GigaVoxels

Ray-Guided Streaming for Efficient and Detailed Voxel Rendering

Presented by:

Jordan Robinson

Daniel Joerimann

Outline

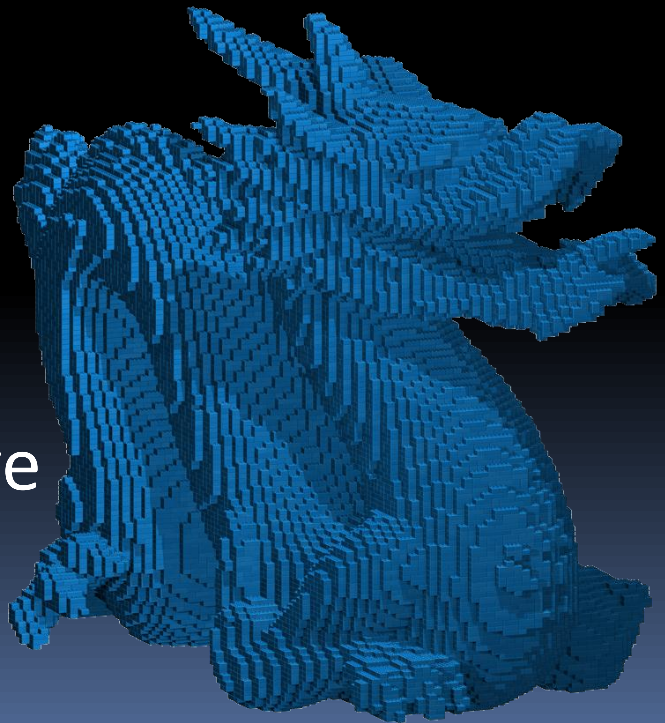
- Motivation
- GPU Architecture / Pipeline
- Previous work
- Support structure / Space partitioning
- Rendering
- Tree updating on the GPU
- Results

Motivation

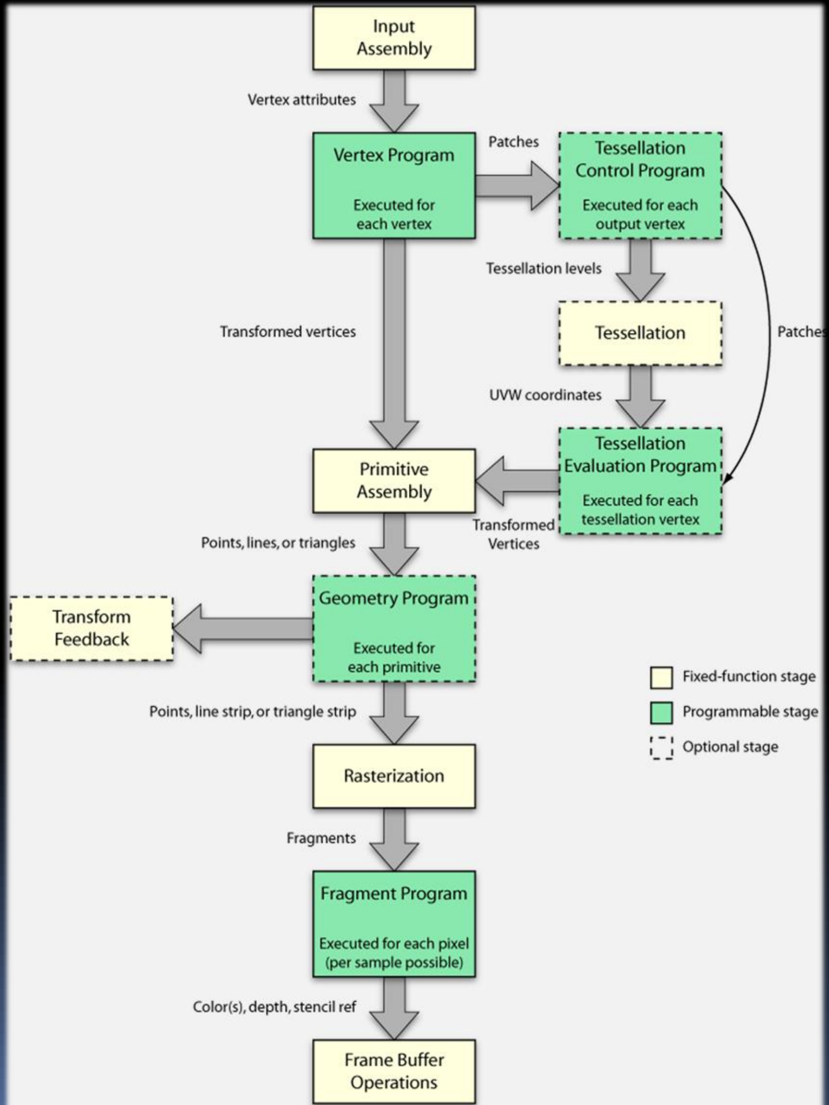
Why Voxels?

