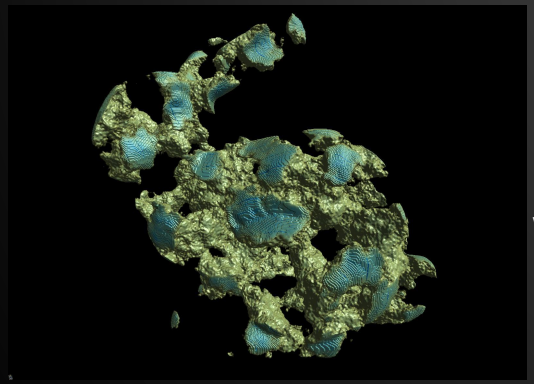
GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, Elmar Eisemann

summarized by Sarah Kushner

Volume Rendering and Voxels

Used to represent pseudo-surfaces like foliage, clouds, smoke and even extremely detailed geometry



These two techniques together are used in conjunction to deal with the level-of-detail problem.

Proposal

Current hardware is ready to render massive amounts (several billion voxels) of volume data in real-time or at an interactive rate.

Problems

1. Memory limitations
    a. The transfer of 512MB of data onto the GPU prevents real-time performance.
2. Costly rendering
    a. Costly because of large amounts of voxels.

Approach

-The norm for concentration of scene details in CG is usually an interface between dense clusters of detail and empty space.

-Focuses on out-of-core voxel rendering.

Optimizations

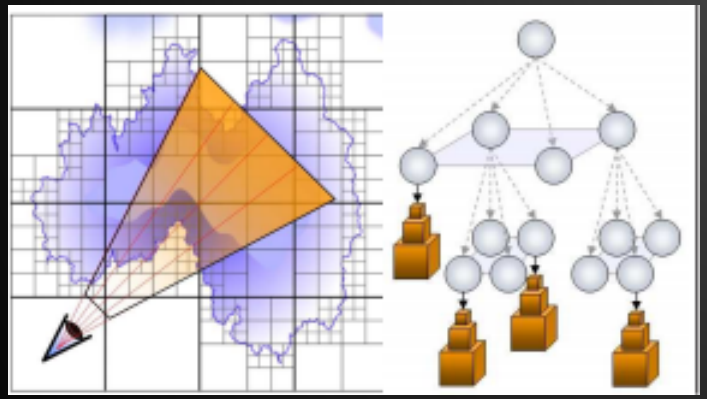
-Depending on the camera, not everything needs to be in memory all the time.

-Distant objects can have lower mipmap levels, less detail, and in turn take less memory.

Data Structures

-N^3 trees and mipmapped texture tiles.

-Each node is single pointer to a "brick" or an empty space.



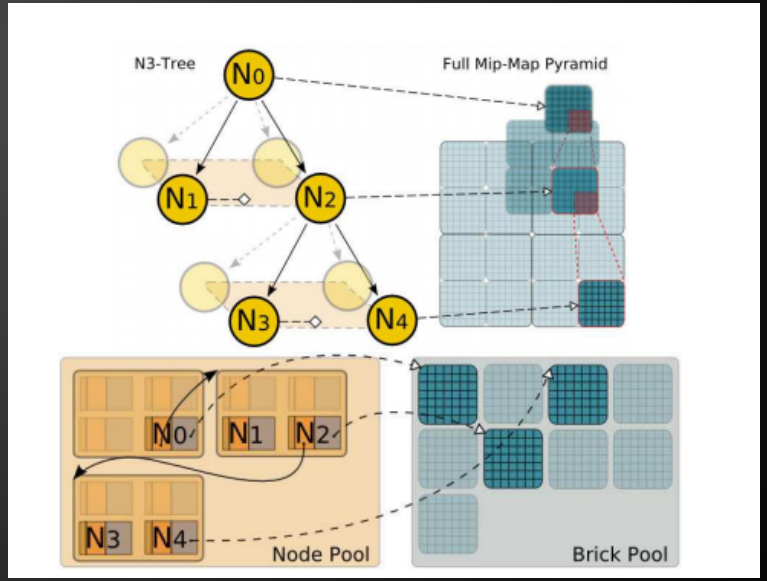
-Brick: small voxel grid of size M^3 (usually M = 32) that approximates part of the original volume.
-All bricks stored in brick pool.
-Updates only triggered if data in the brick pool is missing.
-New bricks stored in oldest locations according to timestamp. (Last Recently Used)

