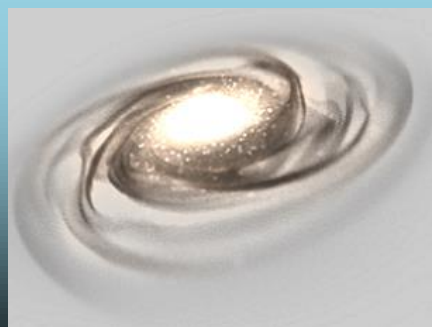
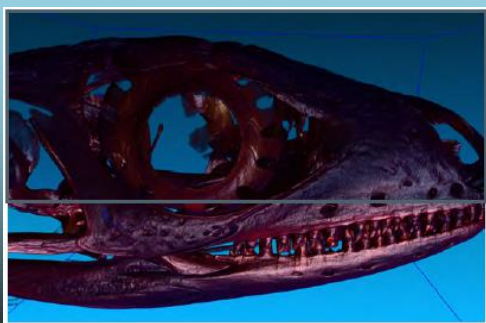


GigaVoxels, Behind the Scene “programming guide”

Pascal Guehl , Fabrice Neyret



TECHNICAL REPORT

N° 9999

23/12/2012

Project-Team Maverick



GigaVoxels, Behind the Scene “programming guide”

Pascal Guehl¹ , 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

GigaVoxels, Behind the Scene “programming guide”

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

1. Introduction	6
1. Coding Rules	6
2. Simple Sphere tutorial	6
3. Amplified Surface/Volume tutorial	9
4. Depth of Field tutorial	11
5. Dynamic load tutorial	12
6. Menger/Serpinski tutorial	14
7. Mandelbulb tutorial	15
8. Video game tutorial	16
9. Graphics Interoperability	17
Default mechanism	17
10. Rendering mechanism	18
Proxy geometry	18
Graphics Interoperability	18
Optimizations	18
11. Software architecture	19
12. The Cache mechanism	20
1. Using automatic references	28
2.1 Figure	28
2.2 Tables	28
2.3 Bibliography	29
Conclusion	29
Bibliography	30

1. Introduction

This is a sample document for the INRIA RT Word model. Page headers/footers are automatically updated with the information you fill in the File/Properties/Summary (Fichier/Propriétés/Résumé) box. Mandatory fields are Title and Author.

Use automatic references for sections, tables and figures. Refer to section 1 for reference examples.

1. Coding Rules

show the usage of a GPU producer

GvIRenderer = interface are prefixed with I

int pLevel = function parameters are prefixed with "p" standing for "parameter"




int level : local parameters are left unchanged.

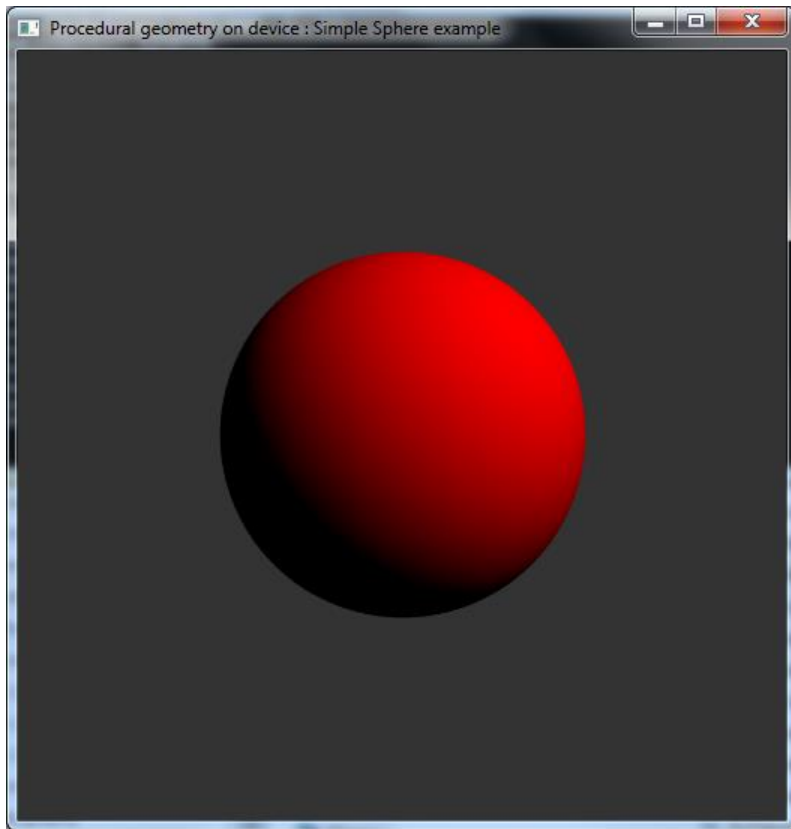
int pLevel = template parameters are prefixed with "T" standing for "template"

2. Simple Sphere tutorial

This example show the usage of a GPU producer used to generate a simple sphere.

The main themes related to volume rendering are :

-  procedural geometry
-  color alpha-premultiplication (to avoid color bleeding)
-  shading model (lambert illumination model)



The main themes related to volume rendering are

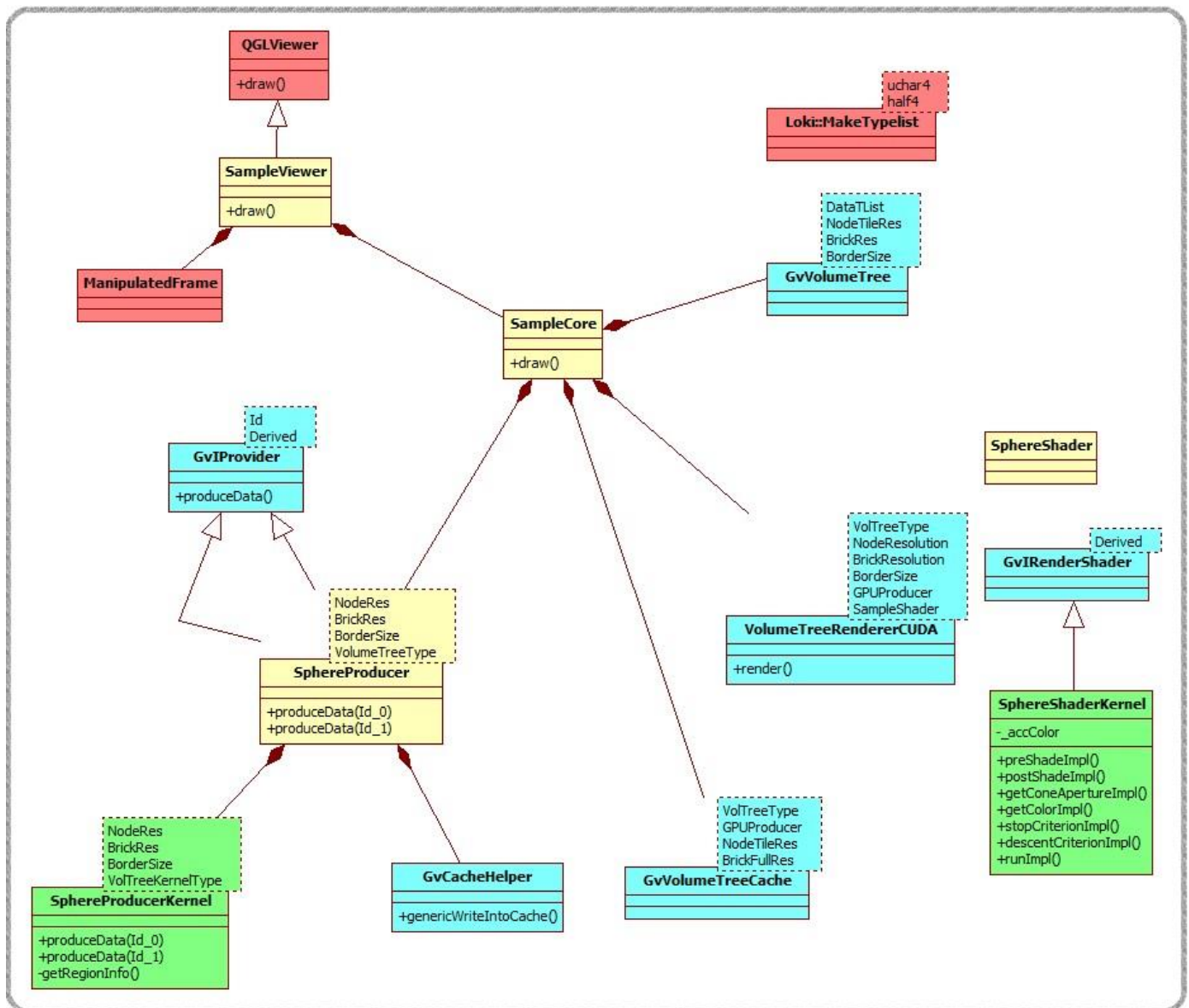
The following screenshot is a simplified class Diagram of the classes used in this tutorial.