- Visualizing scientific data / 3D scans
- Easy to manipulate
- Good for pseudo-surfaces

... but hard to render very large data sets with interactive rates (Real time)



GPU Architecture / Pipeline

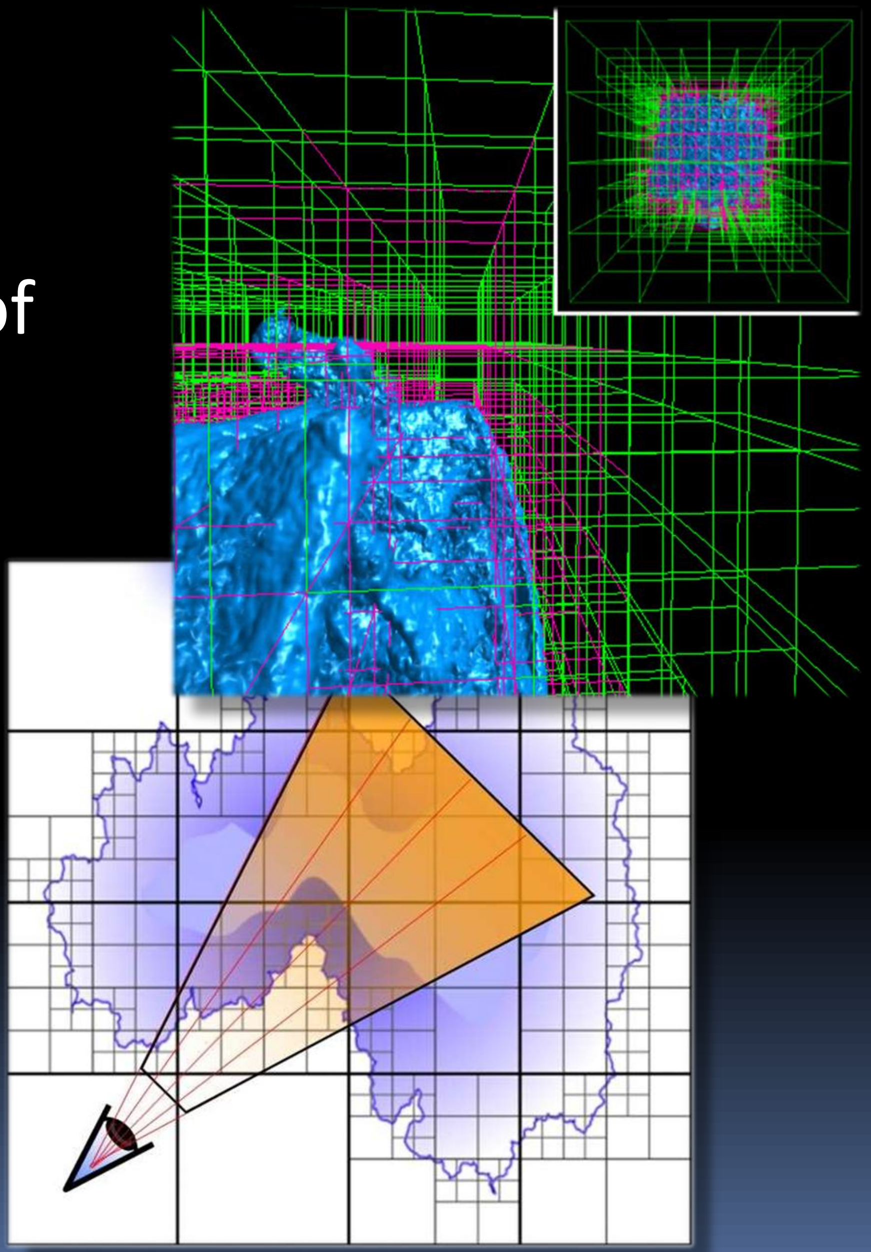


Previous Work

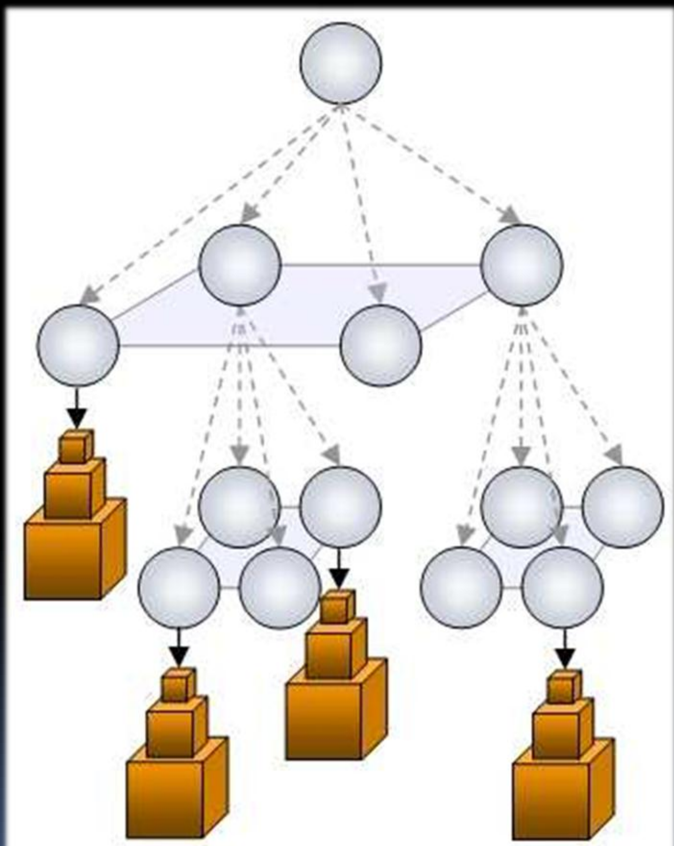
- **GPU Gems 2: Octree Textures on the GPU** by Lefebvre, Hornus, Neyret 2005
- **Rendering Fur With Three Dimensional Textures** by Kajiya and Kay 1989
- **On-the-fly Point Clouds through Histogram Pyramids** by Ziegler, Tevs, Theobalt, Seidel 2006
