What is GigaSpace / GigaVoxels ?

Pascal Guehl , Fabrice Neyret

TECHNICAL REPORT

N° 9999

23/12/2012

Project-Team Maverick

ISSN 0249-6399

What is GigaSpace / GigaVoxels ?

Pascal Guehl[1] , Fabrice Neyret [2]
Project-Teams Maverick

Technical Report N° 9999 — 23/12/2012 —99 pages.

Abstract: This the programming guide of the GigaVoxels library.

Key-words: insérez ici les mots-clés en anglais

[1] Pascal Guehl affiliation – pascal.guehl@inria.fr

[2] Fabrice Neyret affiliation – fabrice.neyret@inria.fr

RESEARCH CENTRE
GRENOBLE - RHÔNE-ALPES

Inovallée
655 avenue de l'Europe - Montbonnot
38334 Saint Ismier Cedex France

What is GigaSpace / GigaVoxels ?

Résumé : This is the GigaVoxels library programming guide for developers. It is an deep insight for make benefit glorious nation of voxels worshipers.

Mots clés : insérez ici les mots-clés en français

Contenu

I. Introduction

…

1. About this document

This document explains what is GigaVoxels, its philosophy, its origin and how it extends to GigaSpace.

II. GigaVoxels

Gigavoxels is an open library made for GPU-based real-time quality rendering of very detailed and wide objects and scenes (not necessarily fuzzy or transparent). It can easily be mixed with ordinary OpenGL objects and scenes. Its secret lies in lazy evaluation: chunks of data are loaded or generated only once proven necessary for the image and only at necessary resolution. Then, they are kept in a LRU cache on GPU for next frames. Thus, hidden data have no impact at all for management, storage and rendering. Similarly, high details have no impact on cost as well as aliasing if smaller than pixel size. Gigavoxels allows to do easily the simple things, and also permits a progressive exposure of the engine depending of the customization you need. It is provided with plenty of examples to illustrate possibilities and variants, and to ease your tests and prototyping by copy-pasting.

In the most basic usage, you are already happy with what does one of the examples (good for you: nothing to code !)

The most basic programming usage consists in writing the callback producer function that GigaVoxels will call at the last moment to obtain data for a given cube of space at a given resolution. This function has one part on CPU side and one part on GPU side. You might simply want to transfer data from CPU to GPU, or possibly to unpack or amplify it on GPU side. You might also generates it procedurally, or convert the requested chunk of data* from another shape structure already present in GPU memory (e.g., a VBO). footnote*: The basic use of Gigavoxels considers voxel grids since it is especially efficient to render complex data on a screen-oriented strategy: data are visited directly and in depth order without having to consider or sort any other data, while OpenGL - which is scene-oriented - has to visit and screen-project all existing elements, an especially inefficient strategy for highly complex scenes. LOD and antialiasing is just a piece of cake using voxels, and soft-shadows as well as depth-of-field are easy to obtain - and indeed, faster than sharp images. So, your producer might be a local voxelizer of a large VBO - BTW, we provide one.

You probably also want to add a user-defined voxel shader to tweak the data on the fly or simply to tune the appearance. In particular, it is up to you to chose what should be generated on GPU, in the producer or in the voxel shader, depending on your preferred balance between storage, performances and needs of immediate change. E.g., if you want to let a user play with a color LUT or tune hypertexture parameters, this should be done in the voxel shader for immediate feedback. But if these are fixed, rendering will be faster if data is all-cooked in the producer and kept ready-to-use in cache.

You might want to treat non-RGB values, or not on a simple way. Gigavoxels let you easily define the typing of voxel data. You can also customize pre and post-rendering tasks, various early tests, hints, and flag for preferred behaviors (e.g., priority strategy in strict-realtime).

The basic Gigavoxels relies on octree dynamic partitionning of space, which nodes corresponds to voxel bricks - if space is not empty there. But indeed, you can prefer another space partitionning scheme, such as N^3-tree, k-d tree, or even BSP-tree. You can also prefer to store something else than voxels in the local data chunck, like e.g., a piece of mesh.

III. GigaSpace

Gigaspace: May be you got it; Gigavoxels and Gigaspace are one and same thing. Gigaspace is just the generalized way of seeing the tool. So, let's now present it the general way: Gigaspace is an open GPU-based library for the efficient data management of huge data. It consists of a set of 4 components, all customizable:

A multi-scale space-partitioning dynamic tree structure,

A cache-manager storing constant-size data chunks corresponding to non-empty nodes of the space partition.

A visitor function marching the data.

A data producer called when missing data are encountered by the visitor. Datatypes can be anything, 'space' can represents any variety. The basic use of Gigavoxels use octrees as tree, voxels bricks as data chunk, and volume cone-tracing* as visitor. But we provide many other examples showing other choices. footnote*: volume cone-tracing is basically volume ray-marching with 3D MIPmapping.

IV. CUDA vs OpenGL vs GLSL

The Gigavoxels/Gigaspace data management is a CUDA library. The renderer is provided in both CUDA and GLSL. So, as long as you don't need to produce new bricks, you can render gigavoxels data totally in OpenGL without swapping to Cuda if you wish, for instance as pixel shaders binded to a mesh. Using the Cuda renderer - embedding the dynamic data management -, Gigavoxels allows various modalities of interoperability with OpenGL:

Zbuffer integration: You can do a first pass with OpenGL then call Gigavoxels providing the OpenGL color and depth buffer so that fragments are correctly integrated. Similarly, Gigavoxels can return its color and depth buffers to OpenGL, and so one with possibly other loops. Note that hidden voxels won't render at all: interoperability keeps the gold rule "pay only for what you really see".

It is easy to tell Gigavoxels to render only behind or in front of a depth-buffer, which makes interactive volume clipping trivial.

Volume objects instances and volume material inside meshes: You can use an openGL (bounding-)box or a proxy-geometry, use their objectview matrix to rotate the volume view-angle accordingly, provide only the visible fragments to Gigavoxels, and tell it to render only behind the front depth and (optionally) in front of the rear depth. This allows for the OpenGL display of transformed instances of a given volume (e.g., to render a forest) as well as shapes seeming carved into openwork 3D material.

auxiliary Volumes for improved rendering effects; ray-tracing as second bounce: Gigavoxels is basically a ray-tracer so it can easily deal with reflexion, refraction, and fancy cameras such as fish-eyes. Moreover, it is especially good at blurry rendering and soft-shadows. Why not using it only to improve some bits of your classical OpenGL scene-graph rendering ? Binding Gigavoxels to a surface shader, you can easily launch the reflected, refracted or shadow rays within a voxelized version of the scene.

