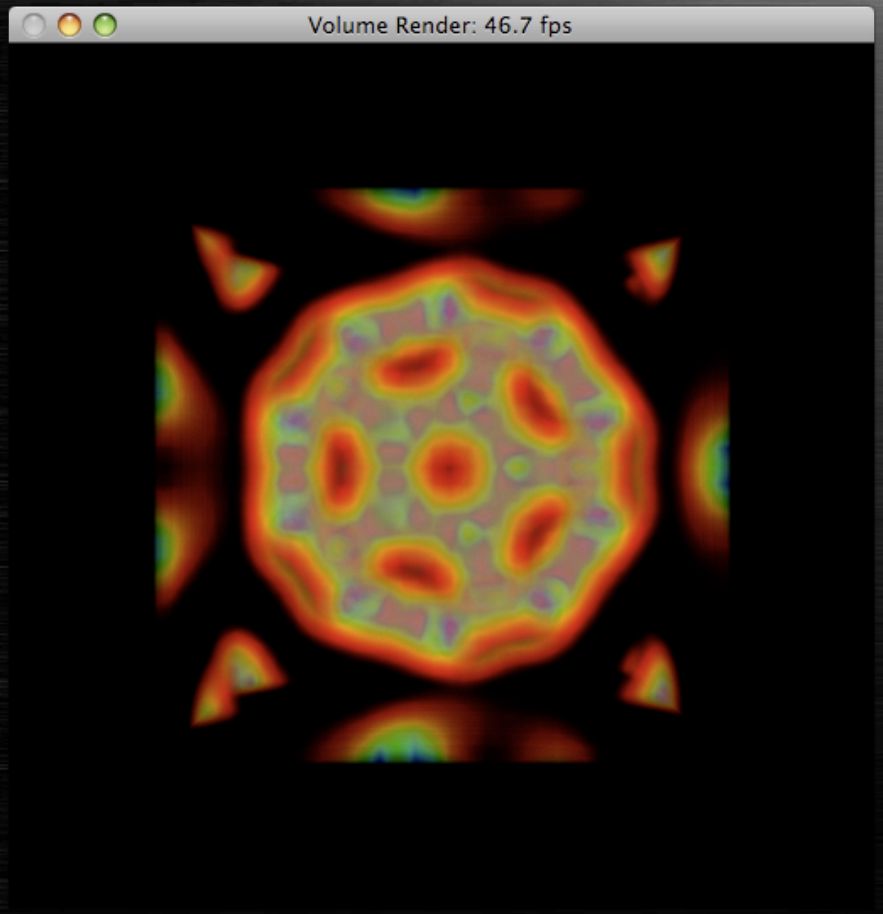
- Interactive volume ray casting on CUDA
- Performance and quality enhancements
  - Early ray termination
  - Supersampling
  - Texture filtering
- Exploration of volume rendering applications
  - Multiple volume rendering
  - Hypertextures
  - Stereoscopic anaglyphs
- Extensible and portable rendering framework

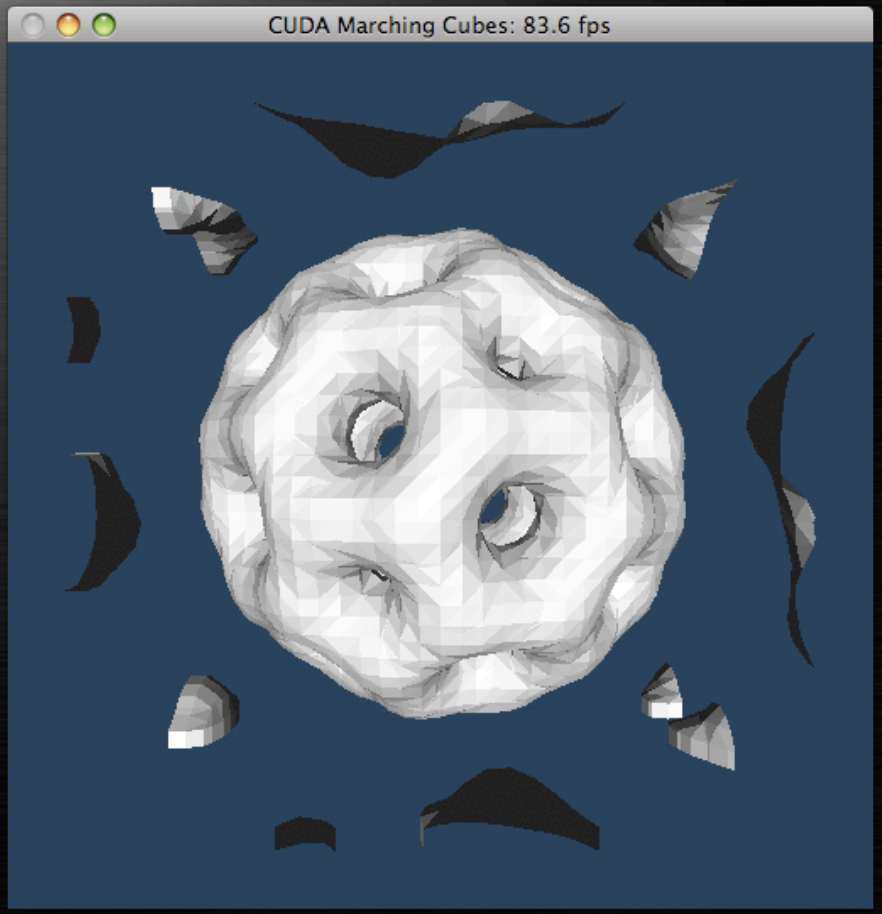
Outline

- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

What is Volume Rendering?

- Purpose of volume rendering
  - Allows visualization of interior of objects
  - Differs from traditional 3D computer graphics, which only describe an objects surface
- Representation of volumes
  - Volumes represented using “voxels”
  - Voxels typically store density information
  - Optical models assign color and opacity
- Rendering volumes
  - Ray casting technique, also used in ray tracing
  - Slicing technique





Volume Rendering

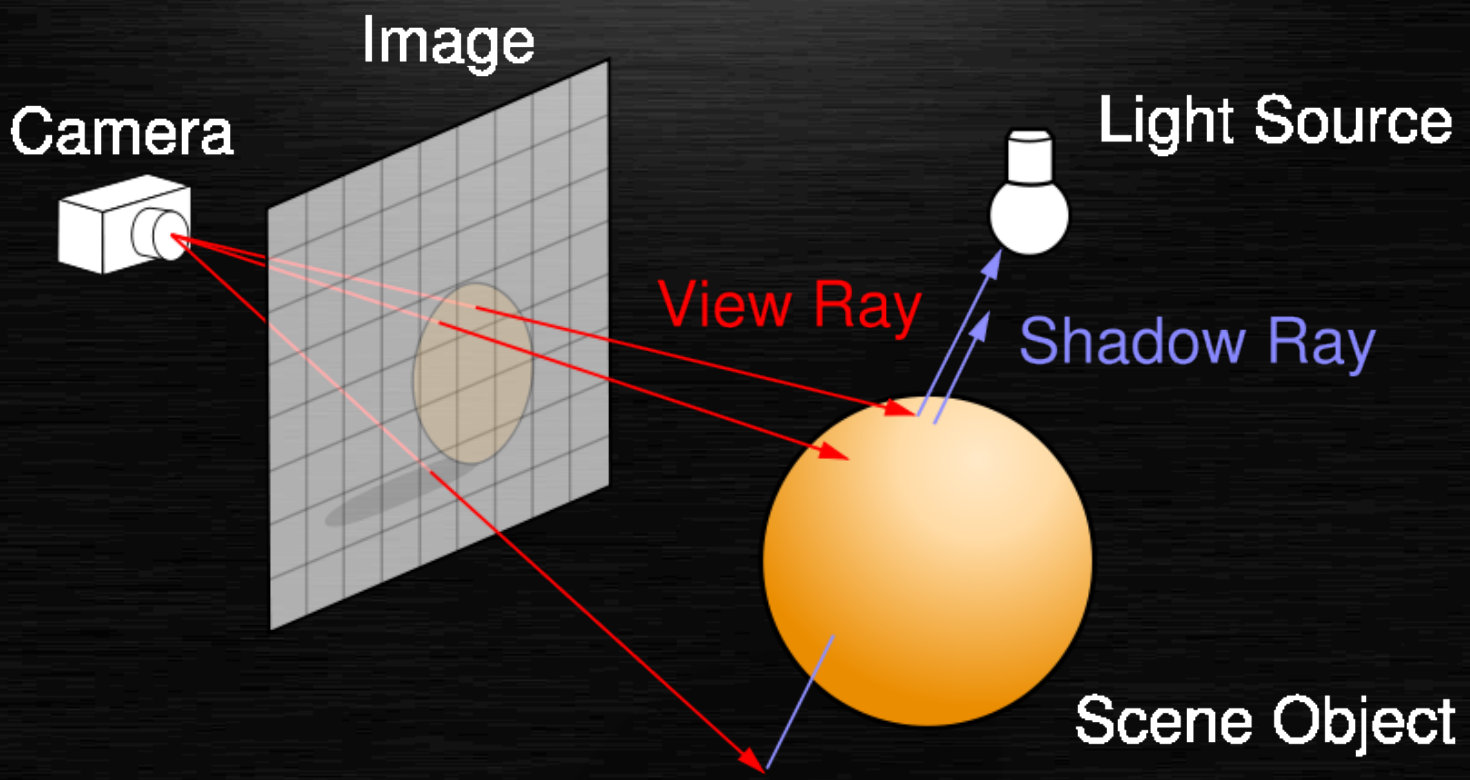
Triangular Mesh

Ray Casting vs. Slicing

- Ray casting is embarrassingly parallel
- Slicing techniques may exhibit aliasing
- Ray casting has a large number of easily implementable performance and quality enhancements
- 3D Textures may be used in both ray casting and slicing techniques
- Ray casting is ideal for a CUDA implementation and the goals of this thesis