- **High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading** by Engel, Kraus, Ertl 2001

Space partitioning

- Sparse distribution of voxels
- Voxels have to be organized
- Accelerates Ray Traversal
- Spatial N^3 –Trees
- Typically N = 2
  - Octree

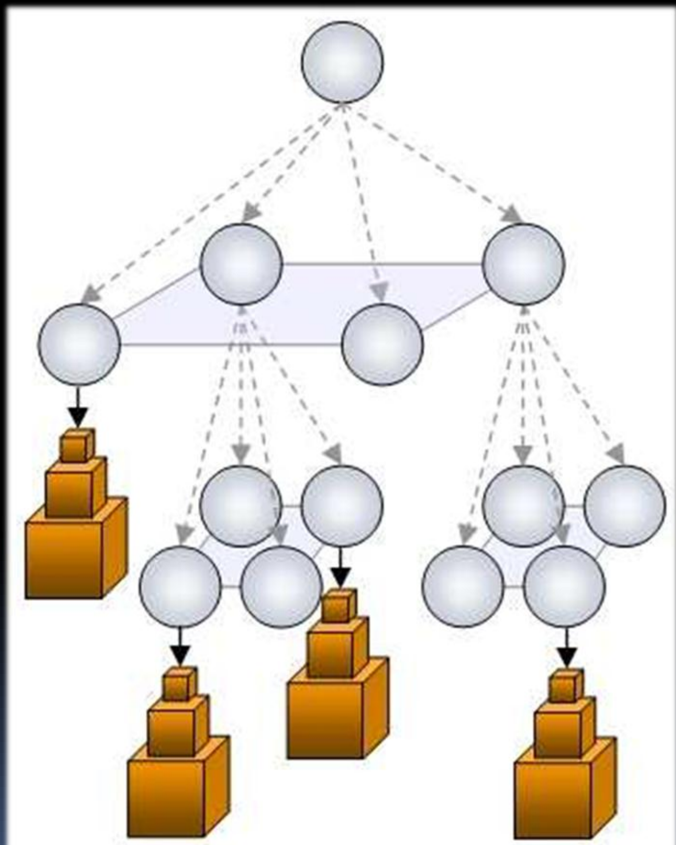


Support structure



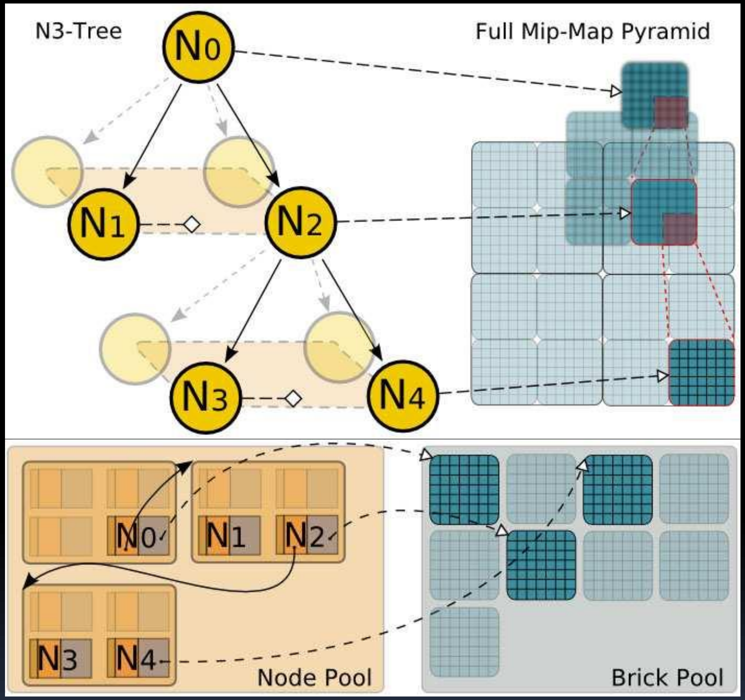
- Split into tree and bricks
- Node:
  - Corresponds to a node in the N^3 tree
- Brick:
  - Contains the Voxel data

