A voxel-based rendering pipeline for efficient exploration of large and detailed scenes

Pascal GUEHL
R&D Engineer

Videos Games
Special Effects
Scientific Visualization

Between Research & Industry

http://gigavoxels.inrialpes.fr/
What is all about?

R&D project about real-time exploration of
- large and detailed objects/scenes
- eventually generated on the fly
- visually realistic

Target audience
- Video games
- Special effects
- Special case: scientific visualization

voxel based medical dataset generated from CT scan (2048x2048x2048, 32GB on disc)
Goal

Present our implementation of the « GigaVoxels » technology

[ 2 ] - Show fonctionnalities through examples of the SDK
[ 3 ] - A survey : explain how to program
**Exploratory Research phase**
- PhD Thesis: « GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes »
- Team: INRIA/CNRS/LJK (Cyril Crassin, Fabrice Neyret)
- Based on an OpenGL / GLSL + CUDA prototype

**Engineering phase**
- Started in 2011: clean, maintain, add fonctionnalities, ...
- SDK (tutorials), doc, tools, ...

**Partnership**
- RSA Cosmos: planetariums [funded by French ANR, 4 years research project]
Website

- image gallery
- videos
- publications
- documentation
- source code (not yet)
- contact

http://gigavoxels.inrialpes.fr/
Key Ideas

[x] Rendering only dependent on what is visible
  - ray-tracing approach

[x] Load only needed data, at the needed resolution
  - occlusion + LOD (level of details)
  - ray-guided streaming

[x] Reuse loaded data as much as possible
  - cache mechanism on GPU

[x] Minimize computation, minimize memory transfers
Tree Data Management (space partitioning) to store and organize data (octree or generalized N3-tree, + SDK example kd-tree)

Cache System on GPU: LRU mechanism (least recently used) (to get temporal coherency)

Data Production Management: on host, GPU, or hybrid mode
Goal: produced data are kept in cache on GPU

Visit algorithm: traverse your data (loaded in cache) as could be done for rendering

Renderer (hierarchical volume ray-casting, cone tracing, emission of requests, brick marching)
Data Structure

**Generalized N\(_3\)-tree** (octree)
- Space partitioning
- Empty space compaction

**Bricks of voxels**
- Linked by octree nodes
- Store opacity, color, normal,…

“Node pointer based” data structure

Node Texel
- Nodes Tile pointer
- Constant color
- Max subdivision flag
- Data type flag
- Address

N3-Tree

Mip-Map Pyramid with stored bricks
GigaVoxels Pipeline

- **GPU**
  - Sparse Voxel Octree
  - Voxel Ray-Tracer
  - Output image
  - Data usage + requests
  - Structure updates
  - GPU Cache Manager
  - Voxel data fetch

- **CPU**
  - Central Memory Store
  - Mass storage
GigaVoxels Pipeline + data structure

- unified data structure (geometry + texture)
- load only needed data at needed resolution
- smooth transitions and continuously reveals details
- Handle semi-transparent objects
- Alias free filtered images (cone tracing)
PART 2 - Fonctionnalities through SDK examples
Data loading

Data streaming
Load data « on demand ». Two modes :
- level by level (from coarse to fine) [ speed trade-off ]
- directly max level of resolution [ quality before ]

File format
By level of resolution and by data types
- 1 - nodes (spatial structure)
- 2 - bricks of voxels
no header, no meta-file...
Import your own data

**Custom file importer**

RAW data [scientific visualization]

- apply transfer function
- threshold data
- clipping plane [region of interest]

Aneurism

Skull

Foot + threshold
Mandelbulb

Goal: generate environments

Fractal object: 3D extension of Mandelbrot set

\[
\langle x, y, z \rangle^n = r^n \langle \cos(n\theta) \cos(n\phi), \sin(n\theta) \cos(n\phi), \sin(n\phi) \rangle
\]

\[
\begin{align*}
  r &= \sqrt{x^2 + y^2 + z^2} \\
  \theta &= \arctan(y/x) \\
  \phi &= \arctan(z/\sqrt{x^2 + y^2}) = \arcsin(z/r)
\end{align*}
\]

pour la nième puissance du nombre hypercomplexes 3D. Les points sont calculés par itération de \(z \mapsto z^n + c\) où \(z\) et \(c\) sont des nombres hypercomplexes dans un espace de dimension 3 et \(z \mapsto z^n\) l’application définie ci-dessus. Ici \(n = 8\).
Procedural data generation