Organization

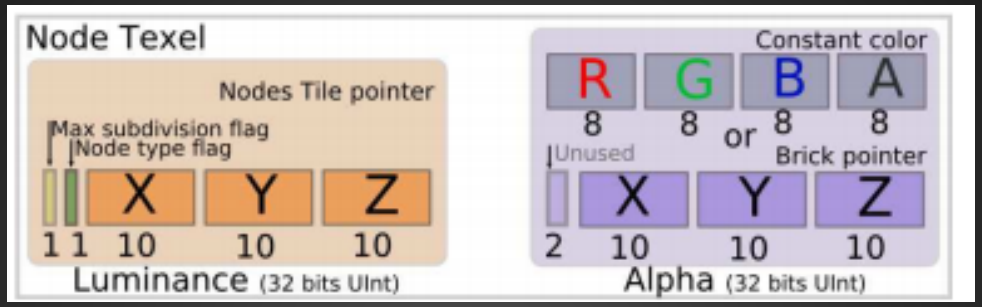
-All nodes in 3D texture called "node pool".

-Grouping gains access to all children through 1 pointer.

-16x memory improvement since nodes are now 64 partitioned accordingly.



Nodes



-30 bits - encode a pointer to the child nodes

-1 bit - whether the node is refined to a maximum or contains more data

-1 bit - whether the content is a constant RGBA8 value or described by a brick

-30/32 bits - either a pointer to an M^3 brick or the average value at this location

Rendering

-March the data in the structure along the view rays while accumulating color and opacity.

-Iterative descent from the tree root like the kd-restart algorithm.

-Level-of-detail determined in shader.

-Ray-casting in one big fragment shader.

-Adapt sampling rate & mipmap level during the ray marching depending on viewpoint.

Communication with CPU

-Brick pool on CPU.

-Texture-update calls -> modifications go to GPU (Node pool).

-Unified management of the brick pool and the node pool as two Last Recently Used controlled caches.

Node List

-During rendering, traverse the node tree. Stop on node corresponding to the needed level-of-detail.

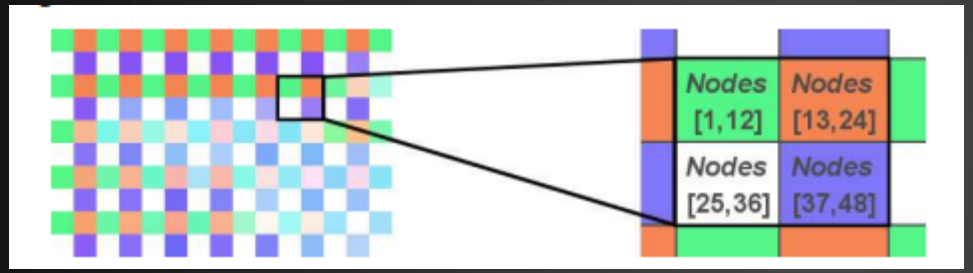
-If the required data is present, we traverse the mipmapped brick.

-For CPU to keep correct nodes in pool, collect the current node index in a node-list.

-If level-of-detail is missing or the node is terminal, it is added to the node-list.

Neighbor Rays

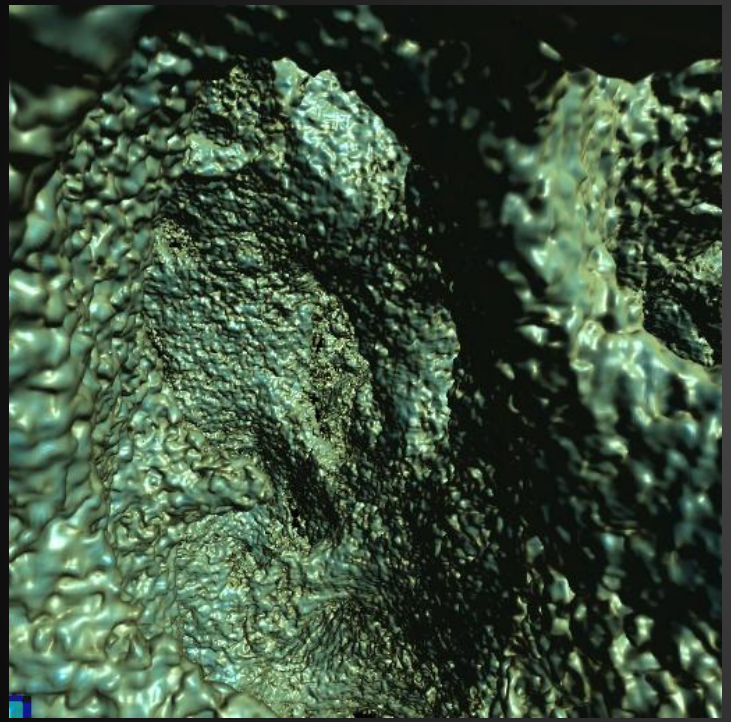
-Group rays in packages in the image plane.