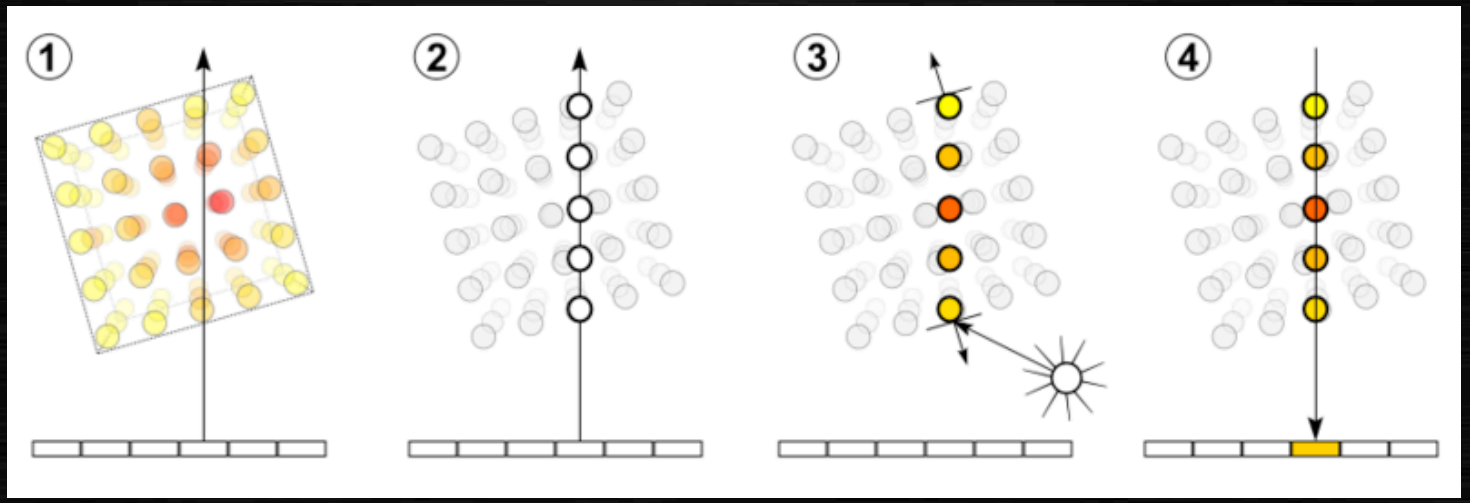
Ray Casting

- Fire ray from a camera, through a viewplane
- Each ray tests intersection with bounding box
- If intersected, ray march within bounding box



Ray Marching

- Step through volume using a given step size
- Sample volume at each point
- Calculate shading if applicable
- Composite the sampled points



Outline

- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

GPGPU

- General-Purpose Computing on Graphics Processing Units
- Affordable, commodity off the shelf high performance computing
- Massively parallel processing
- Excellent price-performance ratio
- Low peak GFLOP/s per watt
- Rapidly developing field in computing

What is CUDA?

- A toolkit for programming NVIDIA graphics cards to perform general purpose computations (GPGPU)
- Most prominent GPGPU platform currently available
- Requires CUDA capable device
  - 8 Series or better, includes mobile platforms
- Requires CUDA specific drivers
  - Download at nvidia.com/cuda

Why do we use CUDA?