V. History and roots of GigaVoxels

Gigavoxels draws on:

The idea that voxel ray-tracing is a very efficient and compact way to manage highly detailed surfaces (not necessarily fuzzy or transparent), as early shown in real-time by the game industry reference John Carmack (Id Software) [http://www.pcper.com/reviews/Graphics-Cards/John-Carmack-id-Tech-6-Ray-Tracing-Consoles-Physics-and-more] with Sparse Voxel Octrees [https://www.google.fr/search?q=sparse+voxel+octree&tbm=isch] (SVO), and for high quality rendering by Jim Kajiya [http://www.icg.tugraz.at/courses/lv710.087/kajiyahair.pdf] with Volumetric Textures [https://www.google.fr/search?q=volumetric+textures&tbm=isch] , first dedicated to furry Teddy bears, then generalized by Fabrice Neyret http://hal.inria.fr/index.php?action_todo=search&submit=1&s_type=advanced&submit=1&p_0=contained&v_0=volumetric%20textures&f_0=TITLE&c_0=&l_0=or&p_1=contained&v_1=texels&f_1=TITLE&c_1=&l_1=and&p_2=contained&v_2=&f_2=TITLE&c_2=&l_2=or&p_3=contained&v_3=&f_3=TITLE&c_3=&search_without_file=YES&search_in_typdoc[0]=ART_ACL&search_in_typdoc[1]=ART_SCL&search_in_typdoc[2]=COMM_ACT&search_in_t ypdoc[3]=COMM_SACT&search_in_typdoc[4]=THESE&orderby=DATEPROD&ascdesc=DESC .

On the idea of differential cone-tracing http://en.wikipedia.org/wiki/Cone_tracing which allows for zero-cost antialiasing on volume data by relying on 3D MIP-mapping.

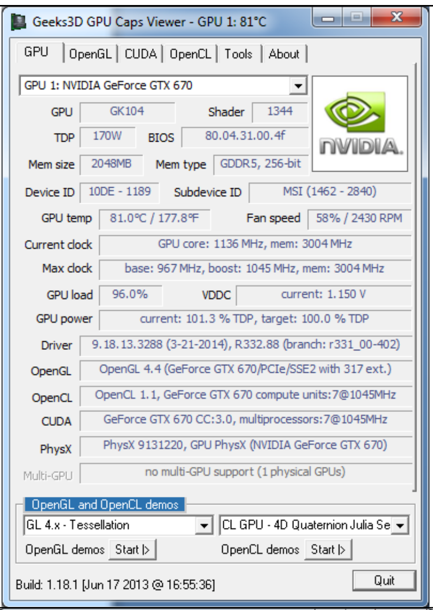
On the technology of smart texture management determining visible parts on the fly then streaming or generating the data at the necessary resolution. Starting with SGI clipmaps and late variants for very large textured terrains in flight simulators, then generalized as a cache of multiscale texture tiles feeded by on demain production, by Sylvain Lefebvre and al. http://hal.inria.fr/inria-00070783

Cyril Crassin connected all these ideas together during his PhD to found the basis of the Gigavoxels system [refs].

VI. Performance Overview

1. System Configuration

Following screenshots shows information about PC, operation systems, memory, GPU, OpenGL features and CUDA.



Geeks3D GPU Caps Viewer - GPU 1: 81°C

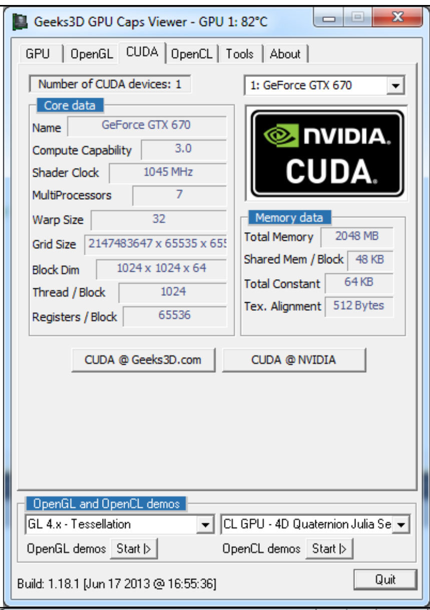
GPU | OpenGL | CUDA | OpenCL | Tools | About

GPU 1: NVIDIA GeForce GTX 670

GPU: GK104 — Shader: 1344
TDP: 170W — BIOS: 80.04.31.00.4f
Mem size: 2048MB — Mem type: GDDR5, 256-bit
Device ID: 10DE - 1189 — Subdevice ID: MSI (1462 - 2840)
GPU temp: 81.0°C / 177.8°F — Fan speed: 58% / 2430 RPM
Current clock: GPU core: 1136 MHz, mem: 3004 MHz
Max clock: base: 967 MHz, boost: 1045 MHz, mem: 3004 MHz
GPU load: 96.0% — VDDC: current: 1.150 V
GPU power: current: 101.3 % TDP, target: 100.0 % TDP
Driver: 9.18.13.3288 (3-21-2014), R332.88 (branch: r331_00-402)
OpenGL: OpenGL 4.4 (GeForce GTX 670/PCIe/SSE2 with 317 ext.)
OpenCL: OpenCL 1.1, GeForce GTX 670 compute units:7@1045MHz
CUDA: GeForce GTX 670 CC:3.0, multiprocessors:7@1045MHz
PhysX: PhysX 9131220, GPU PhysX (NVIDIA GeForce GTX 670)
Multi-GPU: no multi-GPU support (1 physical GPUs)

OpenGL and OpenCL demos
GL 4.x - Tessellation — CL GPU - 4D Quaternion Julia Se
OpenGL demos Start ▷ — OpenCL demos Start ▷

Build: 1.18.1 [Jun 17 2013 @ 16:55:36] — Quit



Geeks3D GPU Caps Viewer - GPU 1: 82°C

GPU | OpenGL | CUDA | OpenCL | Tools | About

Number of CUDA devices: 1 — 1: GeForce GTX 670

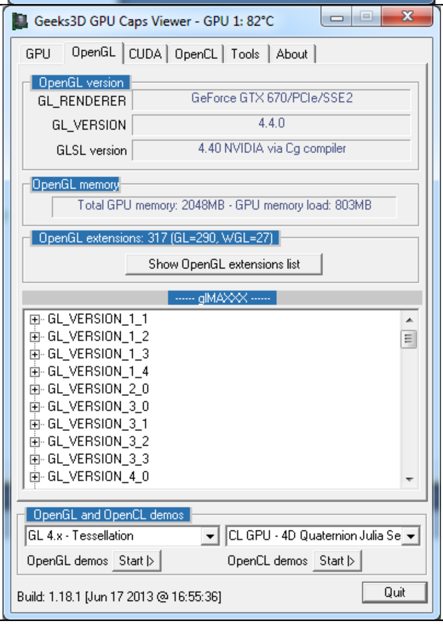
Core data
Name: GeForce GTX 670
Compute Capability: 3.0
Shader Clock: 1045 MHz
MultiProcessors: 7
Warp Size: 32
Grid Size: 2147483647 x 65535 x 65!
Block Dim: 1024 x 1024 x 64
Thread / Block: 1024
Registers / Block: 65536

Memory data
Total Memory: 2048 MB
Shared Mem / Block: 48 KB
Total Constant: 64 KB
Tex. Alignment: 512 Bytes

CUDA @ Geeks3D.com — CUDA @ NVIDIA

OpenGL and OpenCL demos
GL 4.x - Tessellation — CL GPU - 4D Quaternion Julia Se
OpenGL demos Start ▷ — OpenCL demos Start ▷

Build: 1.18.1 [Jun 17 2013 @ 16:55:36] — Quit



Geeks3D GPU Caps Viewer - GPU 1: 82°C

GPU | OpenGL | CUDA | OpenCL | Tools | About

OpenGL version
GL_RENDERER: GeForce GTX 670/PCIe/SSE2
GL_VERSION: 4.4.0
GLSL version: 4.40 NVIDIA via Cg compiler