Mandelbulb
Noise - Hypertextures

Goals
Add details on 3D models
- smooth transitions and continuously reveals details
- alias free filtered images
- handle parallax effect

Theory
- Ken Perlin
- book: « Texturing and Modeling, a procedural approach », chapter 12
Noise

Input

3D model : Signed Distance Field

- Noise : applied on distance and normal
- Transfer Function : convert distance to RGBA color
Noise : optimisation(s)

Idea : (use cache)

Mimic the LOD mechanism that continuously add details :

⇒ Re-use noise computation at previous resolution level

- noise : sum of octaves (frequencies)
- compute only one octave by level of resolution
- store them in GPU cache
- reproduce the sum :

\[ F(\text{level } N) = \text{currentNoise}(\text{level } N) + \text{previousNoise}(\text{level } N-1) \]

Finally : replace “noise computation” by “data fetch in cache”
Hypertexture (shape + solid texturing)

- $F(x) = 0$: outside the object
- $F(x) = 1$: strictly inside the object
- $0 < F(x) < 1$: inside a fuzzy region

Idea: add details on armor soldier

- Distance function: sphere
- Thickness: fuzzy region (Perlin noise)
- Current work: optimizations
  - Stop adding noise components if density is outside
Depth of Field

Goal
Add more realism to images
- mimic camera lens (focus plane)
- double lens cone
**Goal**: Simulate environments

**Instanciation and recursion**
- node pointers based system
Voxel data synthesis

Menger/Serpinski Sponge
OpenGL interoperability

Mix triangles and voxels

- integration with traditional CG rasterized scenes (or compositing)
- renderer takes color and depth buffer as input and updates them with voxels
Voxel-based Instancing

**Goal** (not yet, in study/progress)

Have several GigaVoxels entities (forest)

- Need proxy geometry
- Render projected 2D BBox
- Use GLSL fragment shader
- (rays start + direction)
Goal

Environment to study, test, profile, debug...

- Kind of “generic” API: browsers, editors, etc...
- Tools: transfer function, performance monitor, etc...
- 3D models importer (+ GLSL shaders editor)
- Futur hope:
  - CUDA functions editor
  - “recompile on the fly”
Goal

Plugin « GigaVoxels Pipeline » : dynamic library (.dll, .so)

=> kind of « effect »

- Idea : plug this “node” in a scene graph à la OpenSceneGraph (see the “ppu” project) or Ogre3D
- Not yet finished : it has been “polluted” with Qt to have USER custom editors.
Pre-process step

Future work:
- optimize code
- smooth resulting data
- idea: voxelize data
  “on the fly” in the producers
Use events to record time between kernel functions
Special case: record “device” functions (1 thread per pixel)
Outils : Performance Monitor

Record device function for Rendering kernel (1 thread per pixel), then compute average time on all pixels

```c
#ifdef WIN32
    typedef unsigned __int64 uint64;
#else
    typedef unsigned long long uint64;
#endif

__device__ uint64 getClock()
{
    uint64 result;
    // Using inline PTX assembly in CUDA.
    // Target ISA Notes : %clock64 requires sm_20 or higher.
    //
    // The constraint on output is "=l" :
    // - '=' modifier specified that the register is written to
    // - 'l' modifier refers to the register type "u64 reg"
    asm volatile ("mov.u64 %0,%clock64;" : "=l"(result) : :);
    return result;
}
```
French ANR Research Project
RSA Cosmos : planetariums [2010-2014]

“Real-Time and Interactive Galaxy for Edutainment”