Support structure: Brick



- Bricks are stored in a large shared 3D – Texture (Brick pool)
- Voxel-grid of size M^3 (usually M=32)
- 3D-Mip-Mapped

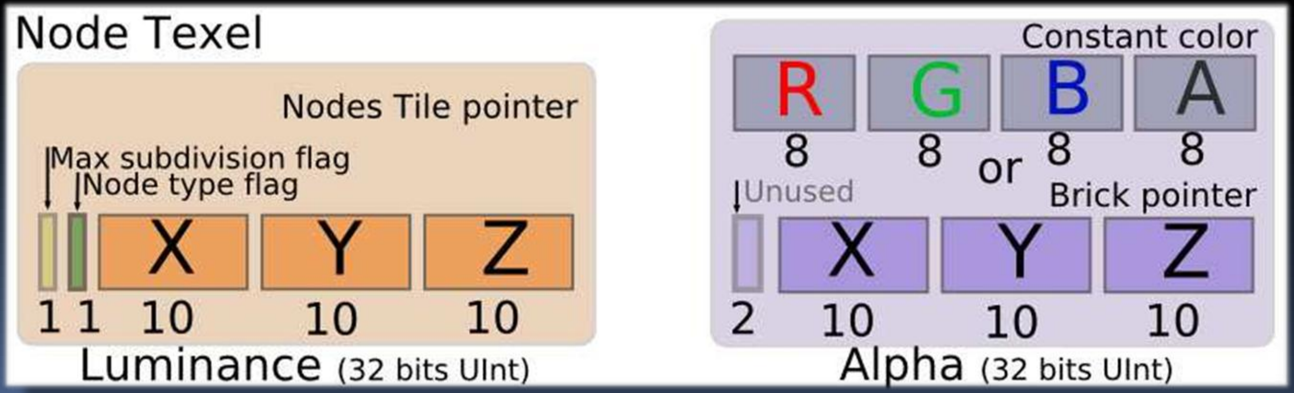
Support structure: Memory layout



- Tree-Nodes and bricks are stored in 3D Textures (Node Pool and Brick Pool)
- Nodes can point to child nodes and a corresponding brick

Support structure: Node Texel

- Contains (64 bits):
  - 3D Pointer (X,Y,Z) to the next level in the tree (N^3 child nodes)
  - Constant Color or Brick Pointer
  - Flag indicating whether it is a leaf node
  - Flag indicating the node type (Constant Color or Brick pointer)

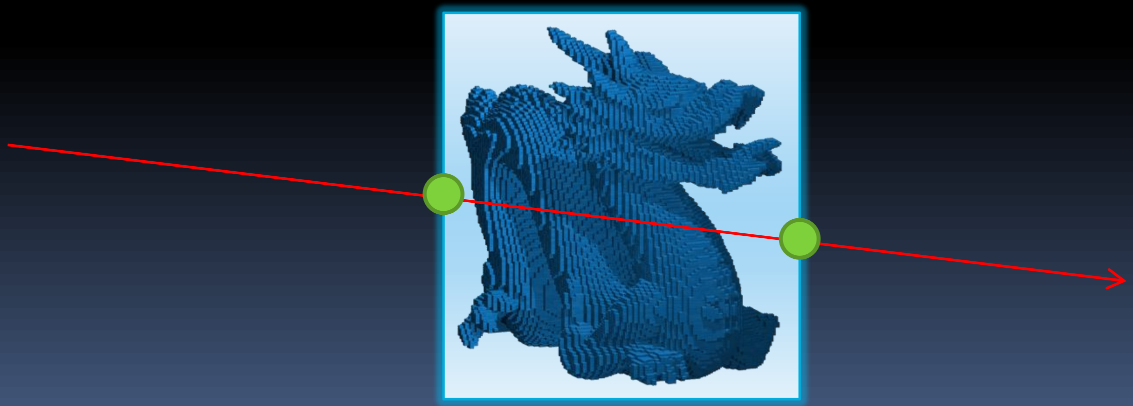


Rendering

1. Rendering of a proxy geometry to generate rays
2. Tracing the rays into the tree
   (Up to the needed LOD)
3. Shade pixel
4. Tree updates