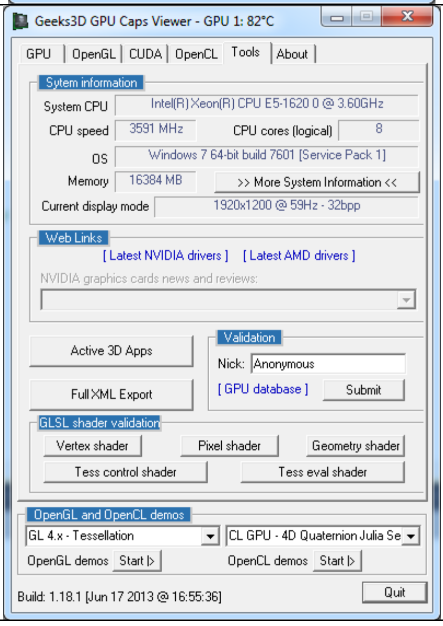
OpenGL memory
Total GPU memory: 2048MB - GPU memory load: 803MB

OpenGL extensions: 317 (GL=290, WGL=27)
Show OpenGL extensions list

------ glMAXXX ------
GL_VERSION_1_1
GL_VERSION_1_2
GL_VERSION_1_3
GL_VERSION_1_4
GL_VERSION_2_0
GL_VERSION_3_0
GL_VERSION_3_1
GL_VERSION_3_2
GL_VERSION_3_3
GL_VERSION_4_0

OpenGL and OpenCL demos
GL 4.x - Tessellation — CL GPU - 4D Quaternion Julia Se
OpenGL demos Start ▷ — OpenCL demos Start ▷

Build: 1.18.1 [Jun 17 2013 @ 16:55:36] — Quit



Geeks3D GPU Caps Viewer - GPU 1: 82°C

GPU | OpenGL | CUDA | OpenCL | Tools | About

Sytem information
System CPU: Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz
CPU speed: 3591 MHz — CPU cores (logical): 8
OS: Windows 7 64-bit build 7601 [Service Pack 1]
Memory: 16384 MB — >> More System Information <<
Current display mode: 1920x1200 @ 59Hz - 32bpp

Web Links
[Latest NVIDIA drivers] [Latest AMD drivers]
NVIDIA graphics cards news and reviews:

Active 3D Apps — Full XML Export

Validation
Nick: Anonymous
[GPU database] — Submit

GLSL shader validation
Vertex shader — Pixel shader — Geometry shader
Tess control shader — Tess eval shader

OpenGL and OpenCL demos
GL 4.x - Tessellation — CL GPU - 4D Quaternion Julia Se
OpenGL demos Start ▷ — OpenCL demos Start ▷

Build: 1.18.1 [Jun 17 2013 @ 16:55:36] — Quit

2. Datasets

a) Aneurism – 256x256x256 – unsigned char (8 bits)

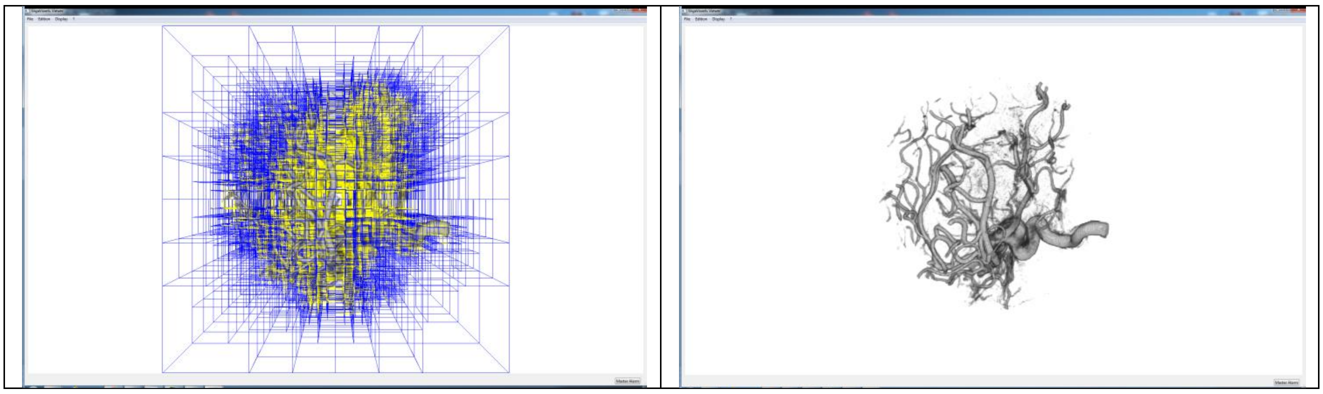
Rotational C-arm x-ray scan of the arteries of the right half of a human head. A contrast agent was injected into the blood and an aneurism is present.

Source : http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/datasets.html

Size / content / spacing : 256 x 256 x 256 - unsigned char (8 bits) – 1:1:1

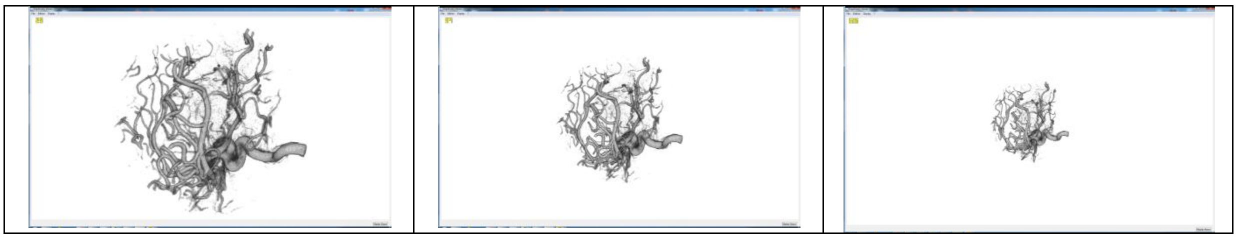
Window Size : 1900 x 1069

i. View of the octree



ii. FPS

(left to right) : 29 - 57 – 126



GigaVoxels Cache :
  Node Pool : 8 Mo (node cache : octree)
  Data Pool : 256 Mo (data cache : voxels)

b) Head Aneuyrism – 512x512x512 – unsigned short (16 bits)

Rotational angiography scan of a head with an aneurysm. Only contrasted blood vessels are visible.