-Sort to ensure only one instance of each node reference.
-Sorting can also be costly.

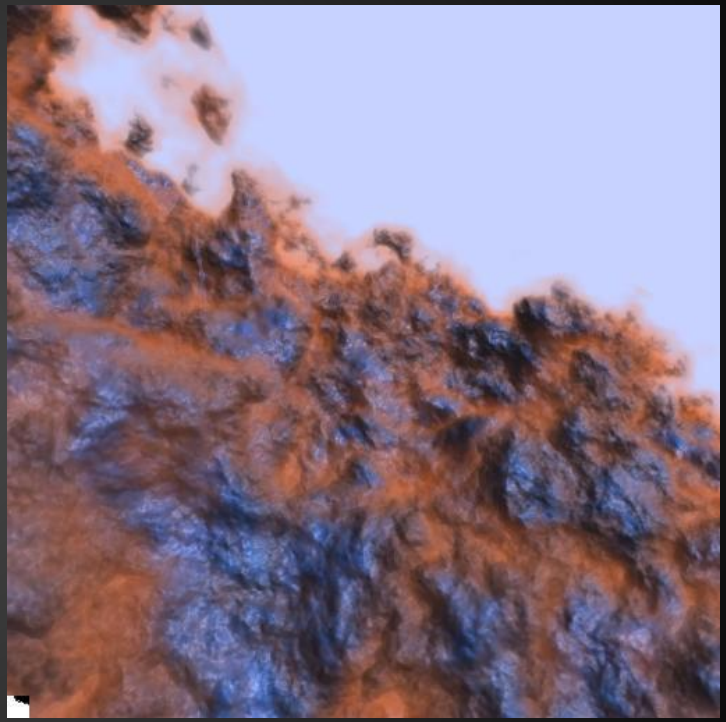
-Instead of brute-force sorting, rays are only compared with neighboring rays.

Results



-1024^3 scanned volume of bone
-N = 2 and M = 16
-60 fps

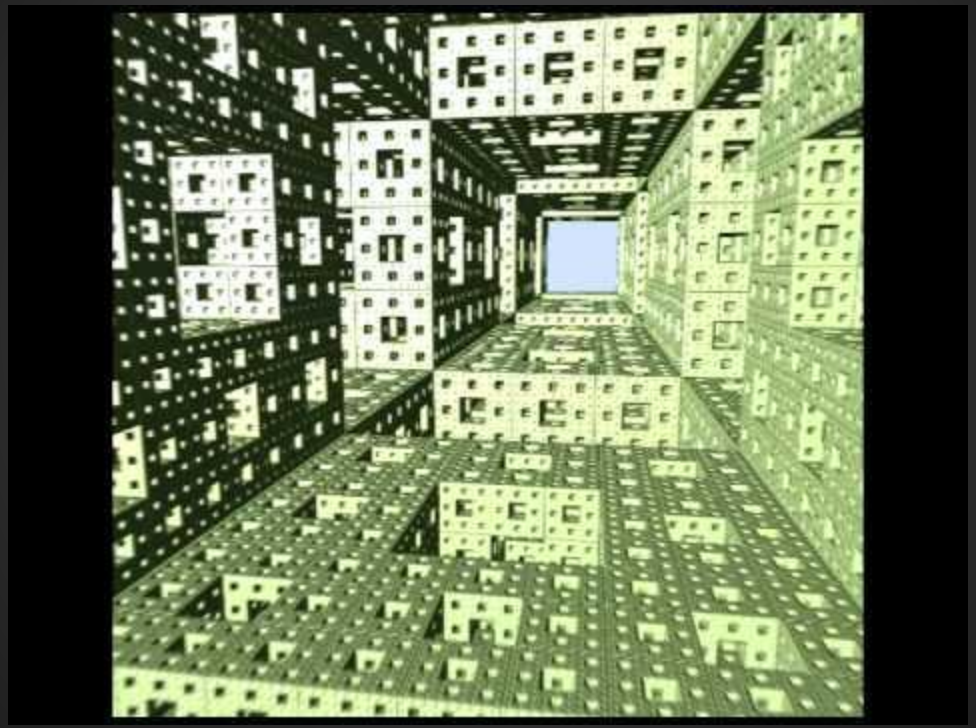
- 20 octaves of Perlin noise, shading, materials
-20 fps for a 1024^3 volume



Conclusion

Through compact data structures and limited, clever CPU to GPU communication, interactive and real-time rendering speeds are possible for extremely large volume data.

Demo



https://www.youtube.com/watch?v=HScYuRhgEJw