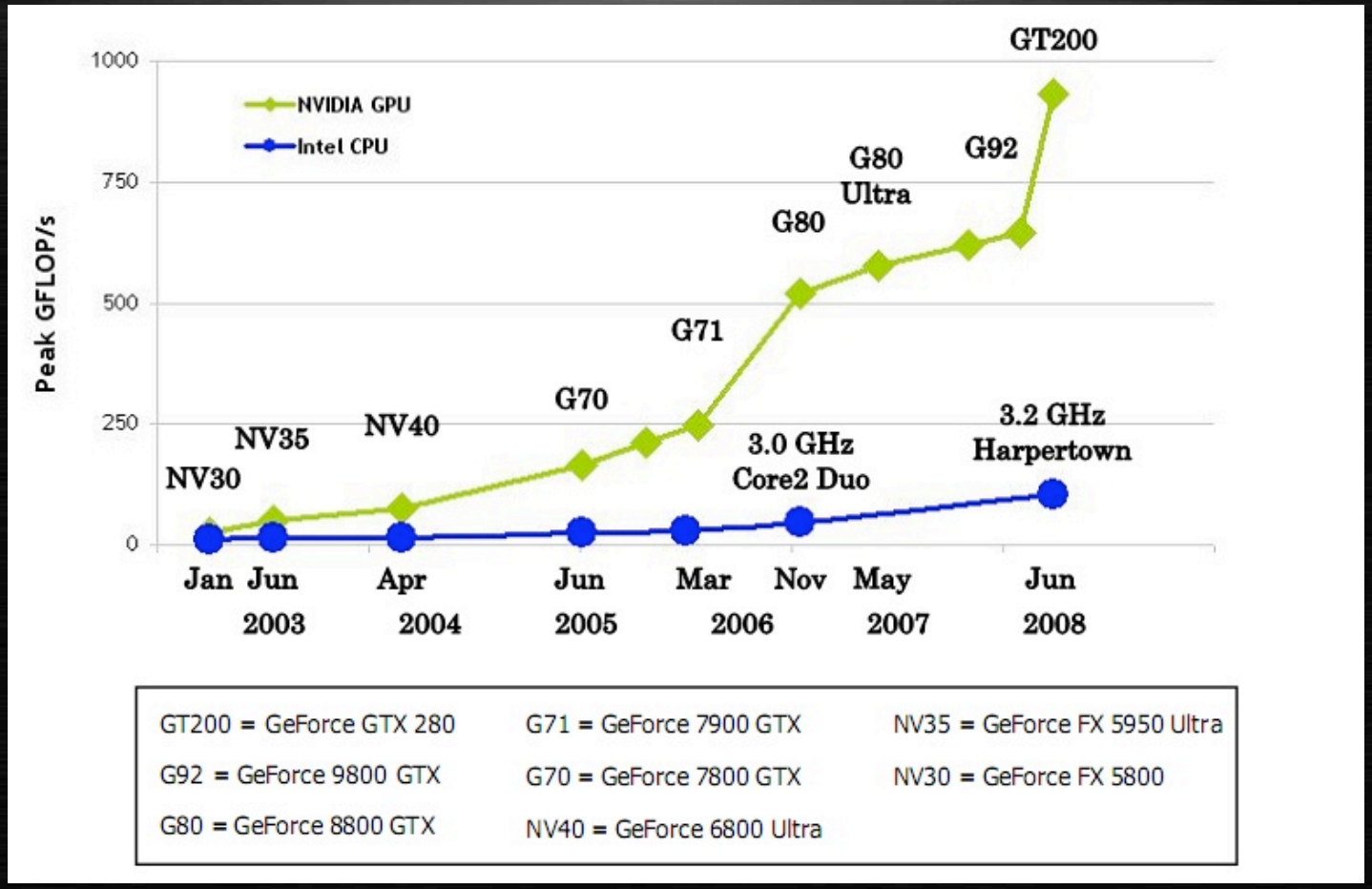


Image Courtesy of NVIDIA

Outline

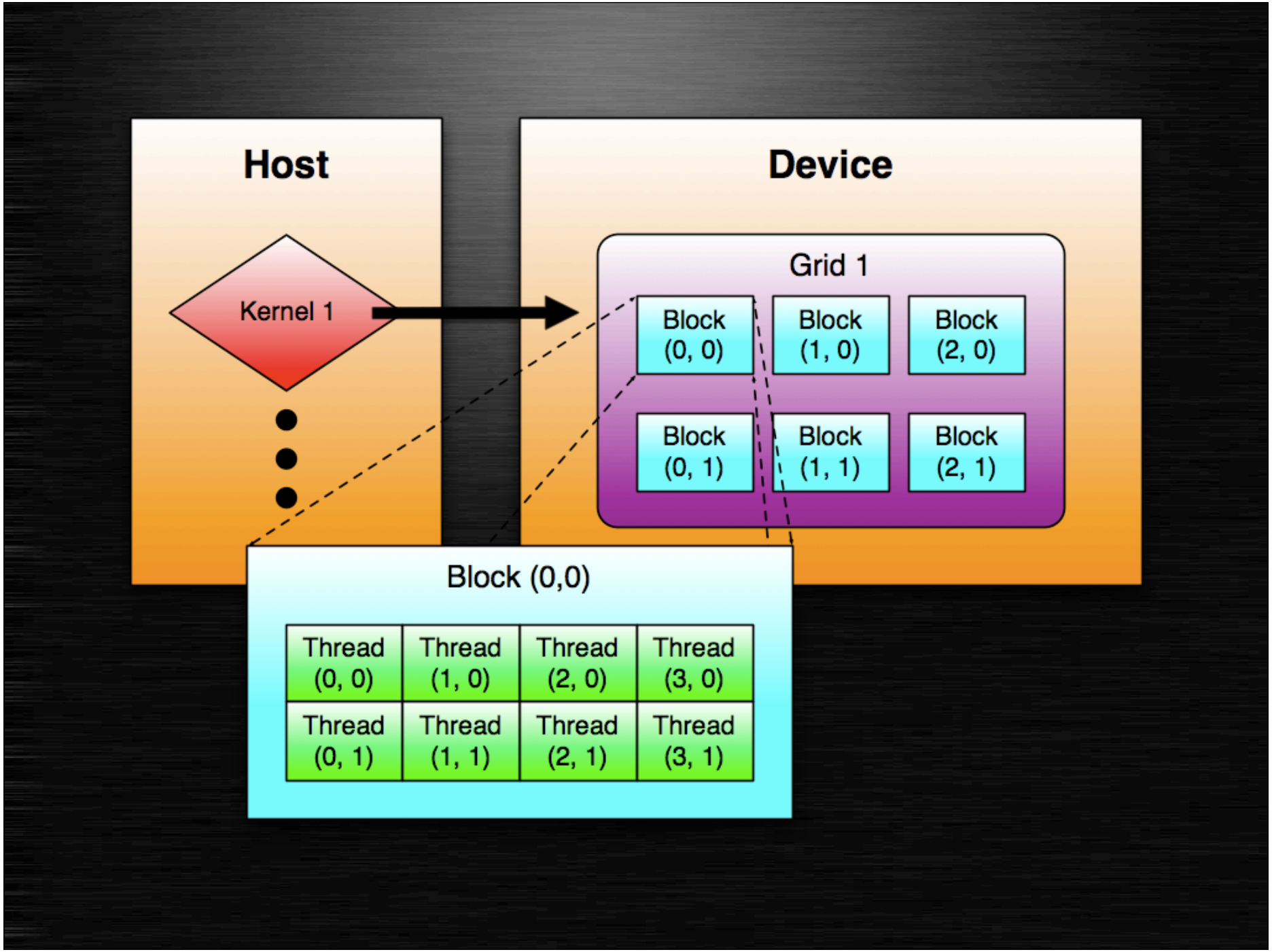
- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

How it Works

- The host farms out work to be done on the device by calling a ‘kernel’
- A kernel is SIMT (Single Instruction Multiple Thread), with multiple threads executing simultaneously
- Threads are the most basic level of the computational hierarchy
- A shared memory system, but not all threads share the same memory space

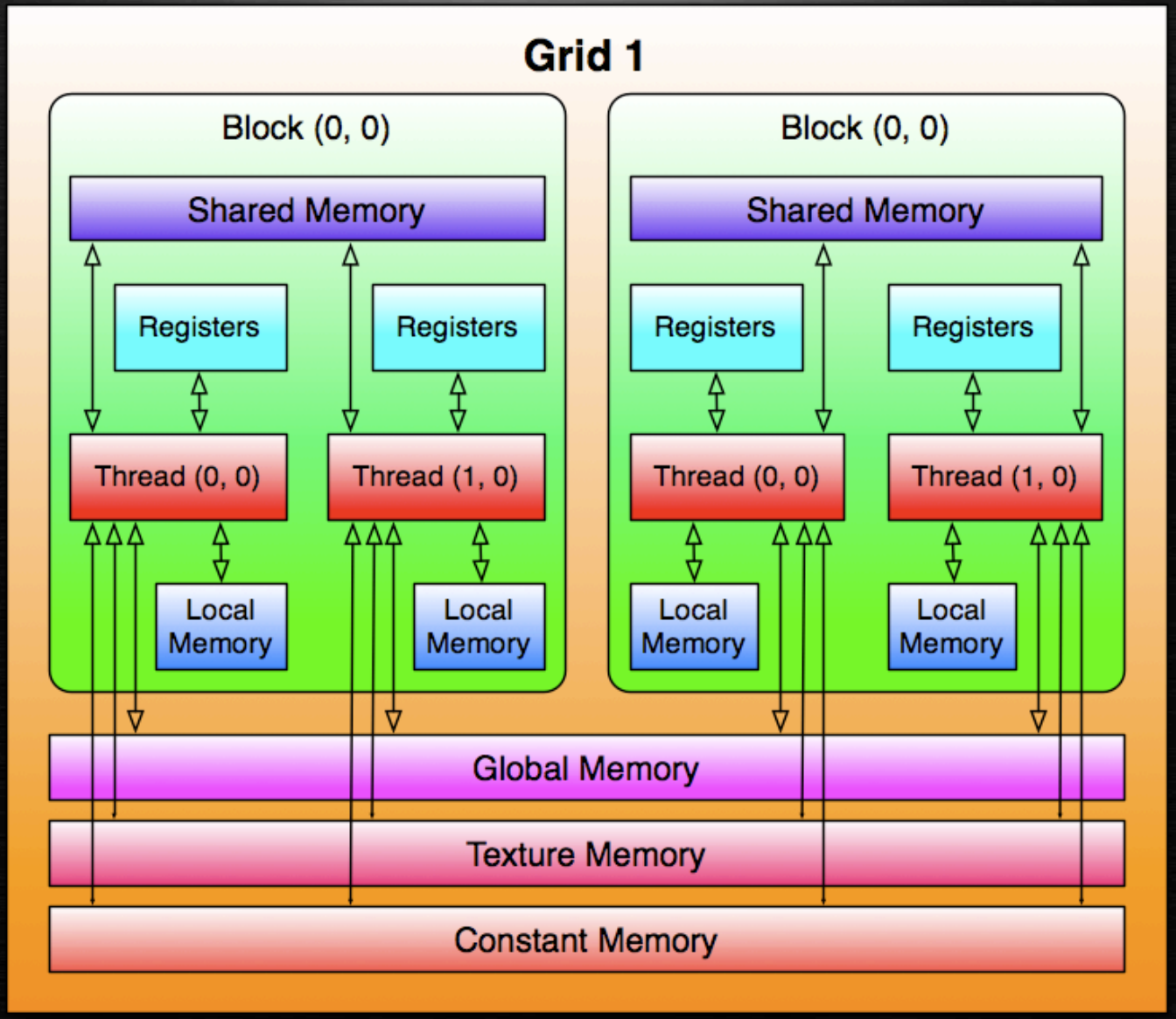
CUDA Architecture

- Threads are grouped into ‘blocks’, which have 16KB shared memory between threads
- There is a limit of the number of threads (512) that can be fit into a block
- Blocks are dynamically issued to be executed on the device
- A ‘grid’ consists of multiple blocks, with no limit on the number of blocks per grid


Host
Device
Kernel 1
Grid 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (0,0)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)

Memory System

- Registers per thread
- Local memory per thread located in main memory
- Shared memory per thread block
- Global memory located in main memory
- Texture memory in main memory – read only
- Constant memory in main memory – read only
- Host can read/write global, texture and constant memory
- cudaMalloc(), cudaFree(), cudaMemcpy()


Grid 1
Block (0, 0)
Shared Memory
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Block (0, 0)
Shared Memory
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Global Memory
Texture Memory
Constant Memory

Outline

- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

Volume Rendering on CUDA

- Ray casting is embarrassingly parallel, therefore easily exploitable by CUDA
- Each thread is responsible for a ray
- Volumes are stored in texture memory
- World structure is stored in constant memory
- Pixel Buffer Object is used so rasterized image is displayed using DMA
- Any CUDA device is capable of rendering volumes, given sufficient memory capacity