Source : http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/new.html

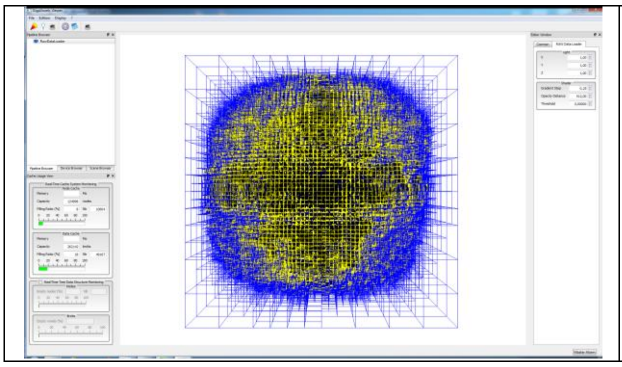
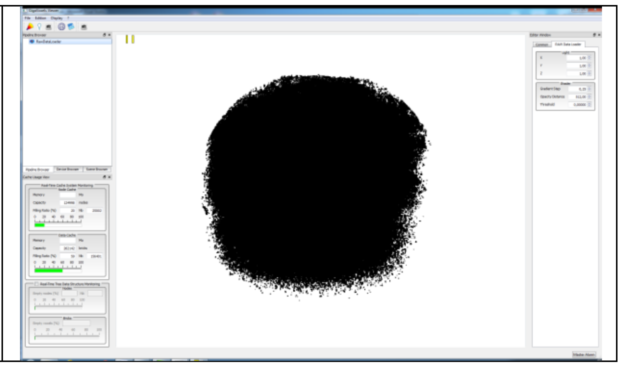
Size / content / spacing : 512 x 512 x 512 - unsigned short (16 bits) – 0.1953, 0.1953, 0.1953

Note : only 12 bits set

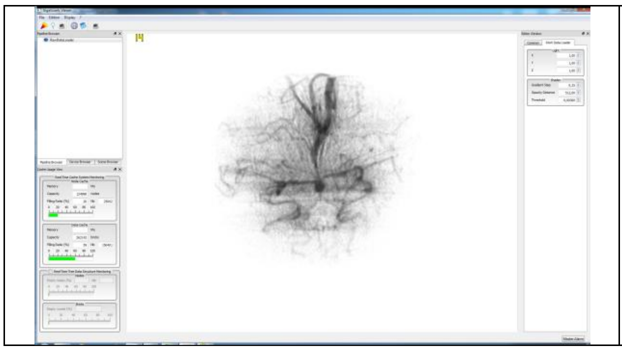
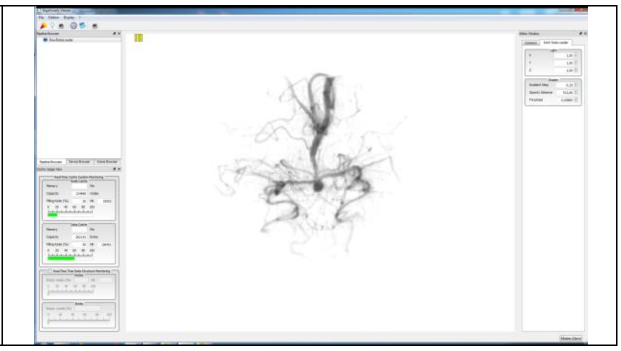
Window Size : 1900 x 1069

i. View of the octree

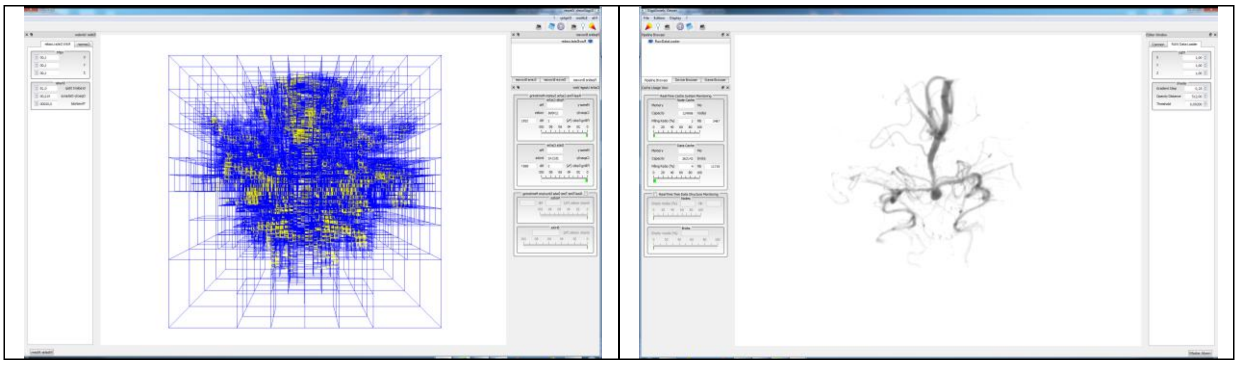
Without thresholding, we have bad octree due to noise.

We can use thresholding at run-time, but the associated octree remains the same. (more threshold on right)

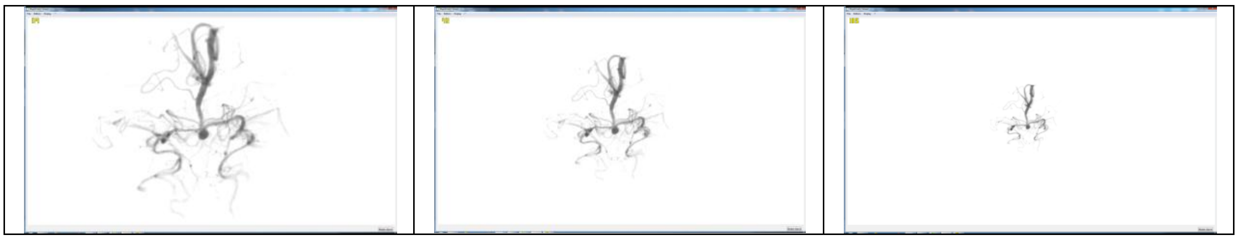



If we use thresholding during mipmap-pyramid construction, we have better octree. But it’s not interactive, it’s a long process done manually. It may be done automatically with an histogram ?



ii. FPS

(left to right) : 34 - 72 – 189



GigaVoxels Cache :

Node Pool : 8 Mo (node cache : octree)
Data Pool : 512 Mo (data cache : voxels)

c) Foot – 256x256x256 – unsigned char (8 bits)

Rotational C-arm x-ray scan of a human foot. Tissue and bone are present in the dataset.

Source : http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/datasets.html

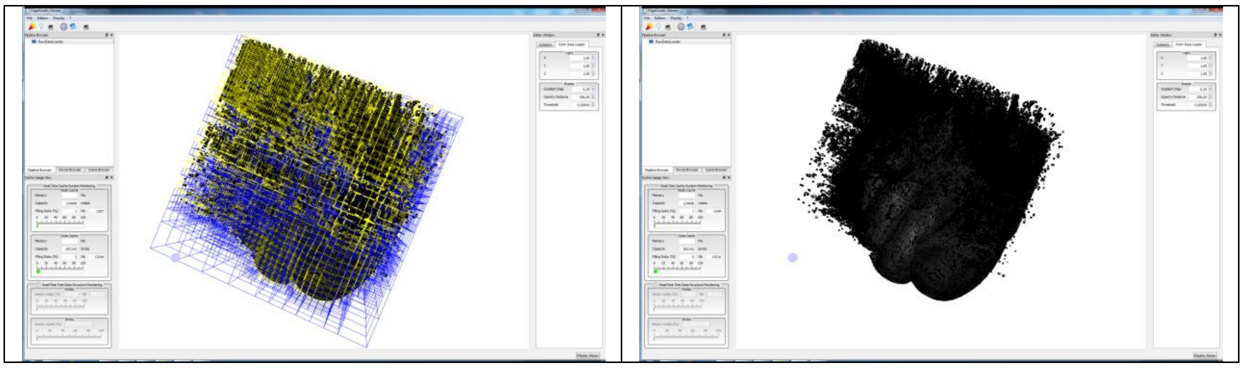
Size / content / spacing : 256 x 256 x 256 - unsigned char (8 bits) – 1:1:1