Different colors are used to ease comprehension:

- ✚ RED : third party dependency classes (QGLViewer, Loki, etc...)
- ✚ BLUE : GigaVoxels classes
- ✚ YELLOW : user HOST classes
- ✚ GREEN : user DEVICE classes

The SampleCore class is the main GigaVoxels pipeline containing:

- ✚ the GigaVoxels COMMON classes
 - DATA STRUCTURE : the sparse voxel octree structure GvStructure::GvVolumeTree
 - CACHE : the cache manager system GvStructure::GvVolumeTreeCache
 - RENDERER : the ray-casting renderer GvRenderer::VolumeTreeRendererCUDA
- ✚ the USER defined classes
 - PRODUCER : a host SphereProducer with its associated device-side SphereProducerKernel
 - SHADER : a host SphereShader with its associated device-side SphereShaderKernel



QGLViewer and ManipulatedFrame are IHM classes to create a 3D window and a light interactor.

Data Structure

The content of a voxel is represented by two channels:

- one for RGBA colors (uchar4 type),
- and one for normals (half4 type).

```
typedef Loki::TL::MakeTypelist< uchar4, half4 >::Result DataType;
```

The data structure is an octree (2x2x2) defined by:

```
// Defines the size of a node tile
```



```

typedef gigavoxels::StaticRes1D< 2 > NodeRes;

// Defines the size of a brick
typedef gigavoxels::StaticRes1D< 8 > BrickRes;

// Defines the size of the border around a brick
enum { BrickBorderSize = 1 };

// Defines the total size of a brick
typedef gigavoxels::StaticRes1D< 8 + 2 * BrickBorderSize > RealBrickRes;

// Defines the type of structure we want to use. Array3DGPUPTex is the type of array used to store the
bricks.
typedef  gigavoxels::VolumeTree<  DataType,  gigavoxels::Array3DGPUPTex,  NodeRes,  BrickRes,
BrickBorderSize > VolumeTreeType;

```

Now let's have a look at the SphereProducerKernal code. As said before, users have to write a produceData() method for each of the two channel :

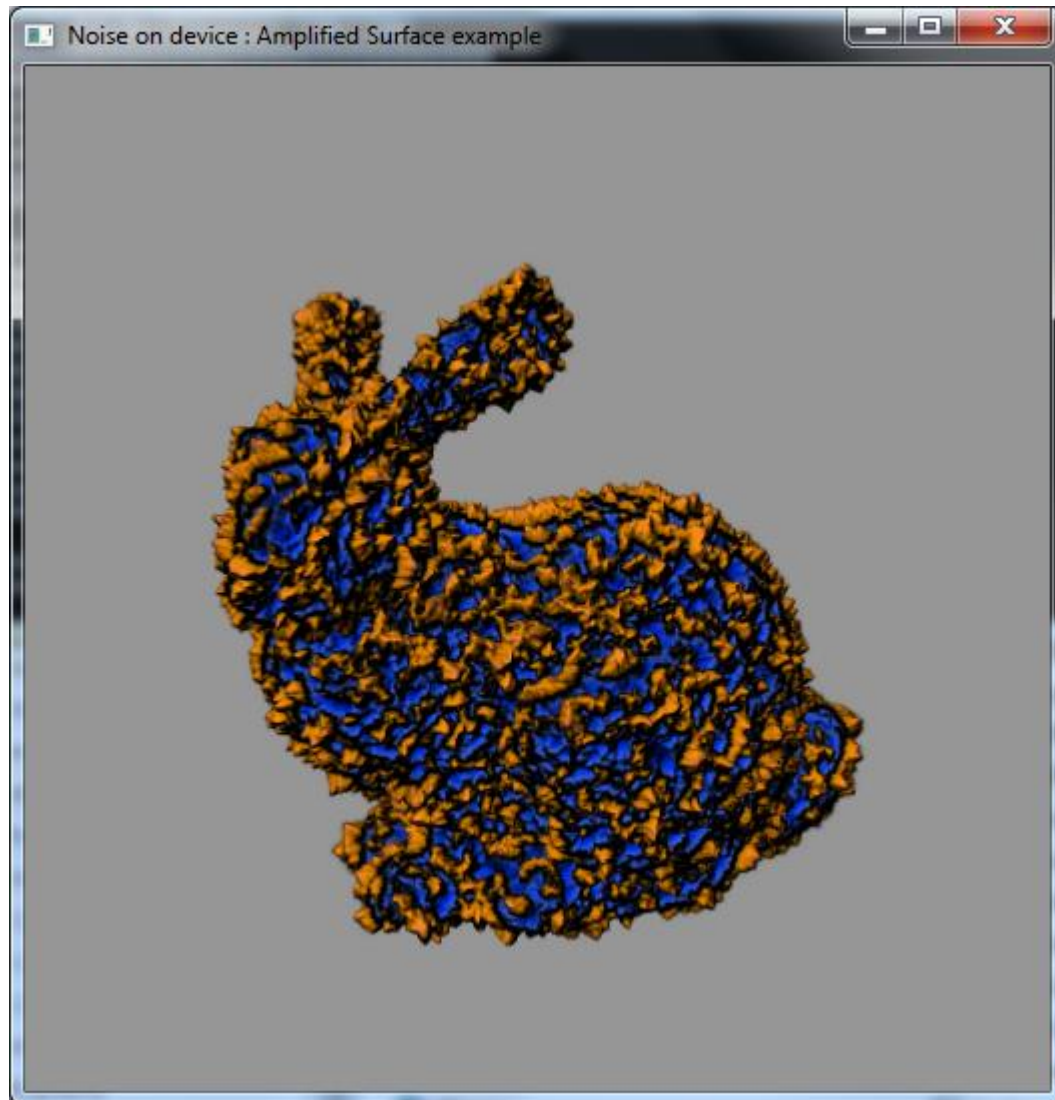
- 🔧 node tiles,
- 🔧 bricks of voxels.