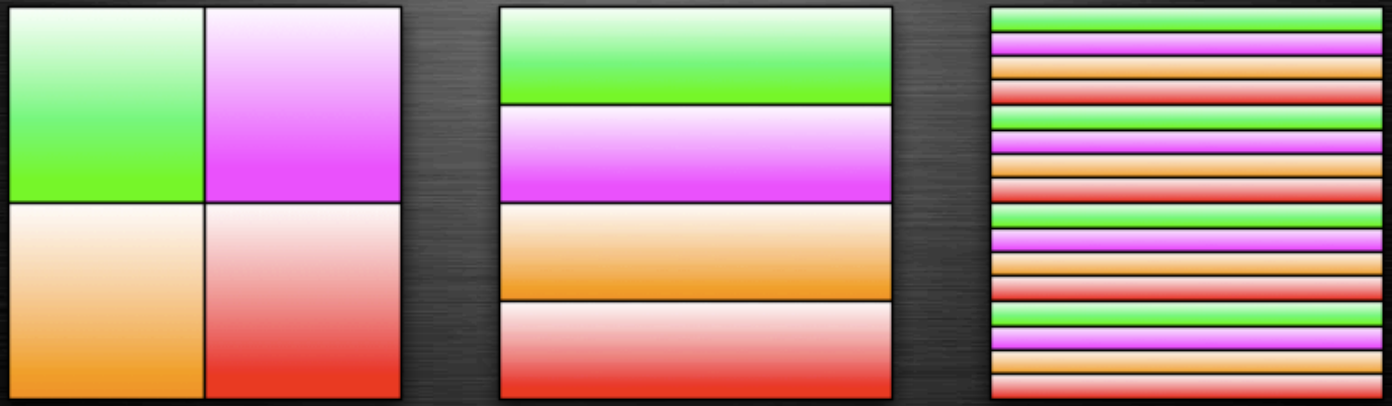
Design Considerations

- Load Balancing
- Multiprocesor Occupancy
- Register Usage
- Threads Per Block
- Memory Usage

Load Balancing

- One thread per ray, and assuming no supersampling, one ray per pixel
- Branching occurs if bounding box is not hit
- CUDA dynamically issues blocks to available multiprocessors
- Ideally, all threads in a block finish simultaneously
- Threads per block behaves like a “grain size”
- Dynamically allocated line segments provide the optimal load balancing

Static Partitioning



Block
(Nodes = 4)

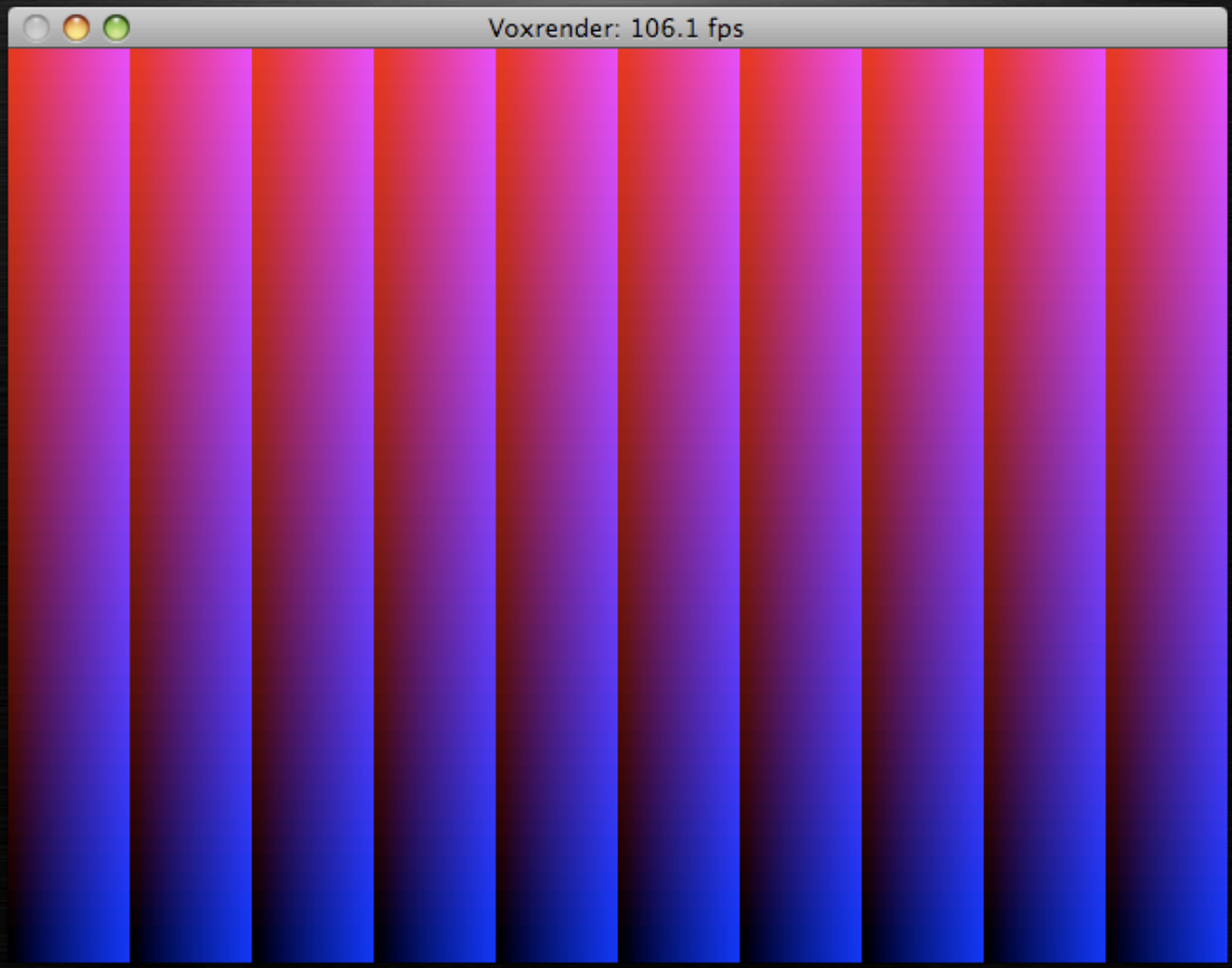
Strip
(Nodes = 4)

Cyclic
(Nodes = 4)

Dynamic Partitioning

Line Segment
(Grain = 9)

Square Block
(Grain = 9)



Blue: Thread ID

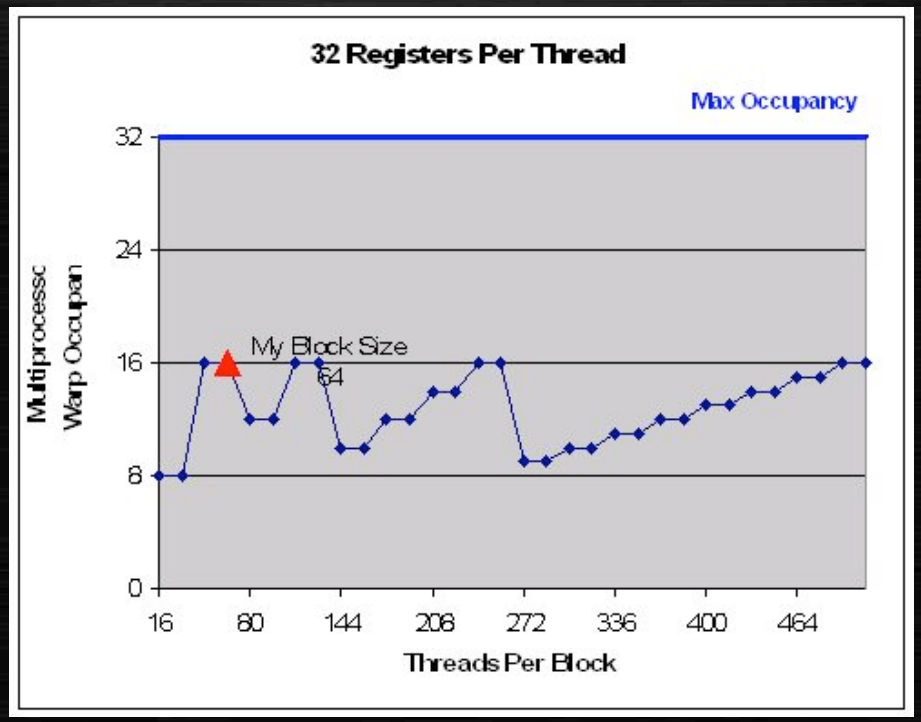
Red: Block ID

Multiprocessor Occupancy

- The ratio of simultaneous computations on a multiprocessor to maximum possible
- The greater the occupancy, the larger the number of available compute nodes
- A number of deciding factors:
  - Threads per block
  - Registers per thread
  - Shared memory usage
- No shared memory used in implementation

Register Usage

- Register usage per thread may be set at compile time
- Excess variables are placed in local memory
- Must balance poor local memory performance against multiprocessor occupancy for optimal performance



Threads Per Block

- Should be a multiple of 32 (size of a warp)
- Essentially the “grain size” for partitioning
- Also determines distribution of resources on a multiprocessor, affecting occupancy
- Experimentation demonstrates that 32 registers per thread and 64 or 128 threads per block yields optimal performance.