Window Size : 1900 x 1069

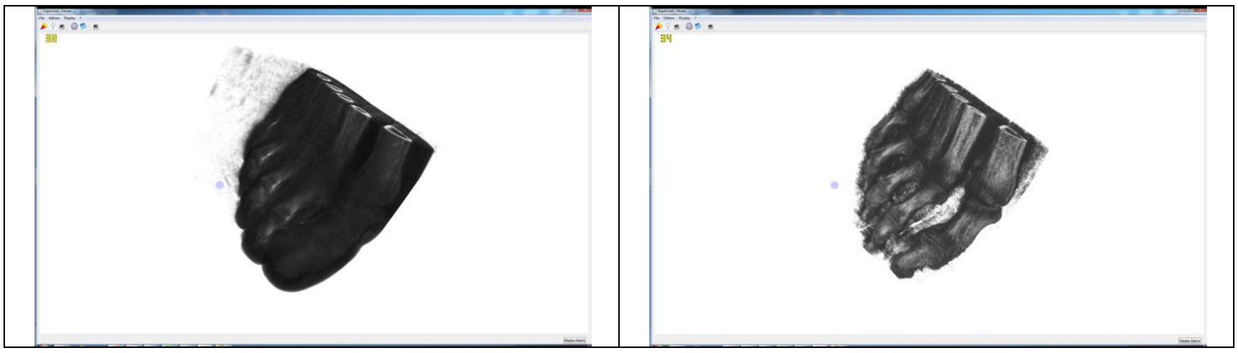
a

i. View of the octree

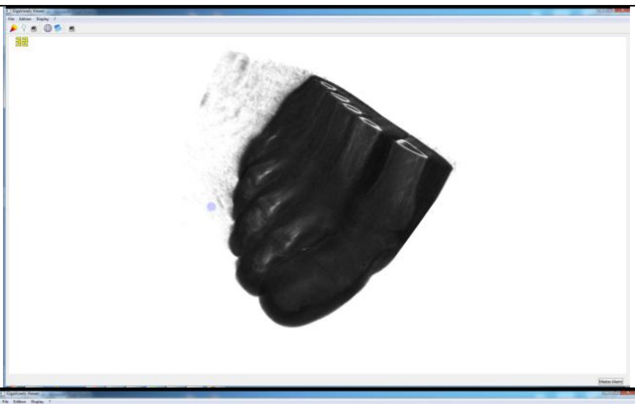
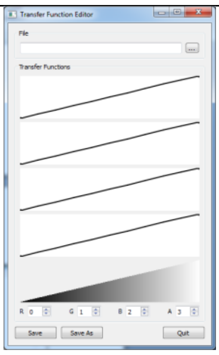
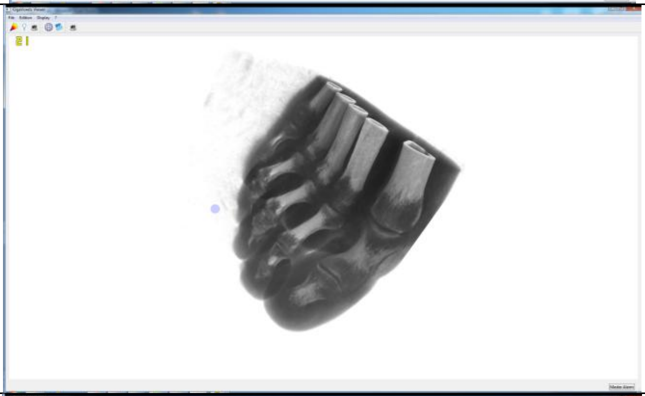
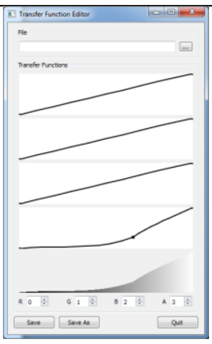
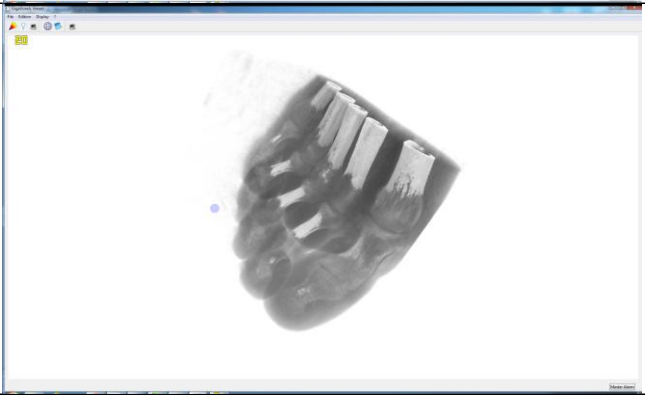
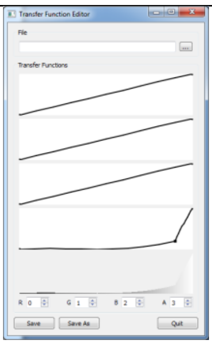
Without thresholding, we have bad octree due to noise.



We can use thresholding at run-time, but the associated octree remains the same. (more threshold on right)

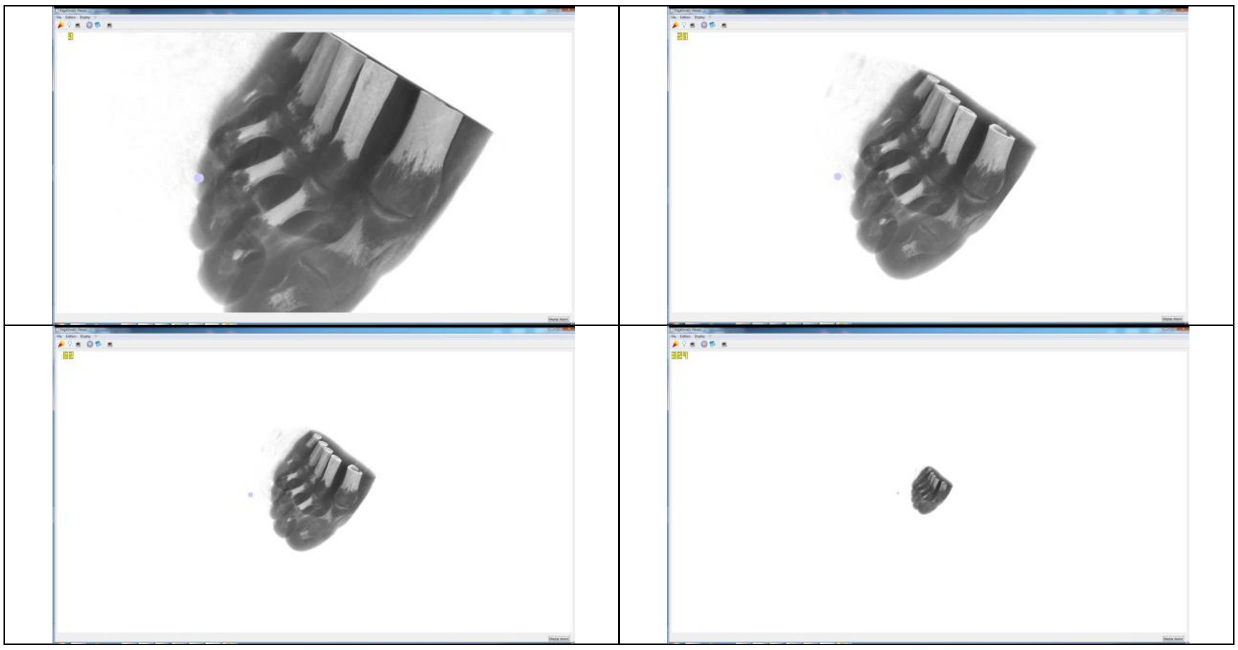


If we use alpha in transfer function in real-time, we can remove skin on bones.