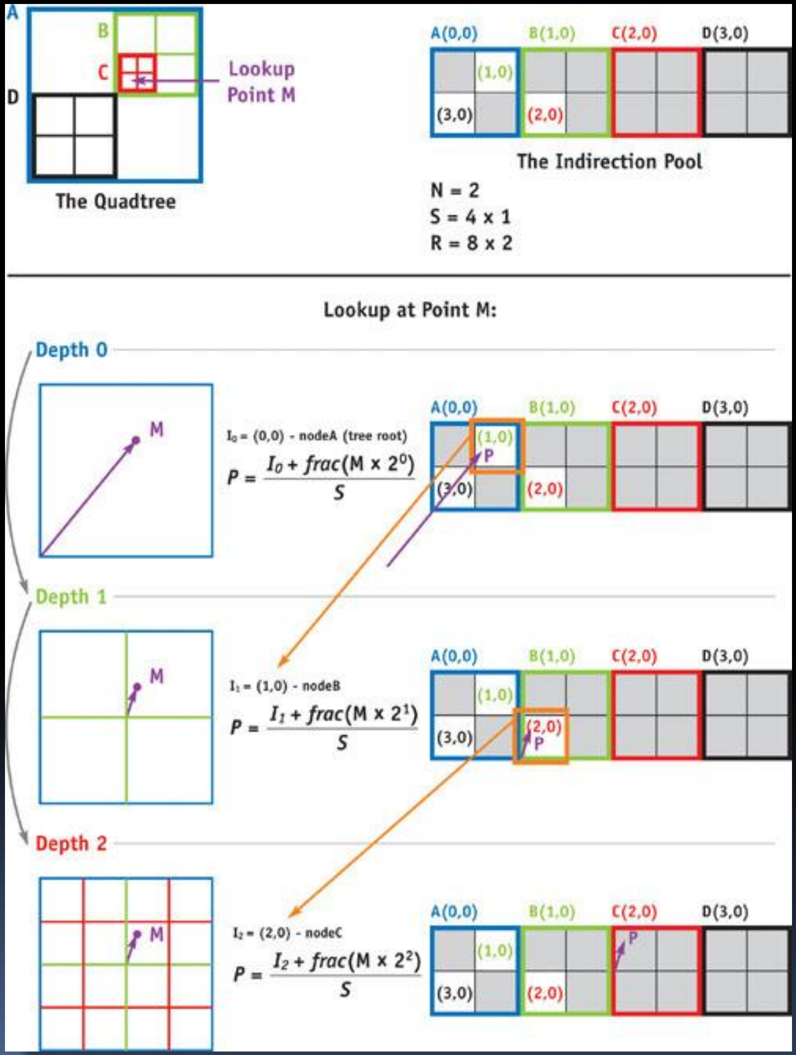
Rendering: Proxy geometry

- Needed to initialize (create) rays
- Either a bounding box or some approximate geometry of the volume
- Render front faces and back faces defining the view rays into a texture



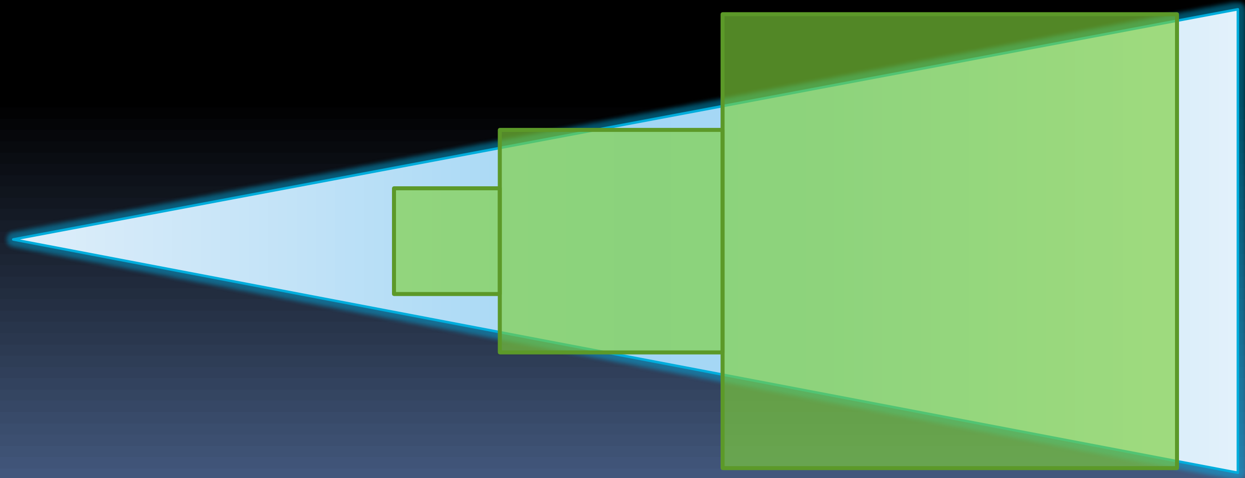
Rendering: Tracing rays

- Render the flat texture (from the step before)
- Walk the tree / bricks for every pixel in the fragment shader
  - DDA could be used but is inefficient on the GPU
  - Iterative descent is faster due to the GPU cache