Memory Usage

- Constant Memory
  - Caching policy exploits temporal locality
  - Only writable by host, outside kernel execution
  - Low capacity
  - World structure resides in constant memory
- Texture Memory
  - Caching policy exploits spatial locality
  - Only writable by host, outside kernel execution
  - High capacity
  - Volume data resides in texture memory

Outline

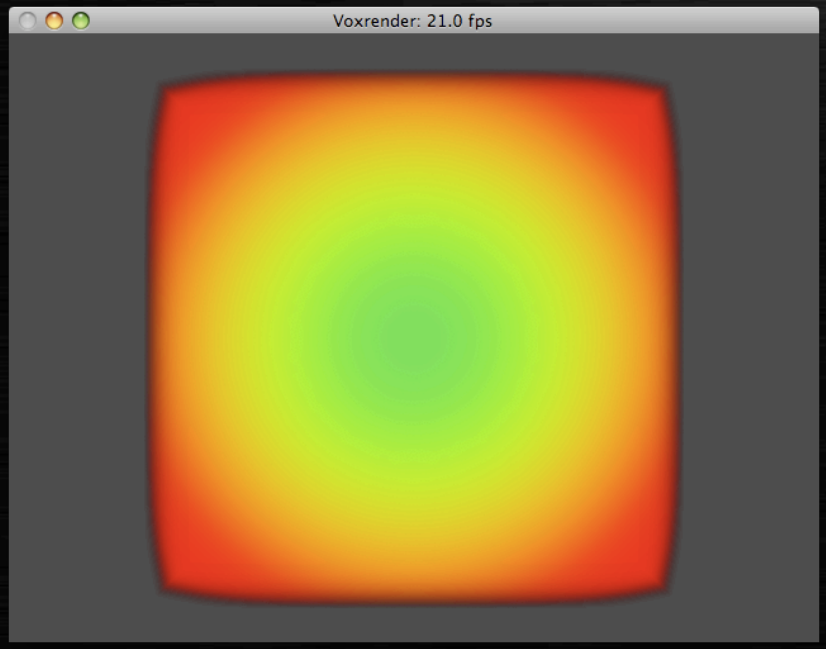
- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

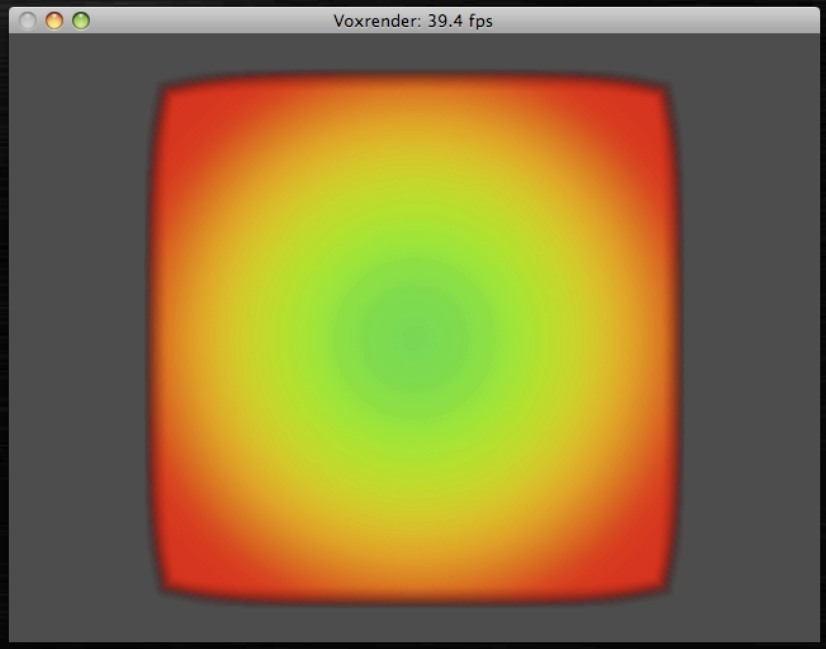
Features and Applications

- Performance Optimizaitons
  - Early ray termination
- Quality Enhancements
  - Supersampling
  - Texture filtering
- Applications
  - Mulitple volume rendering
  - Hypertextures
  - Anaglyphs

Early Ray Termination

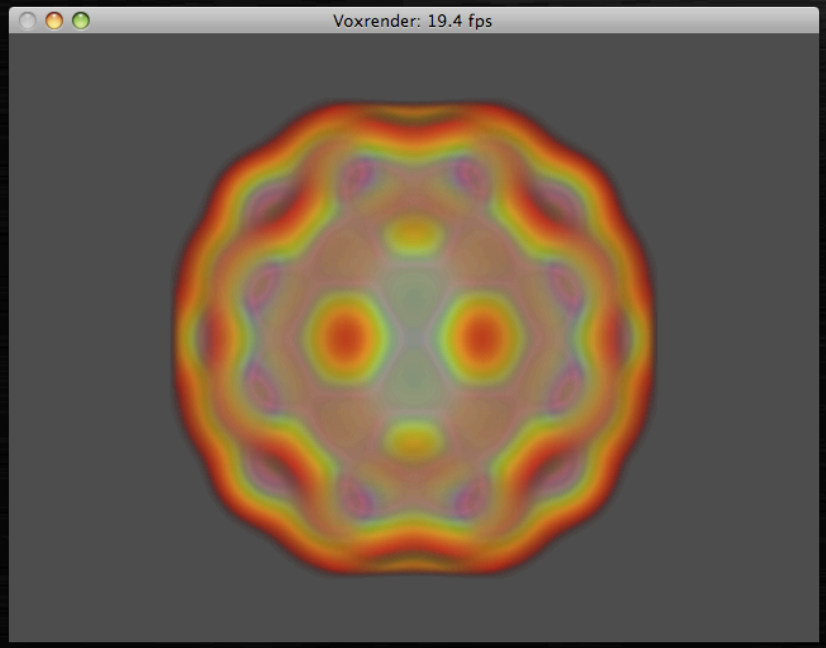
- Terminate the ray marching algorithm once a given threshold has been reached
- Requires marching in front to back order
- Performance dependent on volume
- Capable of 2x and greater speedup

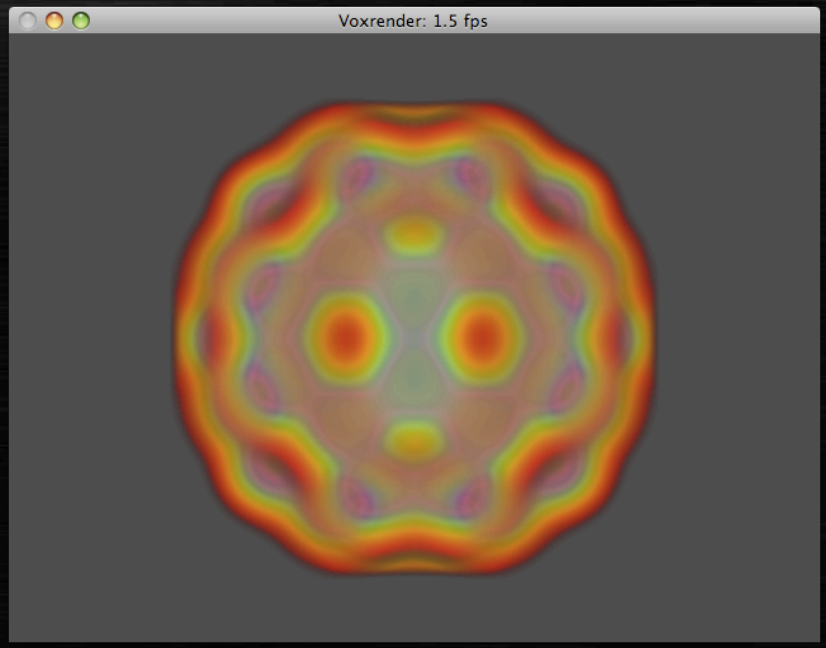




Super Sampling

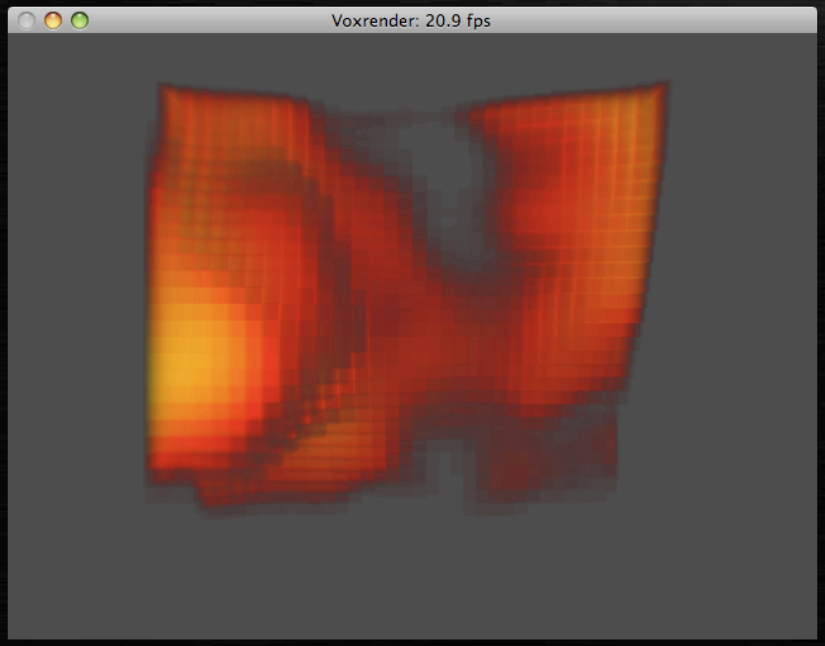
- Intended to reduce aliasing (aka: “jaggies”) along edges of sharp contrast
- Cast multiple rays per pixel and average
- Experimentation yielded underwhelming results, at high performance cost