	 Transfer Function Editor File Transfer Functions R 0 G 1 B 2 A 3 Save Save As Quit
	 Transfer Function Editor File Transfer Functions R 0 G 1 B 2 A 3 Save Save As Quit
	 Transfer Function Editor File Transfer Functions R 0 G 1 B 2 A 3 Save Save As Quit

ii. FPS

(left to right) : 9 - 20 – 62 - 324



GigaVoxels Cache :

Node Pool : 8 Mo (node cache : octree)
Data Pool : 256 Mo (data cache : voxels)

d) Exemple 3D – 512x512x512 – unsigned char 4 (32 bits)

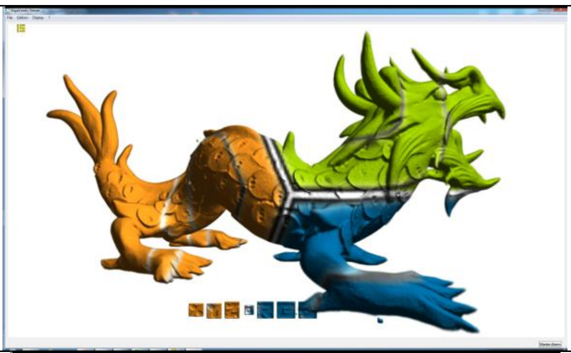
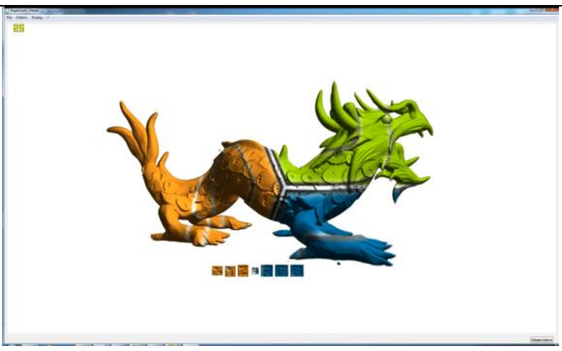
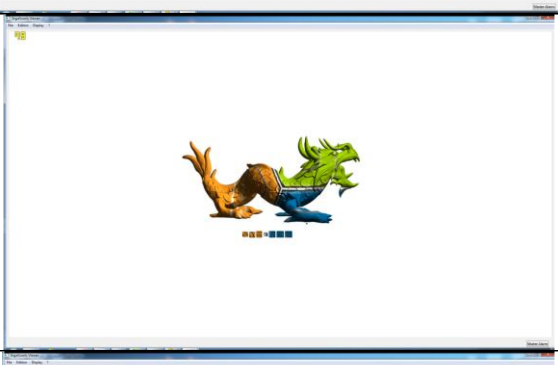
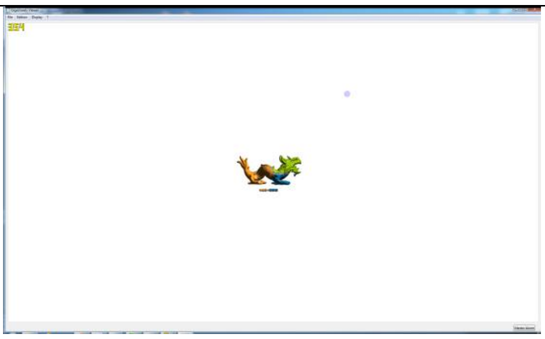
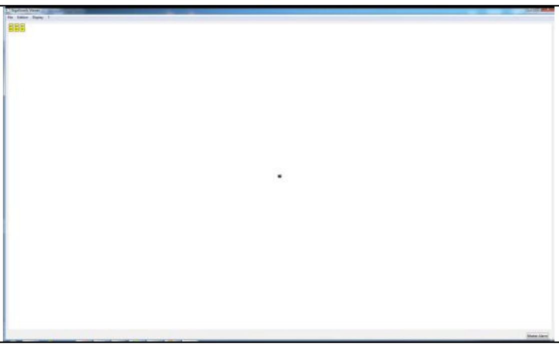
512x512x512 - “unsigned char 4” : rgb+a (4 components color)

Normals are computed at real-time for shading (appearence) by central finite differences (6 neighbors)

Window Size : 1900 x 1069

i. FPS

FPS : 15 – 25 – 79 – 354 – 597 – 666 (as distance to viewer increase)

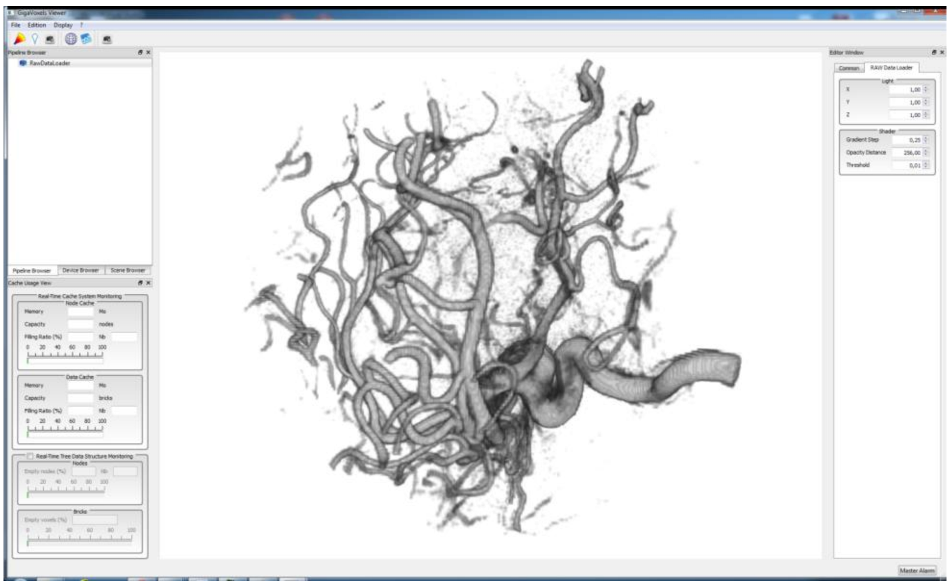
	
	
	

GigaVoxels Cache :

Node Pool : 8 Mo (node cache : octree)
Data Pool : 256 Mo (data cache : voxels)