Rendering: High Quality Filtering

- The filtering quality for the previous ray traversal method could be improved
- 3 MIP-Map levels are used to filter



Pixel shading

- Accumulated color and opacity values
- Phase function
- Pre-integrated transfer function
- Using the density gradient as the normal for pseudo-Phong shading

Tree updates / Memory management

- The entire tree and brick pool are usually too large to fit into the GPU memory
- Interrupting and updating
  - Multiple passes
  - Mark pixels with insufficient data
    1. Interrupt
    2. Load missing data
    3. Continue
  - Early-Z and Z-Cull prevents pixels with terminated rays from being overdrawn

Advanced Algorithm

- Interrupting and updating is too slow: Requires lots of CPU interaction (CPU-GPU bandwidth is limited)
- Try to keep all needed data available in the GPU’s memory
- => Render one frame in one step
- Every node and brick has a Timestamp in the CPU’s memory
- Replaces nodes and bricks by LRU

Advanced Algorithm

CPU:

while (true)

- Render image (using the GPU)
- Get list of accessed/needed nodes from the GPU
- Reset timestamp of accessed nodes
- Expand or collapses nodes
- Update GPU memory with needed nodes (LRU)

GPU: Fragment shader

First pass:

- Trace ray
- if LOD not available
  - Pick next higher available level in Mip-map
- Shade pixel
- Keep a list of accessed nodes / Mip-map levels in result textures

Second pass:

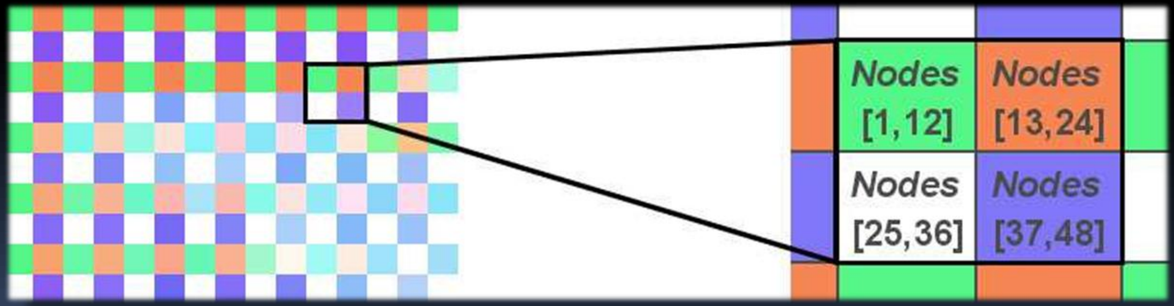
- Compress accessed/needed data

Advanced Algorithm

- Node list is stored in multiple render targets (MRTs)
- RGBA32 = 4 x 32 bit
- One node pointer uses 32 bits
- One channel per node pointer
- Can store up to 12 node id’s per pixel using 3 MRTs

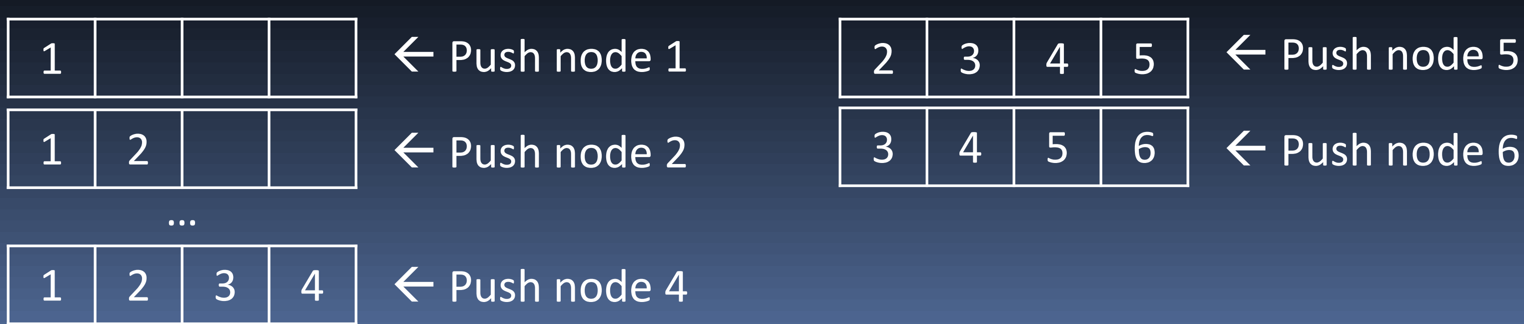
Advanced Algorithm: Compression

- Spatial node coherence
  - Normally 3 MRTs would not be enough
  - Neighboring rays traverse similar nodes
  - Group in 2x2 grid