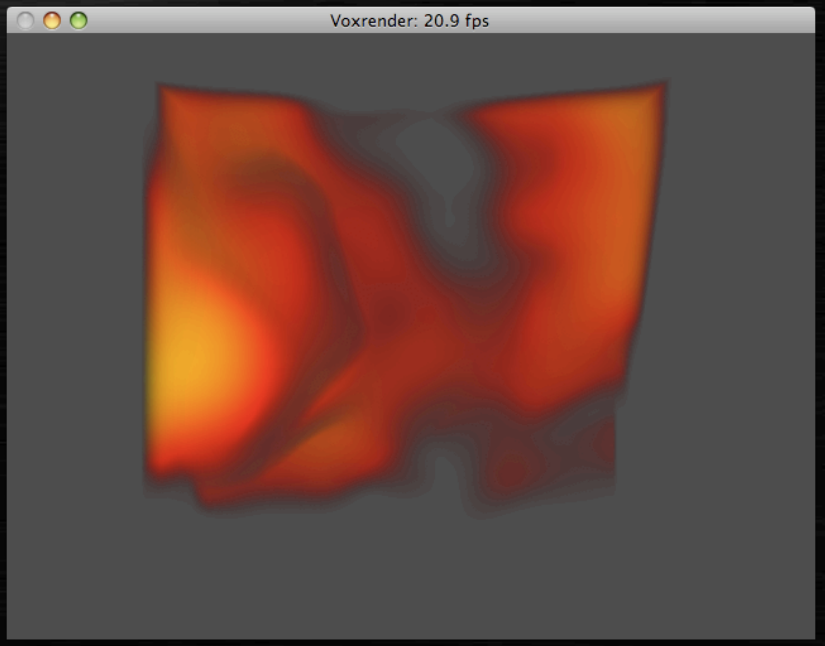




Texture Filtering

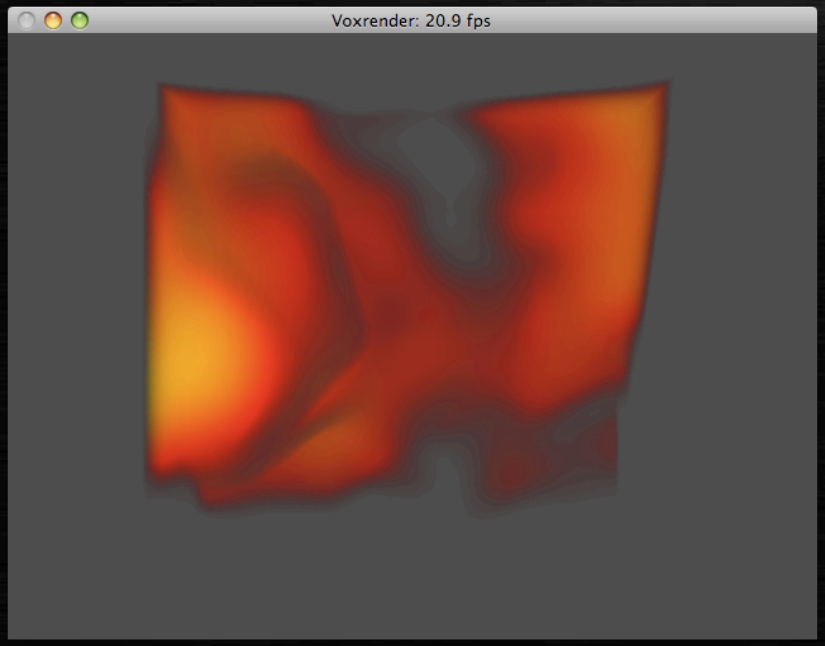
- Two modes for reading from a texture:
  - Point Mode: sample nearest texel to coordinate
  - Linear Mode: sample linear average of surrounding texels to coordinate
- No discernible difference in render time

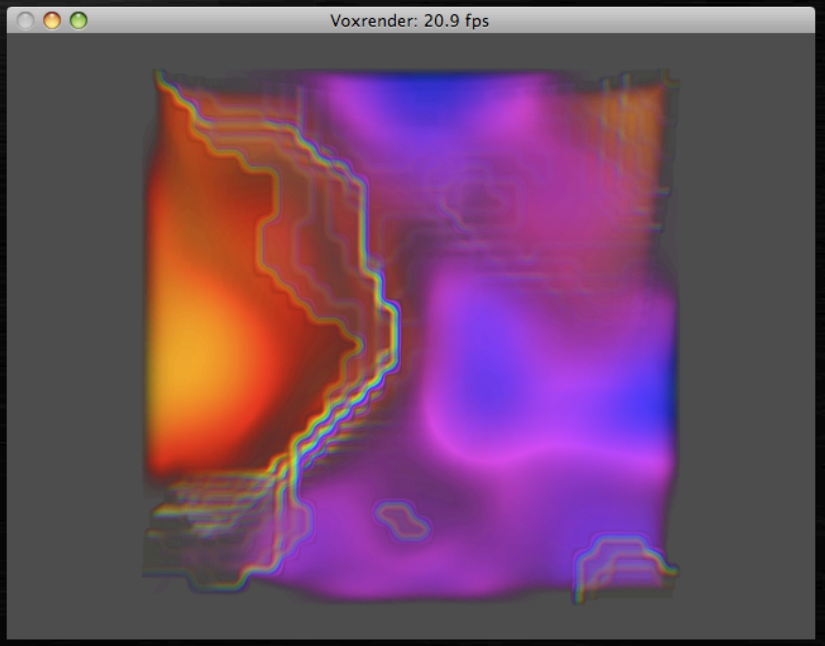




Hypertextures

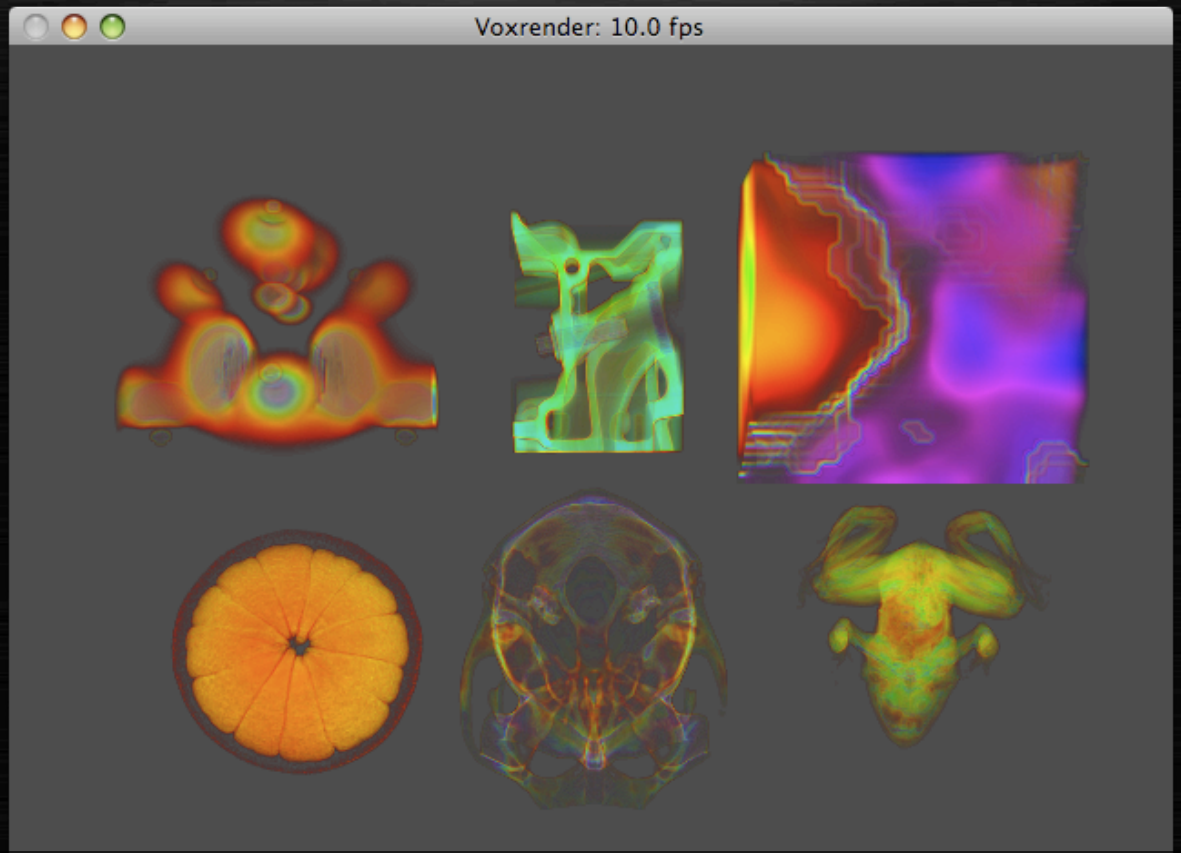
- Invented by Ken Perlin, using Perlin noise
- A technique capable of representing objects difficult to define by triangular meshes, such as fire, smoke and ice