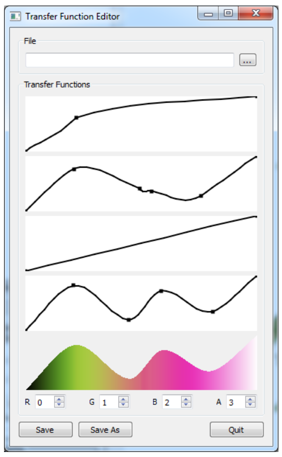
Tools

1. Viewer – Customizable editors



2. Customizable Transfer Function Editor

RGB + A
But customizable



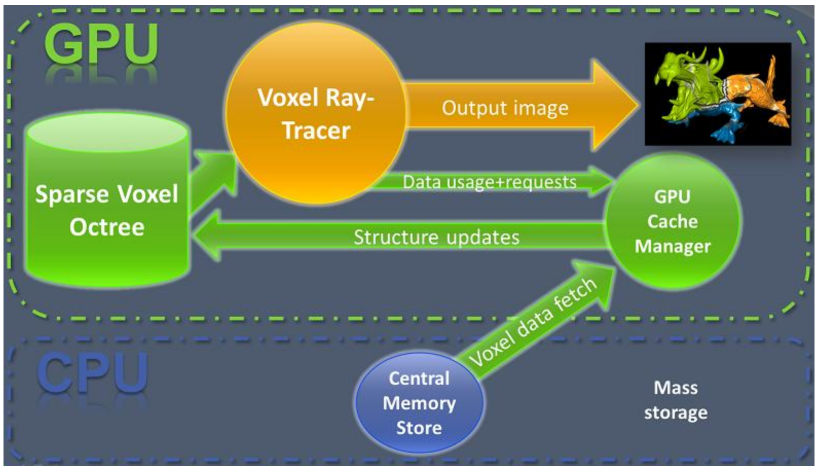
API Features

1. Data Production Management

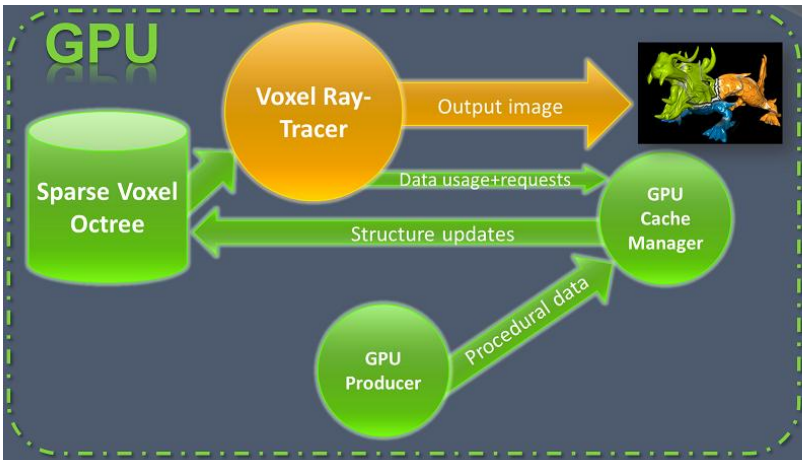
The Data Production Management is responsible for the production of nodes first and then bricks.

There is two different schemes available:
- quality before
- or performances versus quality trade-off

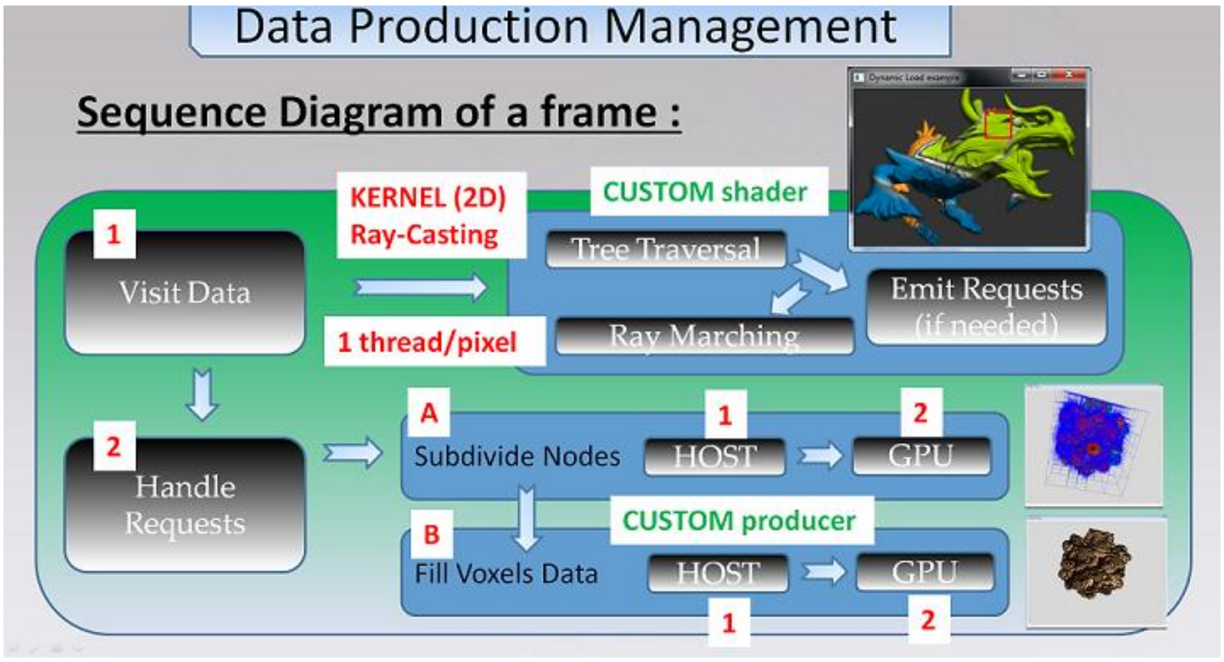
Pipeline of a CPU producer loading file on disk.



Pipeline of a GPU producer.



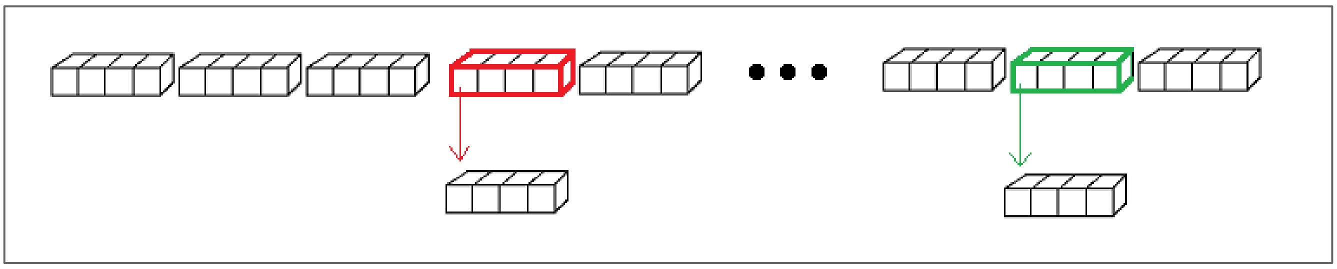
The goal is to handle the requests emitted during the previous rendering stage.