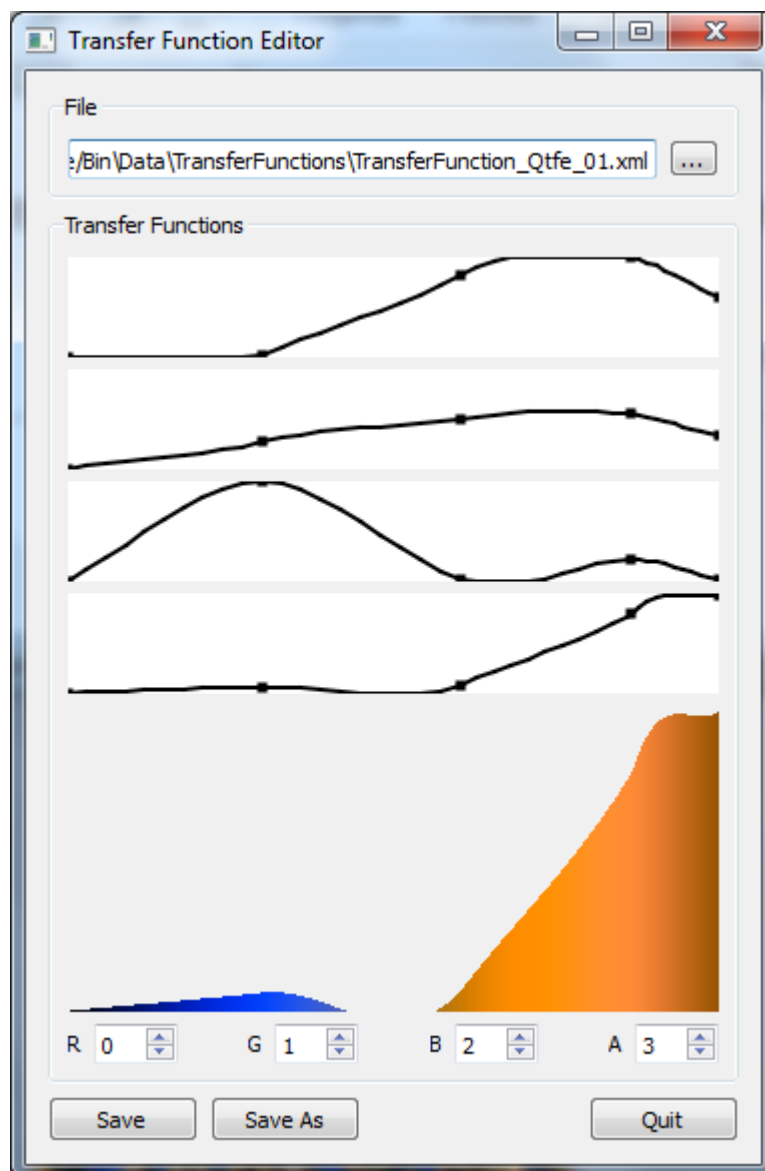
3. Amplified Surface/Volume tutorial

This example show the usage of a GPU producer used to generate a simple sphere.

- 🔧 The main themes related to volume rendering are

This example show the usage of a GPU producer used to generate a simple sphere.





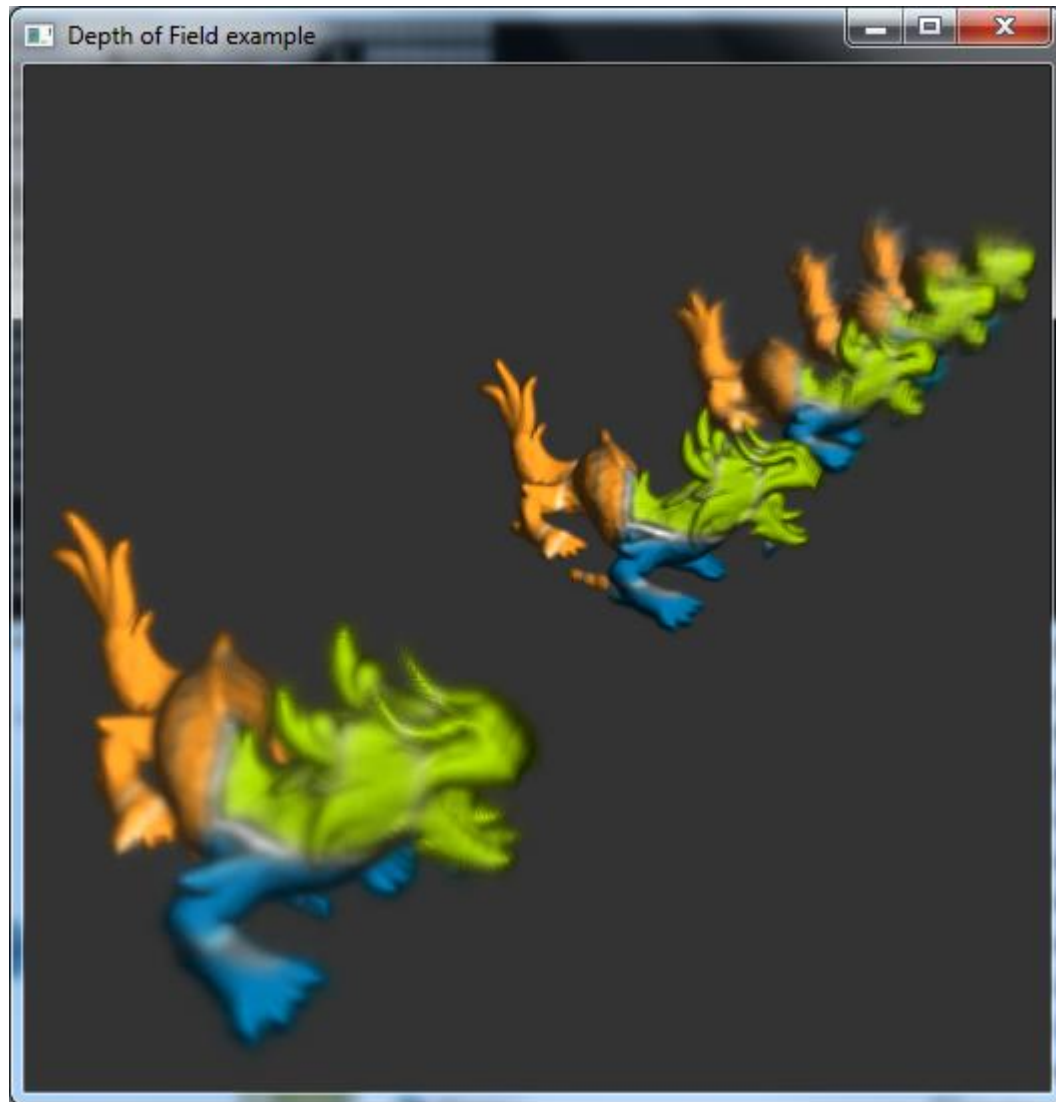
4. Depth of Field tutorial

This example show the usage of a GPU producer used to generate a simple sphere.



The main themes related to volume rendering are

The goal here is to reproduce the “circle of confusion” mechanism.