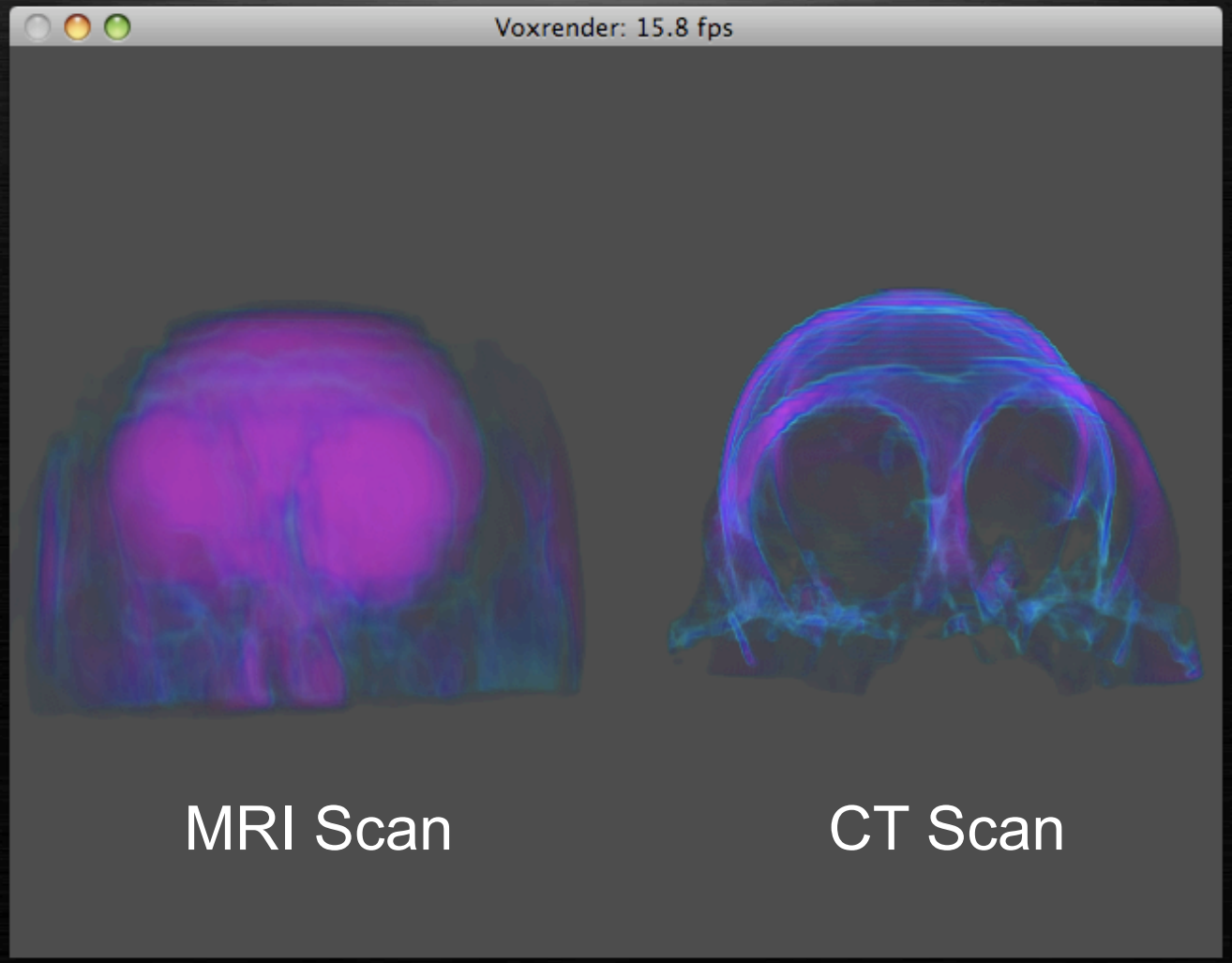


Multiple Volume Rendering

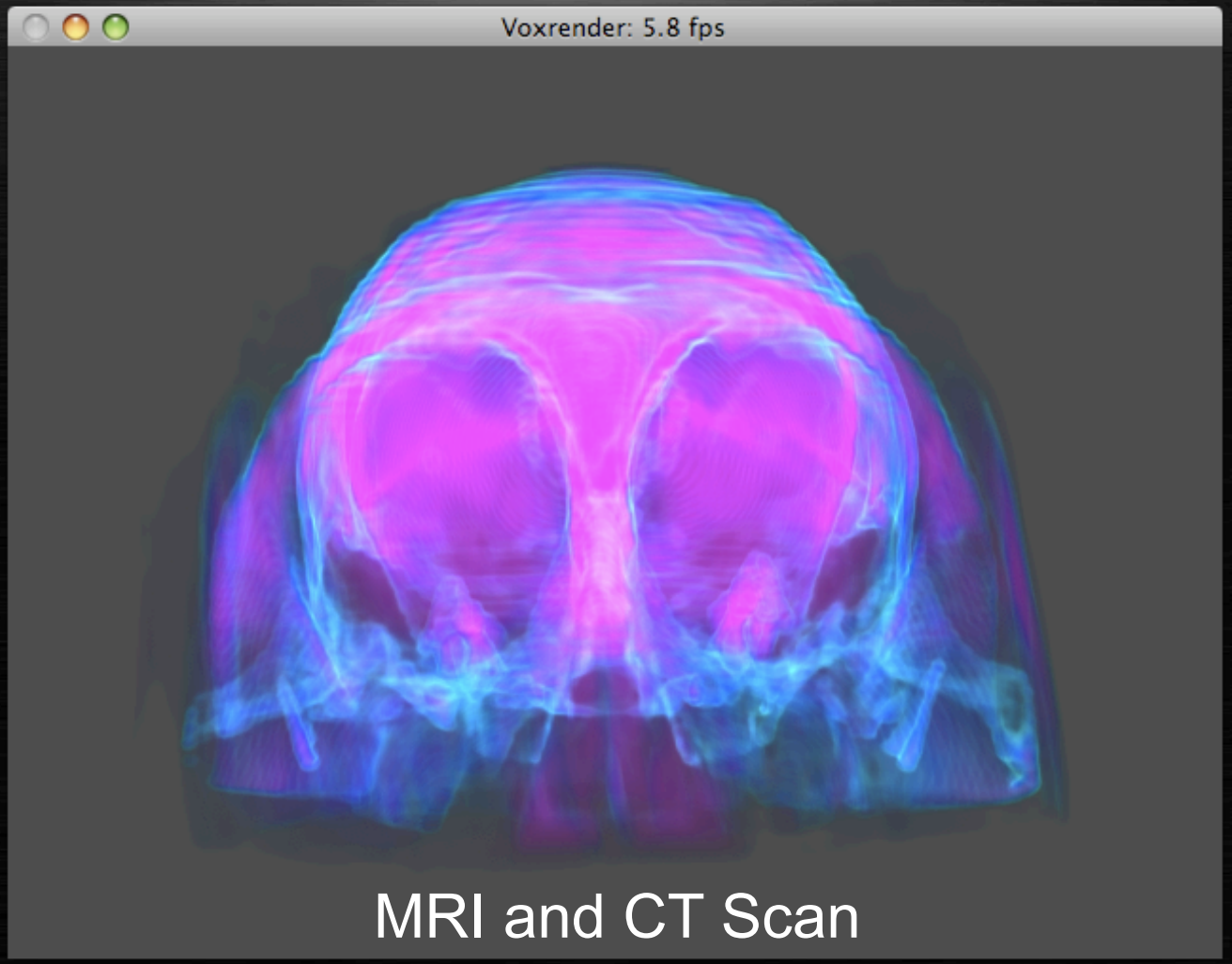
- Allows for arbitrary placement of volumes
- Allows simultaneous rendering of volumes of varying resolution and “true” dimensions