Node production

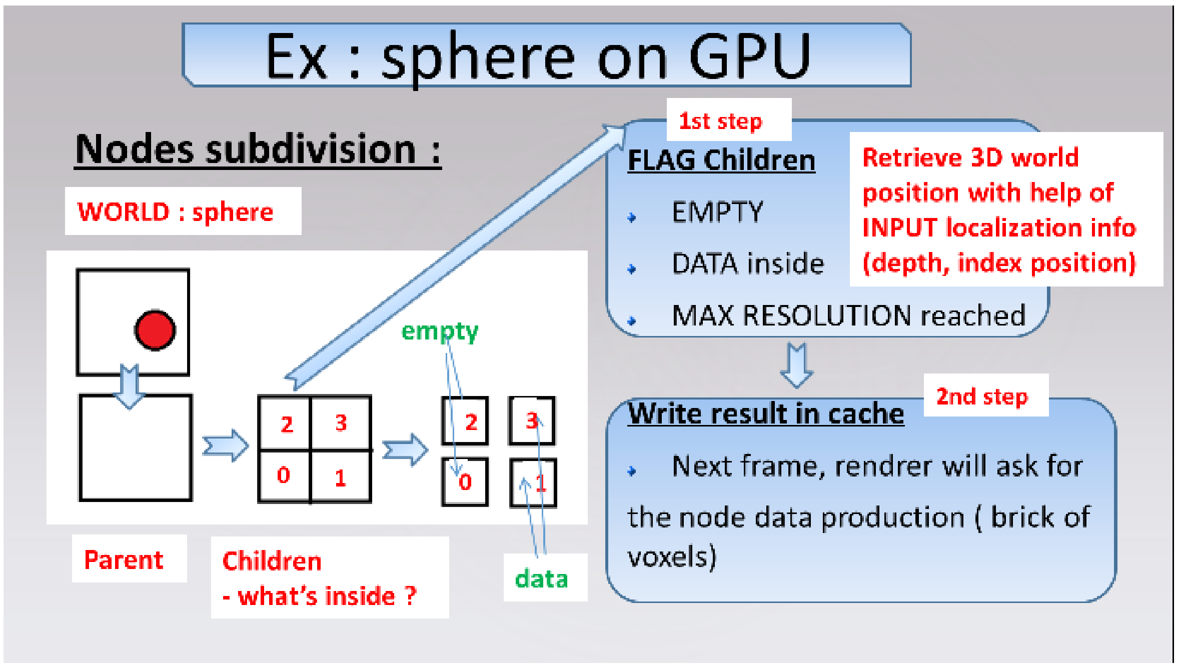
The node pool is a cache in which produced nodes are stored. This cache is an unordered set of nodetiles any level of resolution.

1D memory cache of nodes (in this example a nodetile is made of 4 children)



During the nodes production, user is given the start address of an unused chunk of memory, that consists of a 1D region in cache, the same size as the nodetiles (ex : 2x2x2 memory slots).

At each frame, a kernel is launched to produce a given number of nodes. It is configured so that 1 bloc of threads is in charge of the production of 1 nodetile.



Ex : sphere on GPU

Nodes subdivision :

WORLD : sphere

1st step

FLAG Children

- EMPTY
- DATA inside
- MAX RESOLUTION reached

Retrieve 3D world position with help of INPUT localization info (depth, index position)

Write result in cache

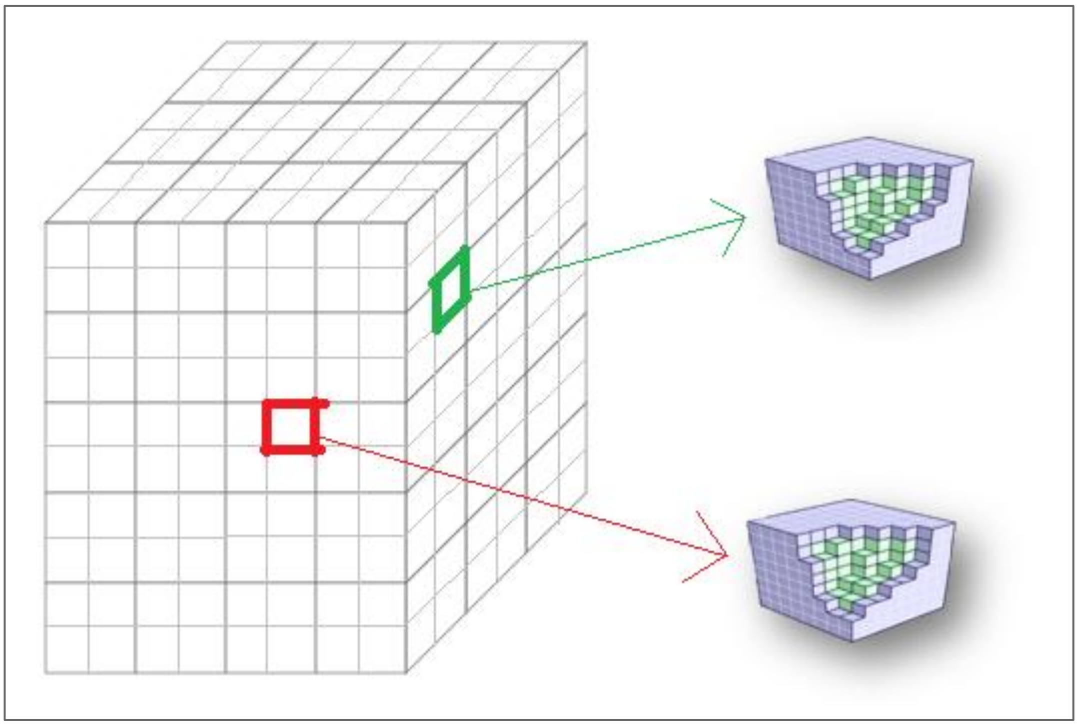
2nd step

- Next frame, renderer will ask for the node data production (brick of voxels)

Data Production

Each user defined channel in the data structure is stored in a large 3D texture. Actually, this texture is the cache in which produced bricks of data are stored. This cache is an unordered set of bricks of data at any level of resolution.

3D memory cache for a given user defined channel (i.e color, density, normal, etc...)



During the bricks production, user is given the start address of an unused chunk of memory, that consists of a 3D region in cache, the same size as the bricks (ex : 8x8x8 + 1 border => 10x10x10 memory slots).

At each frame, a kernel is launched to produce a given number of bricks. It is configured so that 1 bloc of threads is in charge of the production of 1 brick of data.

Limitations

Only N3-tree (octrees, etc) :

same size (512x512x512 is OK, but not un-uniform 120x512x512)

and same spacing in the regular 3D grid

Conclusion

...

Bibliography

[1] Authors – Title – *Proceedings of …*, March 2003.

[2] Authors – Title2 – *INRIA Research Report n°????*, 20

[3] Andrei Alexandrescu – Modern C++ Design: Generic Programming and Design Patterns Applied – *Book (Addison Wesley)*, 2001

[4] Tomas Akenine-Moller, Eric Haines, Naty Hoffman – Real-Time Rendering (Third Edition) – *Book (A K Peters)*, 2008

[5] Richard Wright, Nicholas Haemel, Graham M. Sellers, Benjamin Lipchak – OpenGL SuperBible: Comprehensive Tutorial and Reference (Fifth Edition) – *Book (Addison Wesley)*, 2010

[6] Pascal Guehl, Fabrice Neyret – GigaVoxels, librairie et kit de développement sur GPU pour l'exploration temps-réel et visuellement réaliste d'immenses scènes détaillées à base de SVO – *AFIG, Journées de l'Association Française d'Informatique Graphique*, Calais, France, 2012

[7] Pascal Guehl, Fabrice Neyret – GigaVoxels, Real-time Voxel-based Library to Render Large and Detailed Objects – *GTC NVidia (GPU Technology Conference)*, San Jose, California, 2013

[8] Mike Bailey, Steve Cunningham – Graphics Shaders: Theory and Practice (Second Edition) – *Book (A K Peters)*, 2011

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399