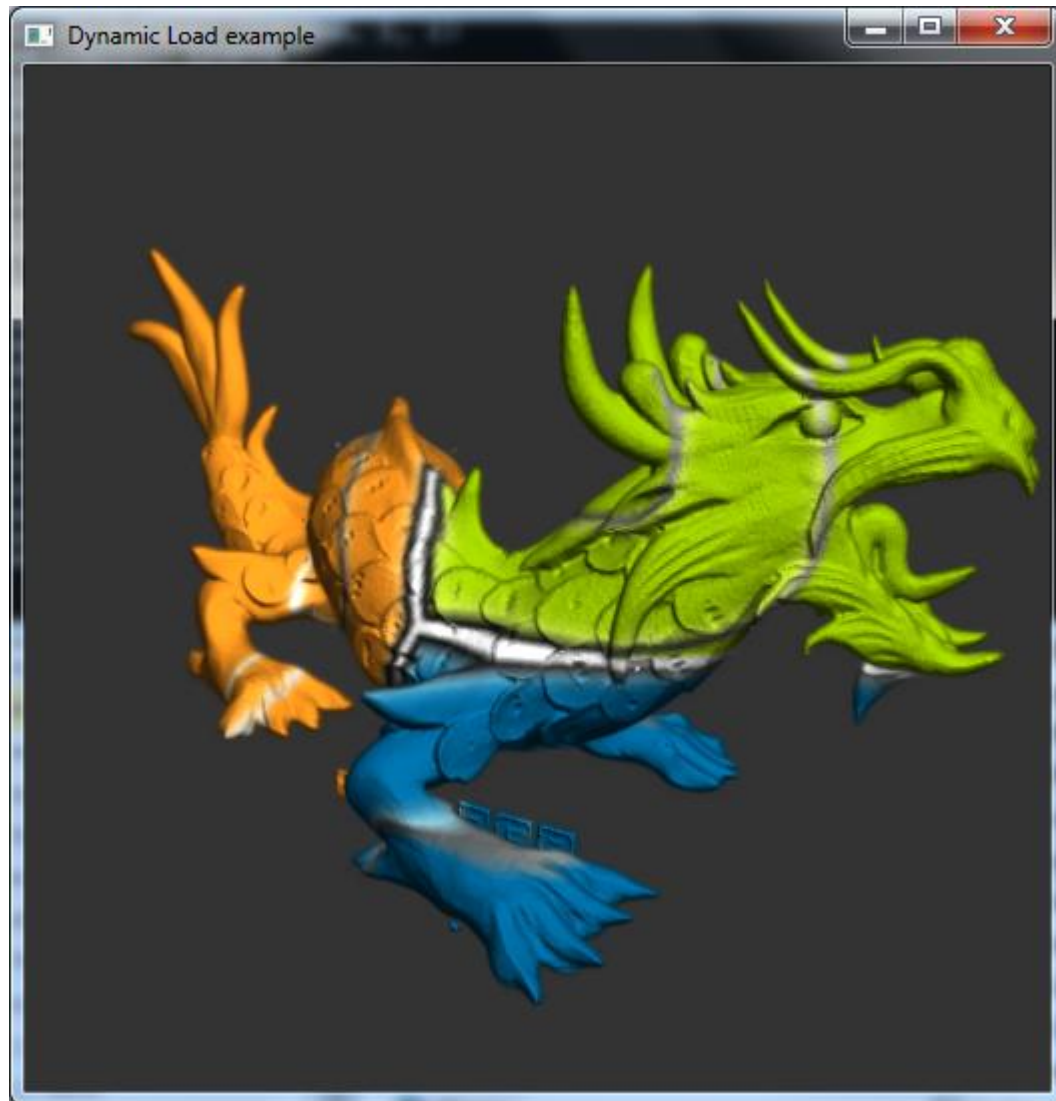


5. Dynamic load tutorial

This example show the usage of a GPU producer used to generate a simple sphere.

✚ The main themes related to volume rendering are

This example show the usage of a GPU producer used to generate a simple sphere.

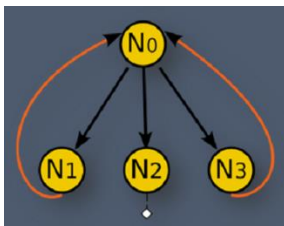
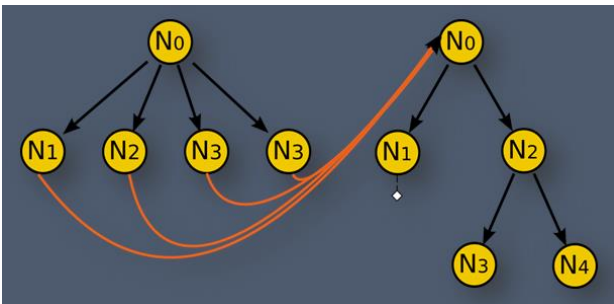


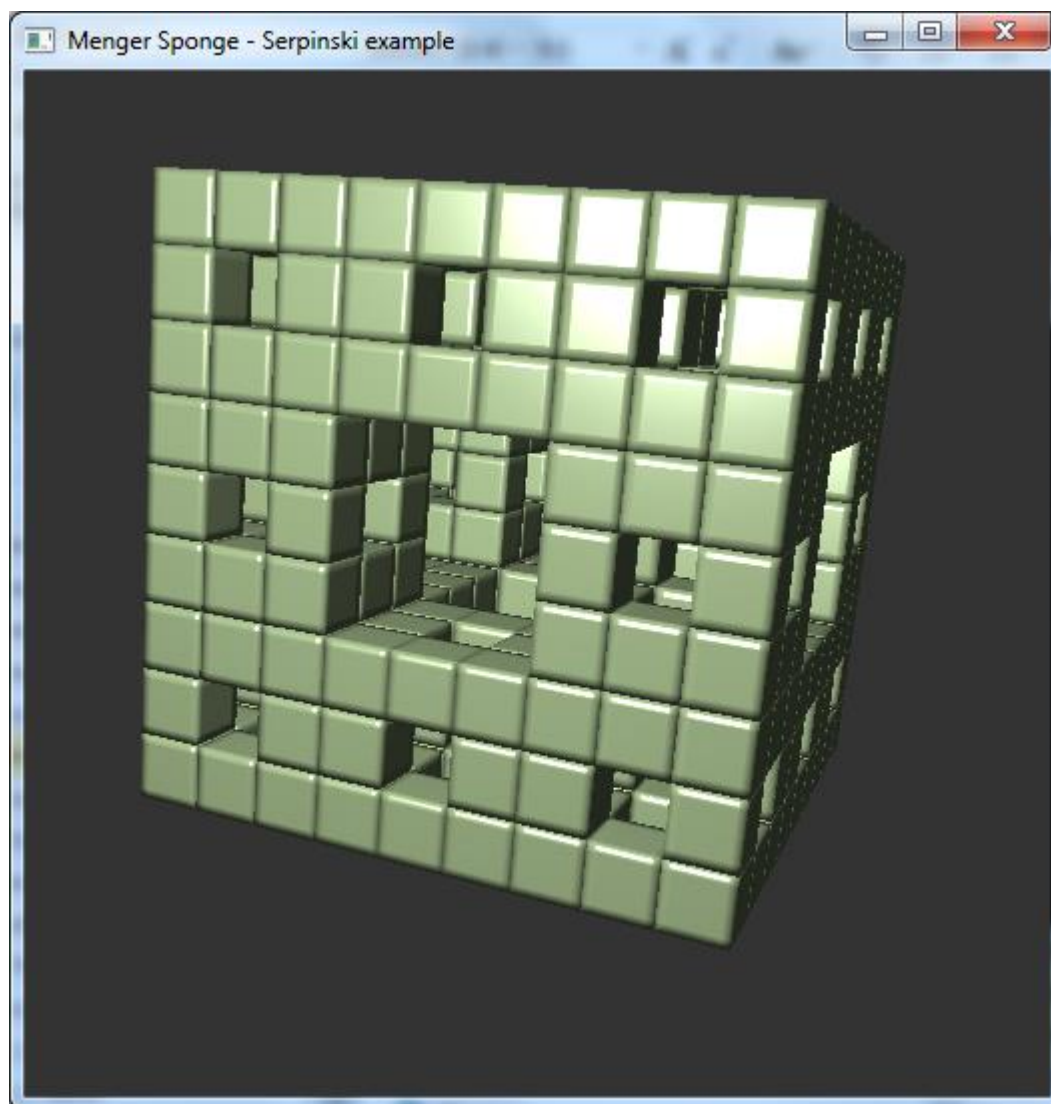
6. Menger/Serpinski tutorial

This example show the usage of a GPU producer used to generate a simple sphere.

✚ The main themes related to volume rendering are

This example show the usage of a GPU producer used to generate a simple sphere.