I. Visualize static galaxies
II. Data amplification (procedural generation)
III. Animate galaxies

Optimisations
- use OpenGL billboards to stop rays (VBO)
- multi-GPUs ?
- time budget ? , priority on bricks ?
PART 3 - The library

- [ 1 ] - Core library
- [ 2 ] - User custom API
The library

Goal
- Capitalize on the knowledge of the team
- Continue research on the subject
- Hide the complexity of the underlying « core » library (cache mechanism, data structure management, etc...)
- Give USER access to customize the « data production » and the « shader »
The library

**Technologies**

- C++ (template)
- GPU Computing: CUDA + libraries (cudpp / thrust)
- CMake
- IHM: Qt
- Viewer: QGLViewer
- 3D: OpenGL (GLSL)
- Others: Loki, Assimp, CImg, ImageMagick
- OS: Windows (7), Linux (Ubuntu) + 32/64 bits
- Requirements: Cuda 4.x and Compute Capability SM at least 2.0
[1] - Common API: « hidden » generic mechanisms

- **Tree Data Management** (space partitioning) to store and organize data (octree or generalized N3-tree, + SDK example kd-tree)
- **Cache System on GPU**: LRU mechanism (least recently used) (to get temporal coherency)
- **Data Production Management**: on host, GPU, or hybrid mode
  Goal: produced data are kept in cache on GPU
- **Visit algorithm**: traverse your data (loaded in cache) as could be done for rendering
- **Renderer**: hierarchical volume ray-casting, cone tracing, emission of requests, brick marching
[ 2 ] – USER access

USER Customizable API

- **Producer** (load or produce our own data)
  - used during Data Production Management

- **Shader** (write your custom shader)
  - used during Visit algorithm and Rendering
Data Production Management:

- Nodes subdivision (data refinement)
- Load or compute bricks of voxels (fill data)

USER has to write:

- a HOST producer
- and its associated GPU producer
Sequence Diagram of a frame:

1. Visit Data
2. Handle Requests
   - A. Subdivide Nodes
   - B. Fill Voxels Data

Kernel (2D) Ray-Casting
1 thread/pixel

CUSTOM shader

Tree Traversal

Ray Marching

Emit Requests (if needed)

CUSTOM producer

HOST → GPU
1
1

2
2
Sequence

Frame 1
Produce Nodes
  Produce Bricks

Frame 2
Produce Nodes
  Produce Bricks

... (Repeat the pattern)

Frame N
Produce Nodes
  Produce Bricks
Nodes subdivision:

- KERNEL: 1 bloc/node and 1 thread/child_node
- Each node has to say what’s inside each of its child
- INPUT: localization info of current node (LOD depth and spatial index pos)
- INPUT: address in “node cache” where to write new child nodes
- Test if it is in a sphere (analytically)
- Write nodes in cache

Ex: sphere on GPU
Ex: sphere on GPU

Nodes subdivision:

WORLD: sphere

FLAG Children:
- EMPTY
- DATA inside
- MAX RESOLUTION reached

Write result in cache:
- Next frame, renderer will ask for the node data production (brick of voxels)

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

(same principe as for nodes)

- KERNEL: 1 bloc/brick and thread is as you want
- USER is in 1 brick and has to populate its voxels
- INPUT: localization info of current node (LOD depth and spatial index pos)
- INPUT: address in “data cache” where to write new voxels
- Write voxels in cache

Ex: sphere on GPU
Ex: sphere on GPU

Reminder: SHADER management

1. Visit Data
   - KERNEL (2D) Ray-Casting
   - Tree Traversal
   - CUSTOM shader
   - Emit Requests (if needed)
   - 1 thread/pixel

2. Handle Requests
   - Subdivide Nodes
     - HOST ➔ GPU
   - CUSTOM producer
   - Fill Voxels Data
     - HOST ➔ GPU
**Ex: sphere on GPU**

**SHADER:**
- **INPUT:** “data sampler” and current position
- Sample data, retrieve USER defined channel (color, normal, etc...) and accumulate color along the current ray

```cpp
// Retrieve first channel element : color
__device__ void ShaderKernel::run( const SamplerType& brickSampler, ... )
{
    float4 color = brickSampler.getValue<0>( coneAperture ); // channel 0 is color
    If ( color.w > 0.0f ) {_accumulatedColor = _accumulatedColor + ( 1.0f - _accumulatedColor.w ) * color; } }
```
**SHADER:**

- CUSTOM api provide “other entry points”:
- `getConeAperture()` [modify cone aperture]
- `getStopCritrion()` [ex: `accumulatedColor.w >= 0.98f`]
- `getDescentCriterion()` [used during “visit” traversal]
- ...