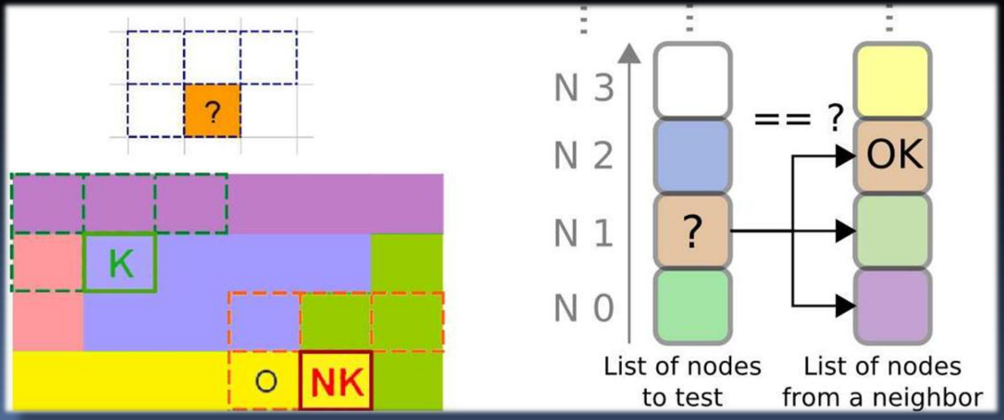
Advanced Algorithm: Compression

- Temporal coherence:
  - Used nodes are similar between subsequent frames
  - FIFO (48 items)
    - 48-element window is shifted after each subsequent frame
    - First frame: push up to 48 nodes into the FIFO
    - Second frame: push up to 96 nodes into the FIFO