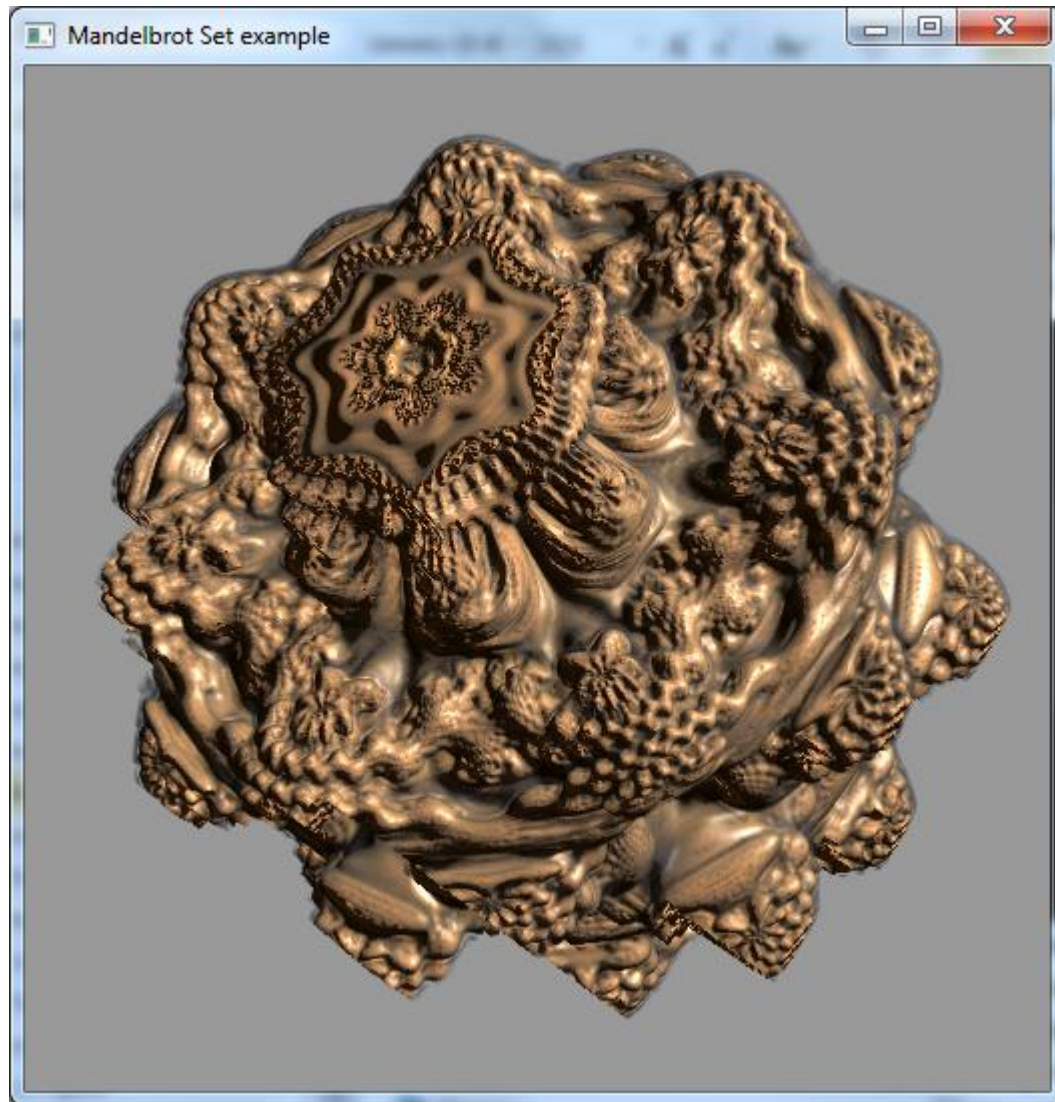


7. Mandelbulb tutorial

This example show the usage of a GPU producer used to generate a simple sphere.

✚ The main themes related to volume rendering are

This example show the usage of a GPU producer used to generate a simple sphere.

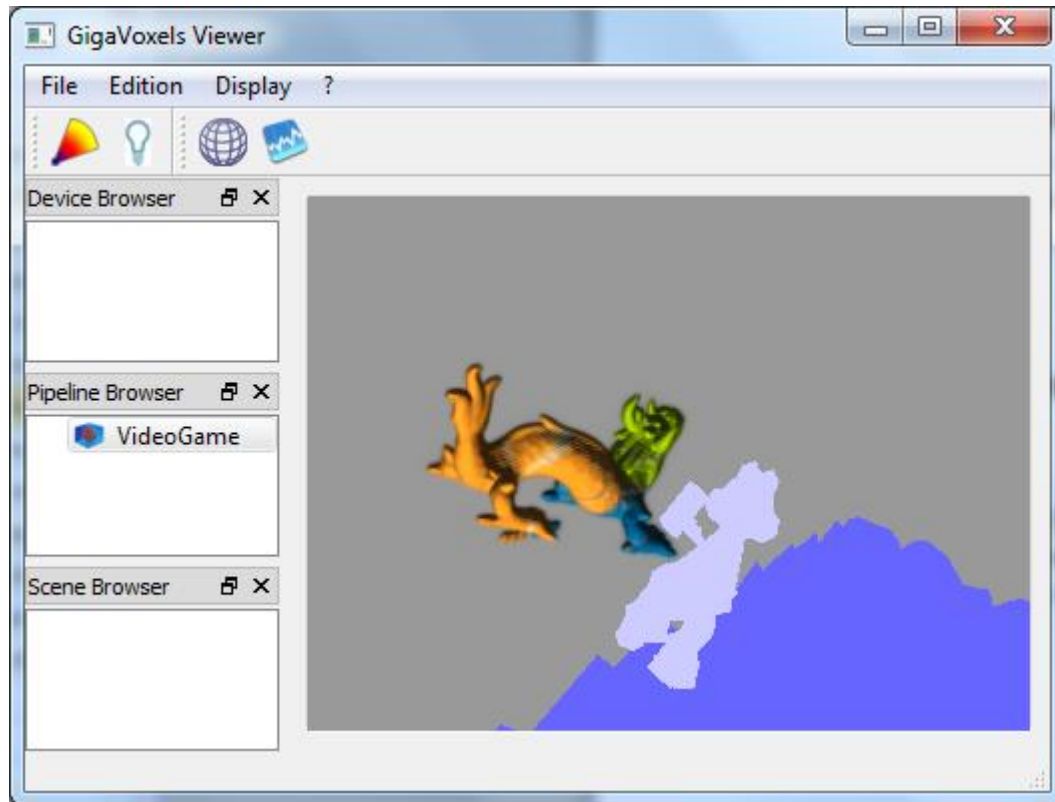


8. Video game tutorial

This example show the usage of a GPU producer used to generate a simple sphere.

✚ The main themes related to volume rendering are

This example show the usage of a GPU producer used to generate a simple sphere.



9. Graphics Interoperability

Default mechanism

The main class is `GvRenderer::GvGraphicsResource`. It is a wrapper to the CUDA Graphics Interoperability API. Users have to declare how the GigaVoxels pipeline will be configured to interact with an OpenGL scene. Here is the most trivial configuration :

```
// Define an OpenGL texture that will be used to display a full quad on screen
GLuint _colorTex;
glGenTextures( 1, &_colorTex );
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, _colorTex );
glTexParameteri( GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
glTexImage2D( GL_TEXTURE_RECTANGLE_EXT, 0, GL_RGBA8, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL );
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, 0 );

// Create CUDA resources from OpenGL
objects_renderer->connect( eColorWriteSlot, _colorTex, GL_TEXTURE_RECTANGLE_EXT );
```

// Internally, it creates CUDA graphics resources mapped from OpenGL

Beware, for the moment CUDA cannot register ImageBuffer from an OpenGL depth buffer. The associated format is not yet supported.

By default, the GigaVoxels render has a graphics interoperability handler that is used to manage the user OpenGL color and depth buffers.




User has to define to the common OpenGL clear color and depth values.

Example : The simple sphere tutorial

We will use two

10. Rendering mechanism

The GigaVoxels pipeline is divided in three main steps:

-  Pre-render
-  Render
-  Post-render

Kljnslknc

Proxy geometry

The GigaVoxels pipeline can used OpenGL Z-buffer information with more precision than the default mechanism. An OpenGL proxy geometry can be rendered in OpenGL with two passes: front and back faces, to retrieve the z-depth in two textures. Then, these textures are used to access the depth info.