GigaVoxels Sequence Diagram
Tips and tricks:

- Template meta-programmation: curiously recursive template pattern (to avoid polyphormism on GPU)

This technique achieves a similar effect to the use of virtual functions, without the costs (and some flexibility) of dynamic polymorphism.

- Data cache: 3D texture to read, and bind with surfaces to write data

- Loki: use to access different textures types

- Cudpp/thrust: cache mechanism on GPU (stream compaction to get “used” and “non-used” elements at current frame)

- Write Z-buffer: slow => idea => use color and do conversion
Mix triangles and voxels:
- integration with traditional CG rasterized scenes (or compositing)
- renderer takes color and depth buffer as input and updates them with voxels
- Opaque objects rendered first (depth buffer) to skip occluded parts
- GigaVoxels registers « textures, renderBuffer, PBO, ... » with optimized flags
  (see GTC 2013 : S3070 – « Part 1 – Configuring, Programming and Debugging Applications for Multi-GPUs » from Wil Braithwaite [NVidia]
  cudaGraphicsRegisterFlagsReadOnly, cudaGraphicsRegisterFlagsWriteDiscard, ... )
OpenGL interoperability

Mix triangles and voxels

Simple mode : [ no input ] + [ only write color ]

// [ 1 ] - Init texture (with window size)
glGenTextures( 1, &colorTex );
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, colorTex );
gTexImage2D( GL_TEXTURE_RECTANGLE_EXT, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL );

// [ 2 ] - Register texture in WRITE ONLY slot in GigaVoxels
renderer->connect( Gv::eColorWriteSlot, colorTex, GL_TEXTURE_RECTANGLE_EXT );

// Launch GigaVoxels rendering engine
renderer->render( /*OpenGL matrices*/ );

// Draw a full screen textured quad
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, colorTex );
gBegin( GL_QUADS );
...
OpenGL interoperability

Current work

[ 1 ] - Proxy geometry
“rasterized (approximate) geometry” containing non-empty areas of volume, providing starting and exit point for each ray, speeding-up the skipping of empty spaces
- Help skip empty regions
- Can add volumetric details on a triangle mesh

- Would like to have demo with forest
OpenGL interoperability

Current work: based on a galaxy visualization research project

Number of stars represented by points (VBO) can be very huge:

[1] Sphere Ray-Tracing (change representation)
- When bricks are near empty, generate spheres coordinates in « producer »
- Replace brick-marching by Ray-Tracing at rendering phase (fetch positions)

[2] VBO generation
- add details by generating stars (OpenGL vbo points) in a GigaVoxels « producer »
- dump result in VBO dynamically (only visible nodes)
- => kind of view frustum culling
... the futur...

**Ideas**

- Working on performance (Visual Profiler, NSight)
  - Handle time budget for a frame, priority on bricks
- Use CUDA 5: linker (recompile USER functions on the fly?)
- Store points inside voxel (VBO generation according to visibility)
  - Geometry instancing (forest)
  - Voxel animation (with shellmaps)

...
User / developer guide

Library / SDK content
- Core library documented
- SDK tutorials fully documented
- Developer guide: soon
  - kind of “behind the scene”): technical implementations detailed
Thanks

- Fabrice Neyret (CNRS, INRIA, LJK), M. Armand, E. Heitz, E. Eisemann, ...

- Cyril Crassin (NVidia)

- Nvidia: for cards donation (Chandra Cheij)

- RSA Cosmos: for testing the library

- French Research ANR:

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

Thanks for your attention

http://gigavoxels.inrialpes.fr/