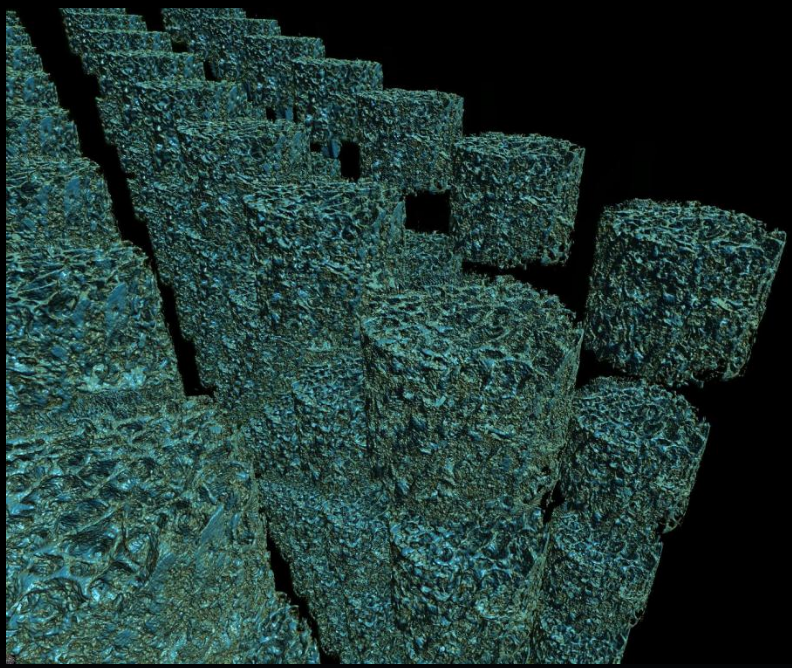


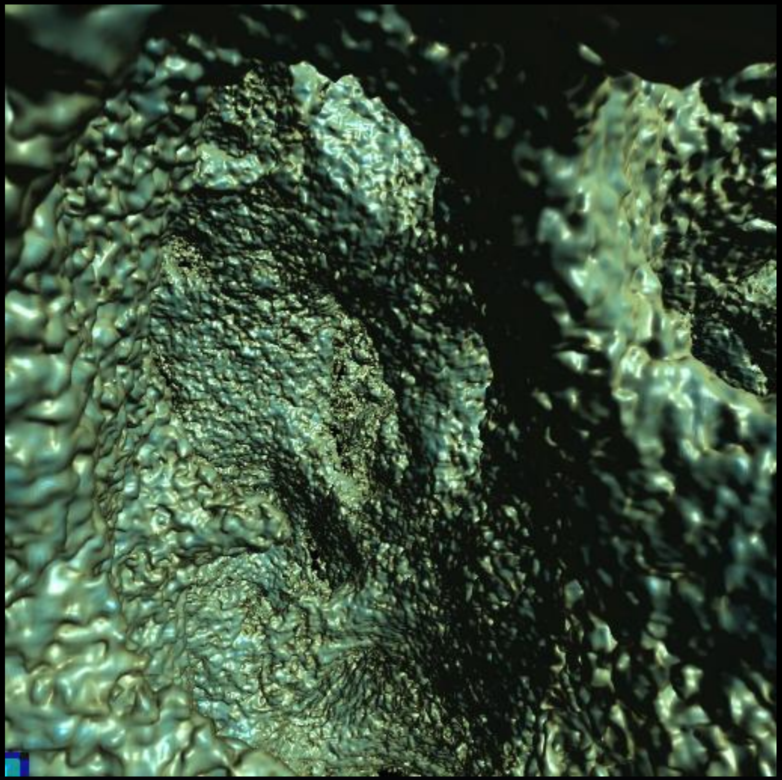
Advanced Algorithm: Compression

- Compaction of update information
  - Preprocess update information before compaction
  - Use mask to remove redundant node selections
  - Compaction step by using Histogram pyramids covered in:
http://www.mpi-inf.mpg.de/~gziegler/gpu_pointlist/paper17_gpu_pointclouds.pdf
  - Final step
    - Fit as much as possible in one RGBA32 texture (4 Nodes per pixel)
    - Postpone to next frame if the limit is exceeded
    - Usually 2-3 nodes per pixel are selected



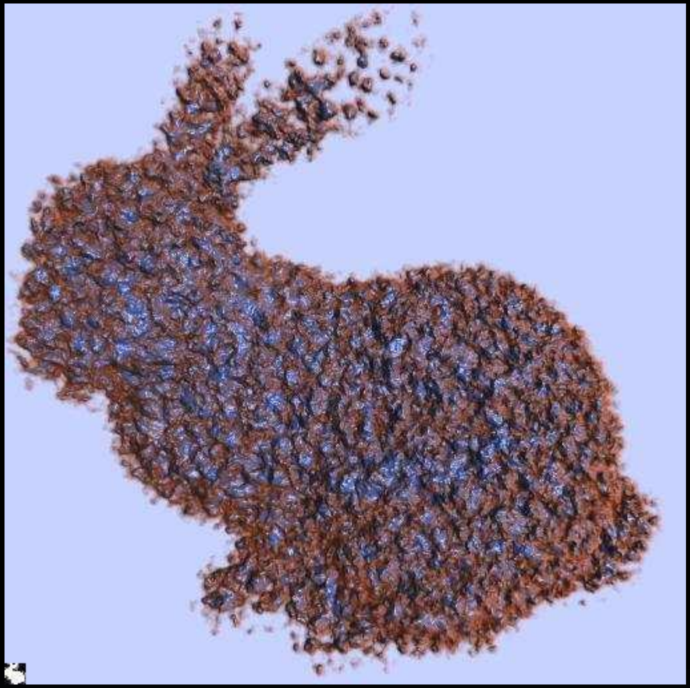
Results

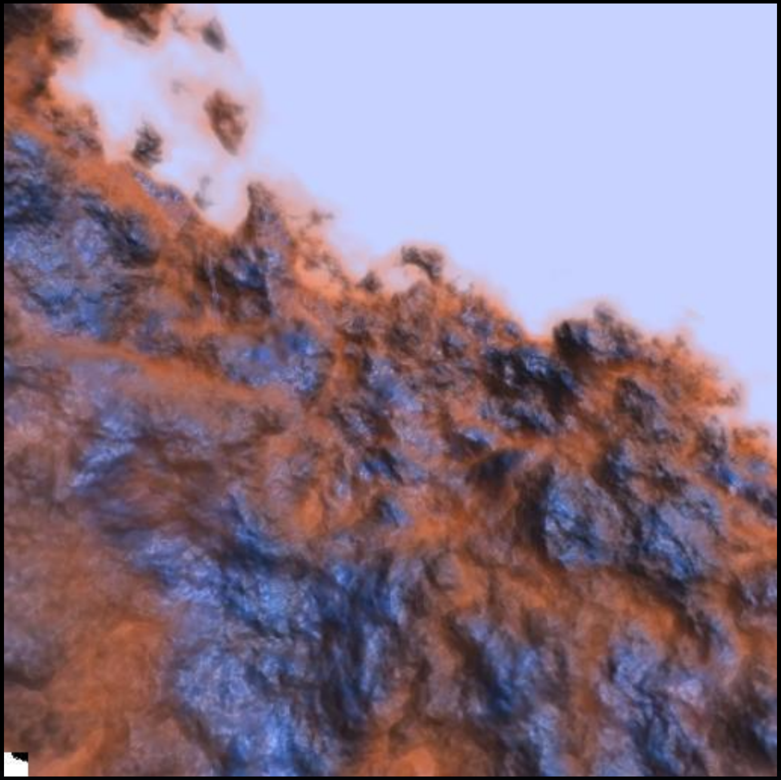




- *Explicit volume (trabecular bone)*
  - 8192^3 Voxels
  - 20 – 40 Fps (Mip-mapping enabled)
  - 60 Fps (Mip-mapping disabled)
  - System: Core2 bi-core E6600 at 2.4 GHz & NVIDIA 8800 GTS 512MB

Results





- *Hypertextured bunny*
  - 1024^3 Voxels
  - 20fps
  - System: Core2 bi-core E6600 at 2.4 GHz & NVIDIA 8800 GTS 512MB

Video

Questions?