Graphics Interoperability

The GigaVoxels pipeline is divided in three main steps:

Optimizations

View Frustum Culling

Launch computation only in 2D projected BBox

11. Software architecture

The GigaVoxels library has been designed as a dynamic link library. It is divided in packages:

- ✚ Core
- ✚ Structure
- ✚ Cache
- ✚ Rendering
- ✚ Utils

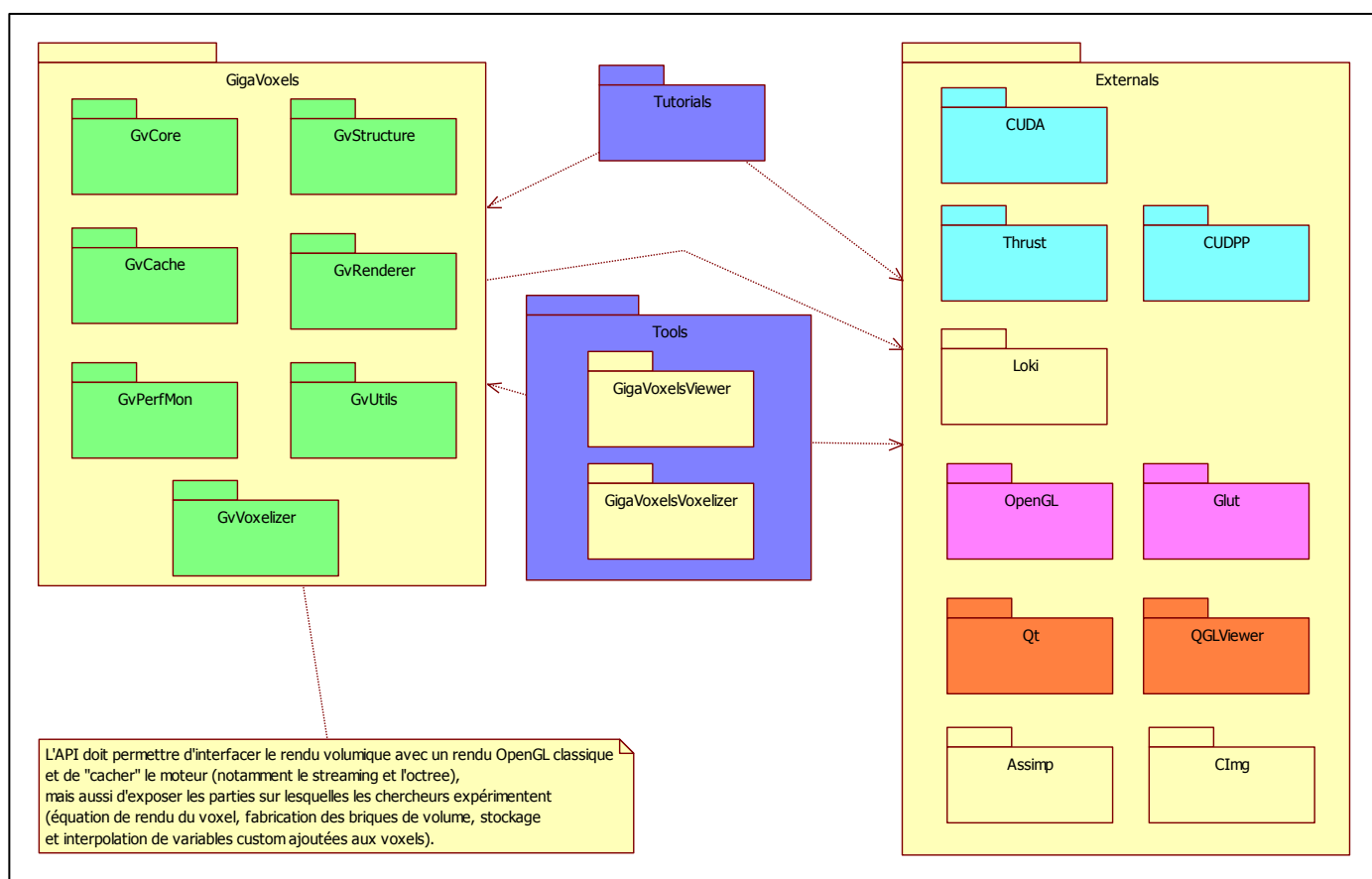
The SDK provides several tutorials. They are executable based on this library.

In order to be able to study, test we have added a viewer.

Viewer.

Below is a screenshot of the main UML diagram of the library. It shows:

- ✚ library modules,
- ✚ external dependencies,
- ✚ tools,
- ✚ tutorials.



12. What is GigaVoxels ?

Data Structure

Tree data management (space partitioning).

Octree, or BSP tree handled on the GPU (to organize your data) [SVO : sparse voxel octree]

Cache System on the GPU

At the core of the system, GigaVoxels is a cache mechanism implemented on GPU. It uses a LRU mechanism (least recently used) in order to get temporal coherency.

Data Production Management

The cache system is associated to the production of data.

Handle are given to be able to generate data on the host (CPU), on the device (GPU) or eventually in an hybrid mode.

Visit algorithm

Traverse your data (loaded in cache) as could be done for rendering.

A stackless traversal beginning from the root has been choosen.

Renderer

On top of this, a renderer has been added to generate images.

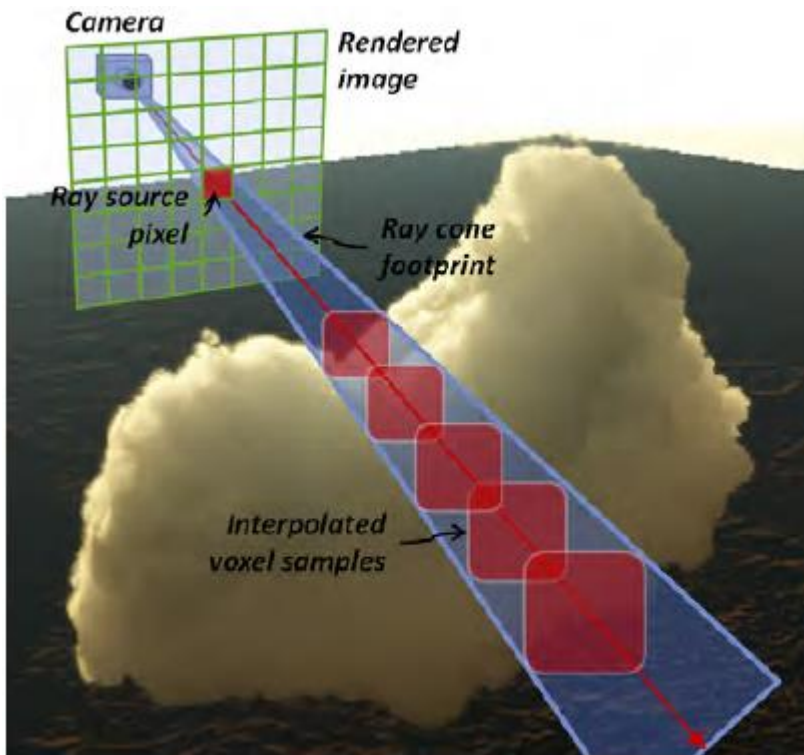
Ray-marching brick by bricks

Using a cone tracing approximation algorithm to avoid anti-aliasing.

It is as if the ray has a thickness.

The cone tracing is used during the rendering phase, but also in the traversal phase.

In fact, rendering and traversal are interleaved.



The Data Structure

Generalized N3-Tree

GigaVoxels uses a generalized N3-Tree, an example could be an octree (2x2x2).

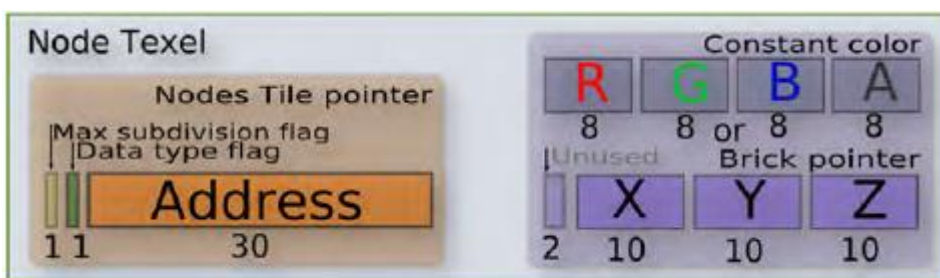
Easy to traverse

Space Partitioning Structure

The node

A node is a simple structure of two elements:

- ✚ the node child address
- ✚ the brick address



For the nodes, their 3D addresses are packed on 30 bits (10 for each dimension), the first 2 bits are used to “flag” their type (terminal node, containing data or not)

For the bricks, we store their addresses on 30 bits too.