Rigid Registration



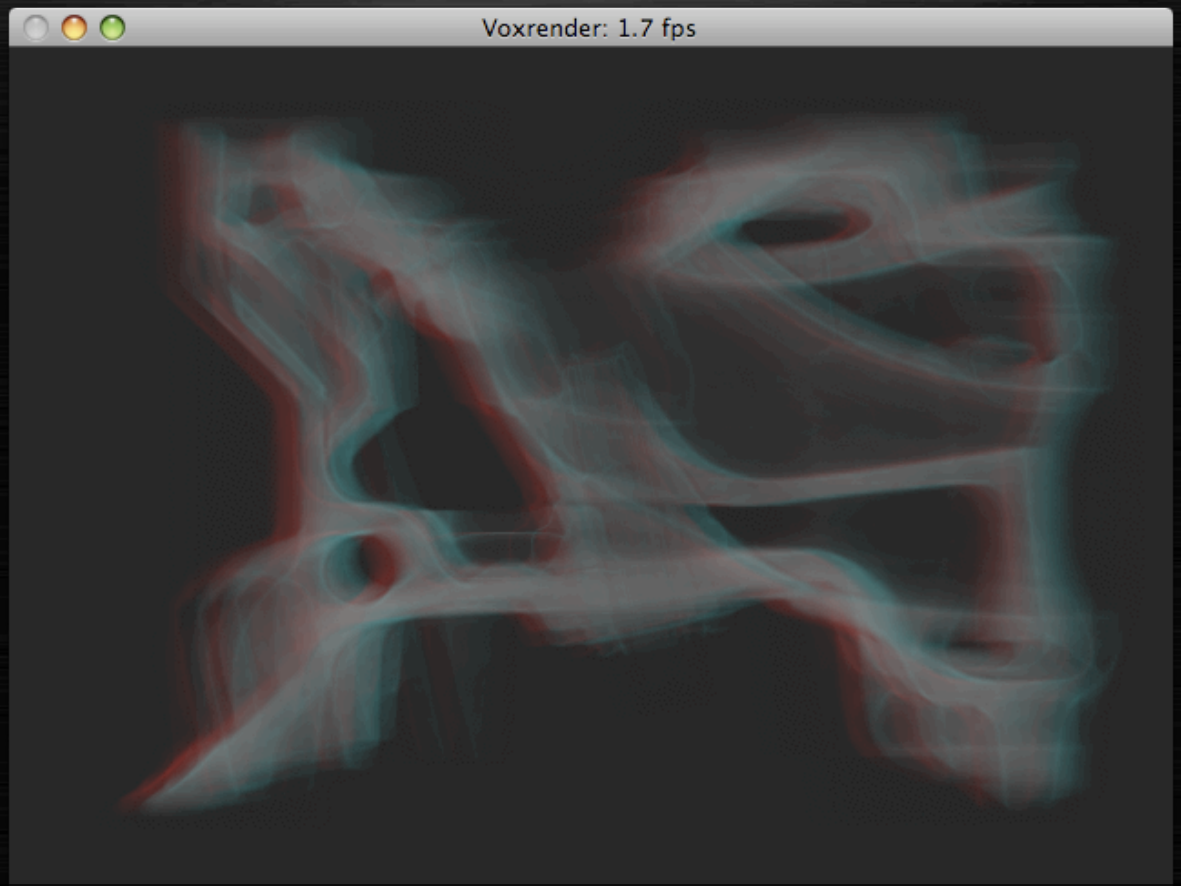
Rigid Registration



MRI and CT Scan

Anaglyphs

- Allows for visualization of stereoscopic images using “3D glasses”
- Each thread fires a ray from two perspectives

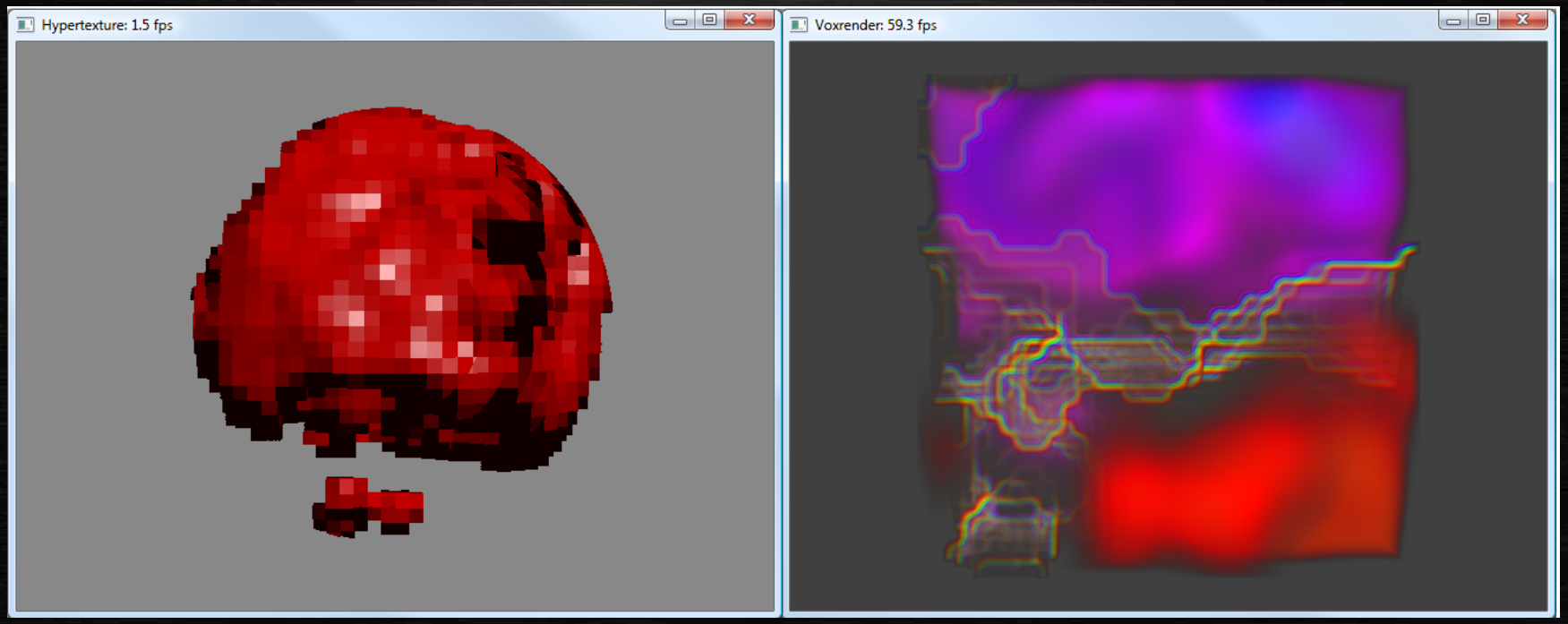


Sequential vs. CUDA

- Primitive volume rendering system developed for sequential execution on CPU
- Sequential system does not perform ray marching (the majority of rendering workload), but does perform phong illumination
- Intel Core 2 Quad Q9550 achieves 1.5 FPS
- Nvidia GeForce GTX260 achieves ~60 FPS
- Speedup of ~40x despite significantly reduced workload of sequential system

Price-Performance Ratio

- Intel Q9550: ~$280 as of May 2009
- Nvidia GTX260: ~$160 as of May 2009
- GPU capable of approximately 70 times the FPS per dollar of CPU with tested hardware



Outline

- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

Future Work

- Performance
  - Reduce register usage
  - Reduce branching
  - Empty Space Skipping
  - Occlusion of volumes
- Image Quality
  - Illumination model
  - Adaptive antialiasing or step size
- Behavior
  - Robust camera functionality

Future Work

- Applications
  - Real time deformable registration
  - Volumetric animation
  - Real time hypertexturing of objects
  - Constructive Solid Geometry for volumes
  - Volume mipmapping
  - Volume compression algorithms
  - New methods of interacting with volumes

Outline

- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration

Conclusion

- Successful parallelization of volume rendering
- Developed extensible and portable framework for interactive volume rendering on CUDA
- 60 FPS on mid-high range hardware
- 40x speedup or more over sequential
- GPU has 70x FPS per dollar of CPU
- Early ray termination big performance gain
- Linear texture filtering for great quality
- Multiple volumes and rigid registration
- Unique volume rendering applications

Outline

- Volume Rendering
- GPGPU Background
- CUDA Architecture
- Volume Rendering on CUDA
- Implemented Features and Analysis
- Future Work
- Conclusion
- Demonstration