These two information are stored in two separate arrays in memory. This is a structure of array (and not an array of structure).

Data

In memory, voxels are grouped in a bigger entity called a brick.

Each brick is spatially contained in its associated node.

Hierarchical data structure

Sparse mipmap pyramid

Most of the time, details are stored at the frontier of objects.

Sparse mipmap pyramid

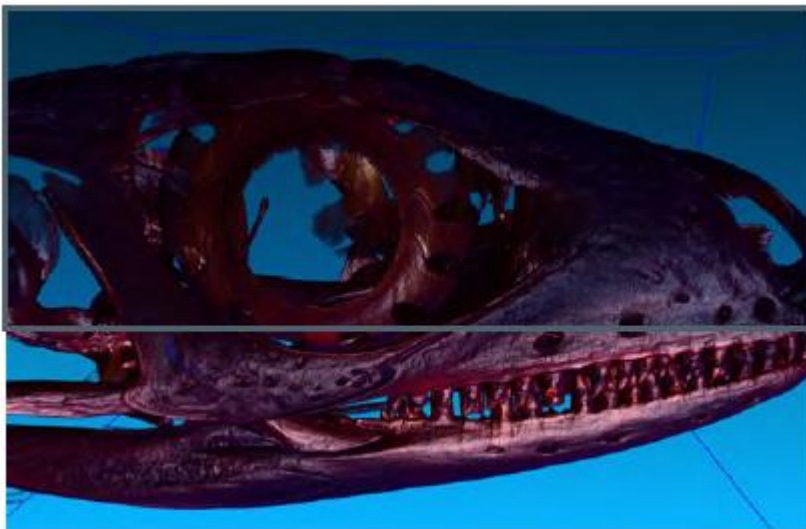
Most of the time, details are stored at the frontier of objects.

Implementation

The space partitioning structure is built “on the fly” on the device (GPU) during the traversal. It is stored in cache.

When user chooses to load data from disk, a voxelizer tool has been used in a pre-process step to generate this structure in files on disk.

Voxels are stored in a mipmap pyramid in 3D textured.



GigaVoxels is supported and funded in the project
ANR "RTIGE" (Real-Time Interactive & Galaxy for Edutainment)
of reference ANR-10-CORD-0006

13. Cache System Management

The GigaVoxels pipeline has two caches:

- ✚ Nodes pool
- ✚ Data pool

Node pool

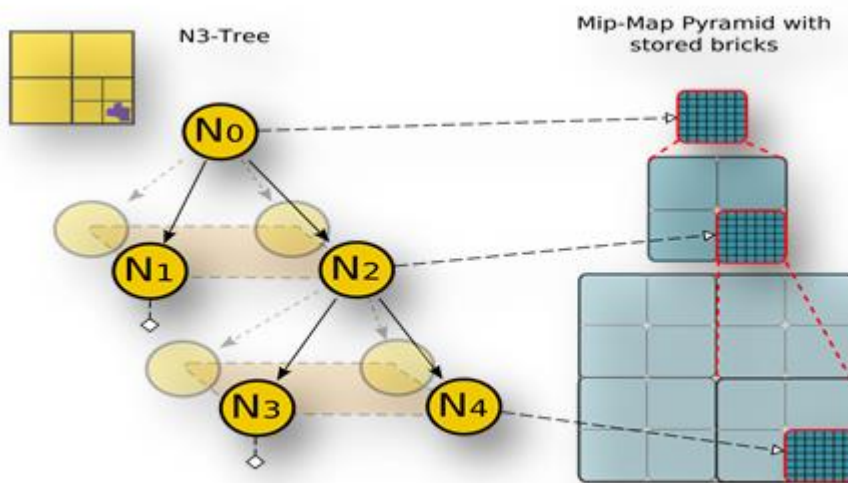
Elements are grouped in node tiles. Each node points to its children.

For instance, for an octree, a node tile is a group of 8 children.

Data pool

The data pool is made of at least one channel of data.

Users have to define all the different channels they want to use.



The cache are based on "surfaces" that act as readable and writable 3D textures available since CUDA 4.1 or 4.2.

3D textures provides hardware trilinear interpolation and internal cache with locality.

Surfaces have no hardware interpolation, but enable us to write voxels data in texture indexed by voxels addresses in the cache.

Cache Updating Strategy

Previously based on the CUDA Thrust library, we use Cudpp, the library written by Mark Harris. Because, it doesn't allocate data at runtime as Thrust do.

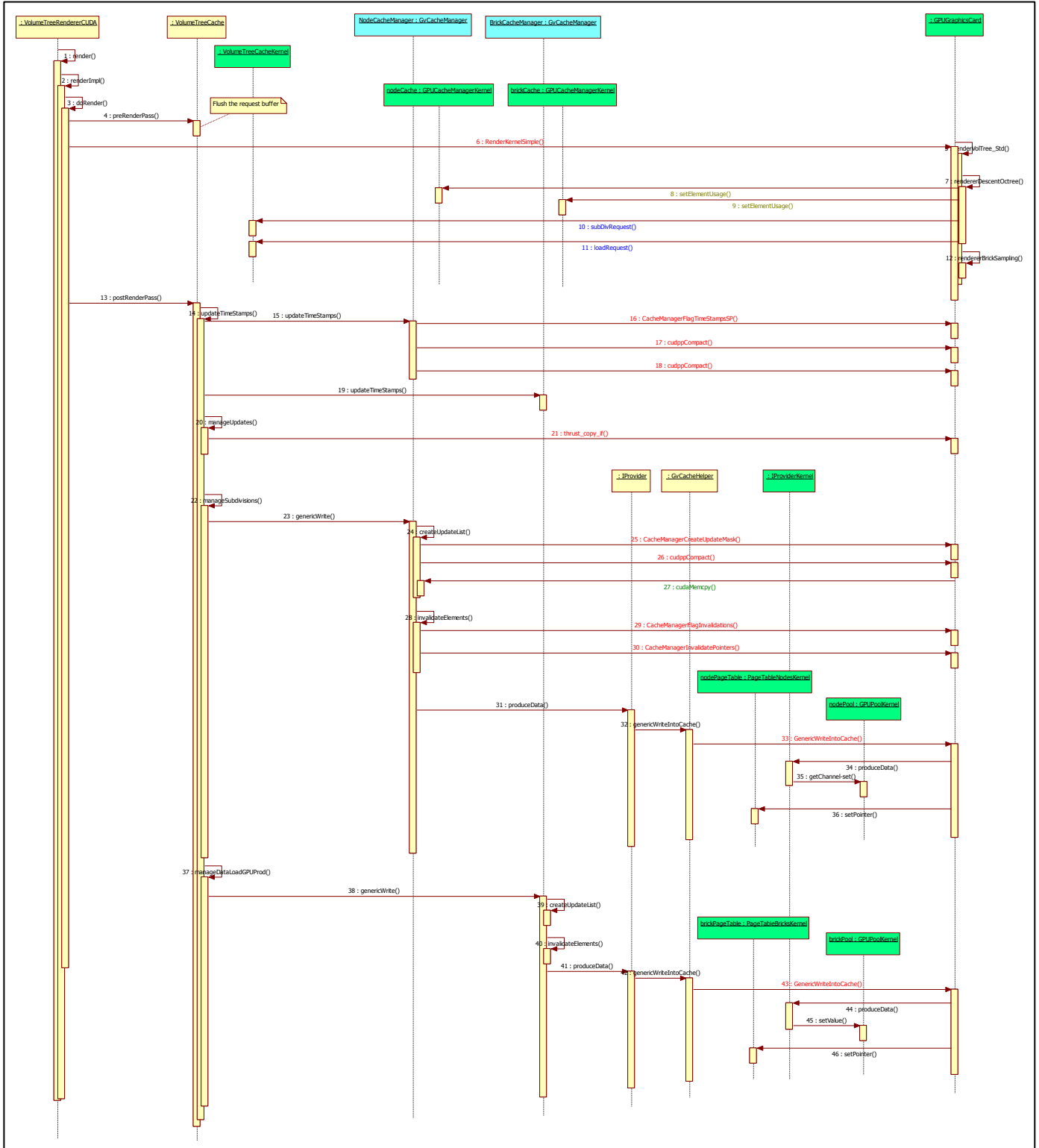
During rendering, all traversed nodes are flagged as “used” . Then, in last post-rendering phase, where all requests are processed, all redundant elements need to be reduced. So, we use stream compaction algorithms.

Three steps are used:

- ✚ Find the minimal set of used elements
- ✚ Process node requests (first CPU, then GPU stage)
- ✚ Process bricks requests (first CPU, then GPU stage)

This is where users have to define their own CPU and GPU producers.

The “request buffer” holds the nodes addresses that have been updated with the type of requests emitted during the rendering phase.



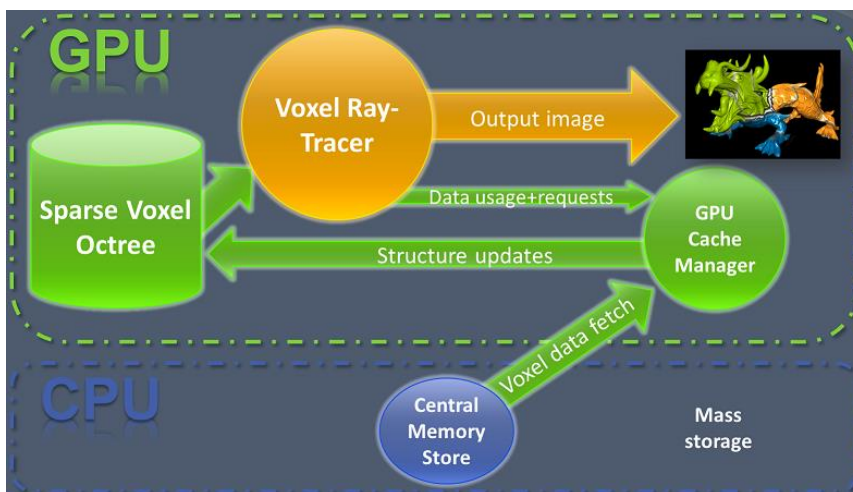
14. Data Production Management

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

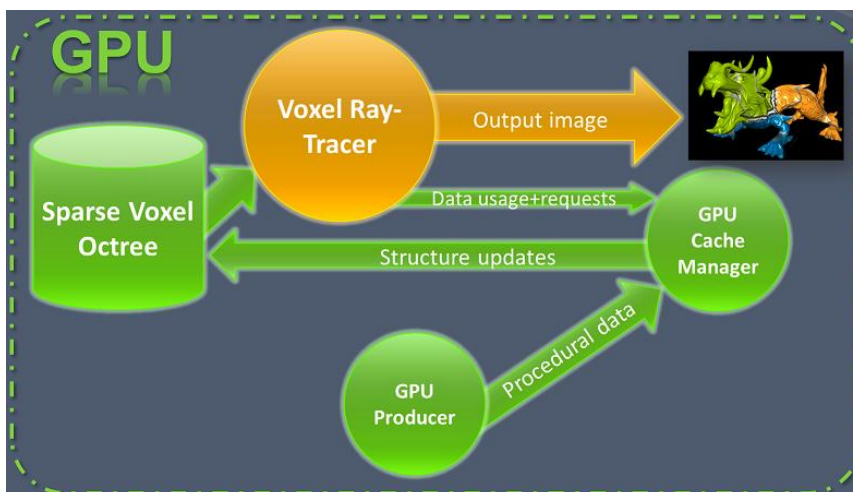
There is two different schemes available:

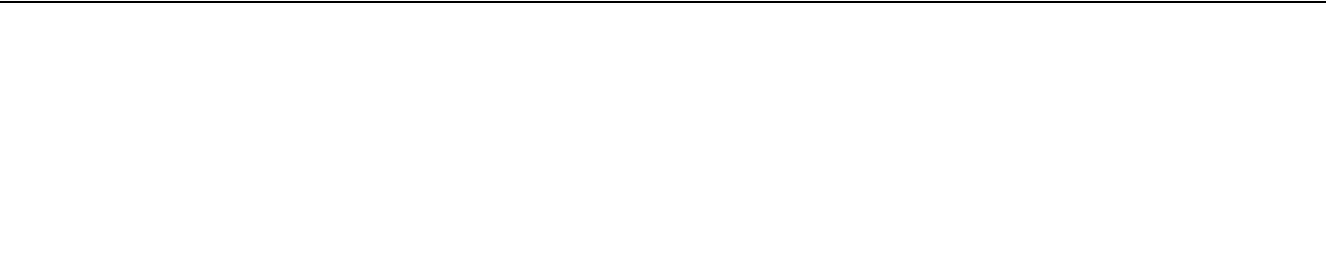
- quality before
- or performances versus quality trade-off

Pipeline of a CPU producer loading file on disk.



Pipeline of a GPU producer.





Node pool
Elements are grouped in node tiles. Each node points to its children.
For instance, for an octree, a node tile is a group of 8 children.

1. Using automatic references

2.1 Figure

If you need to refer to a figure, first select Figure, then Reference/Insert Legend (numbering is automatic).
To refer to this figure, Reference/Cross reference (Renvoi), then select Figure, Text and number (Texte et numéro de légende), and then select the figure you want to refer to. FIGURE 1 refers to the figure below. If you need to have a lower case reference to this figure, you can edit the field (basculer les codes de champs) and add *Lower to the reference, the you will obtain a reference like: figure 1



FIGURE 1 : This is a figure

2.2 Tables

Tables work the same way as figures except that you must select Table instead of Figure in the Reference menu. Table 1 is an automatic cross reference example.

Table 1. A very nice table.

	<i>tota</i>	<i>rea</i>	<i>writ</i>	<i>tota</i>	<i>rea</i>	<i>writ</i>	<i>tota</i>	<i>rea</i>	<i>writ</i>
	/		é	/	é	é	/	é	é
cust	4.5	4.1	0.4	9.0	7.5	1.4	21.	17.	3.9
o	é	é	é	é	é	é	é	é	é
m	é	é	é	é	é	é	é	é	é
e	é	é	é	é	é	é	é	é	é
r	é	é	é	é	é	é	é	é	é
s	é	é	é	é	é	é	é	é	é

addr e s s	1.2	1.1	0.1	2.8	2.6	0.2	7.7	7.4	0.3
Tota l									

2.3 Bibliography

You can use automatic numbering for your bibliography. You must assign a specific bookmark (Signet) to each bibliographic reference and then refer to this bookmark.

If you select the number assigned to the first bibliographic reference in section 0 (that is to say number 1 between the []), then go to Insert/Bookmark and give a name (note that spaces are not allowed in bookmark names). To insert a cross-reference, Insert/Reference/Cross Reference, then select Bookmark and select the right bookmark. Note that you have to add the [] yourself in the reference, just the numbering is automatic.

Here is a single reference example [1], and here is a multiple bibliographic reference [1, 2].

Conclusion

Reference updates are not always automatic, and you may have to force a refresh. To do so, select the whole document (Ctrl+A) and press F9 (or find the equivalent in the menu). Printing the document also usually refresh all fields (including header/footer).

If you need help about this Word model, you can contact anne.bres@inria.fr.

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



**RESEARCH CENTRE
GRENOBLE - RHÔNE-ALPES**

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

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