GigaVoxels/GigaSpace Tutorials - Nov.,12 2015

http://gigavoxels.imag.fr/

Table of Content

1. Introduction

1.1. What you will find here

DISCLAIMER :
This documentation is a shameful on-going work at early stage that has been stopped at the end of our funding (we were pretty late...). Sorry for that ! It's mostly a skeleton, but we hope it can still bring some help.

[Present a digest of the table of content :]

- installation guide for demos
- installation guide for developers
- packaging guide for wizards
- Tutos:
    - using GigaVoxels demos
    - black-box use of GigaVoxels [picking existing funcs, maybe calling in OpenGL]
    - white-box use of GigaVoxels: overview [simple sphere: datatype+producer+render(4steps)]
    - making a data producer function [only and all pure voxel producers. GPU+CPU]
    - making a renderer function [material color, on the fly LUT, on the fly procedural]
    - easy rendering effects with GigaVoxels [soft shadows, DoF]
    - pre-post and interoperability with OpenGL [clipping plane / shape, then all basic interops]
    - using pure OpenGL GigaVoxels renderer (bye bye Cuda) [openGl renderer examples]
    - GigaVoxels as effects-booster for OpenGL [complex shadows/reflect/refract to OpenGL]
    - beyond SVO: alternatives to octrees [27-trees, graph-tree, N3, kd]
    - beyond SVO: alternatives to voxels [points, mesh]
    - beyond rendering [point clipper/sorter, collisions]
    - extra: tuning strategy flags
    - extra: animation
- Extra demos
- Notes for the future

Note that even if some tuto examples could be directly used as tools, or reused and modified as a base for your applications, none of the provided examples pretend to be programmed optimally: some are kept naive for pedagogical simplicity, while some others are quickly made "proof of concept" demos - possibly ugly and buggy, done by various collaborators at various stage of the project (BTW, you will find more demos in the SDK than in the precompiled set). So all this material are provided "as it is"; the spirit is to use them as illustrations or kickstand starters before writing you own for each aspect !

1.2. What is *GigaVoxels / GigaSpace* ?

[notes about links have disappeared at copy-paste. Check back in the GoogleDoc]

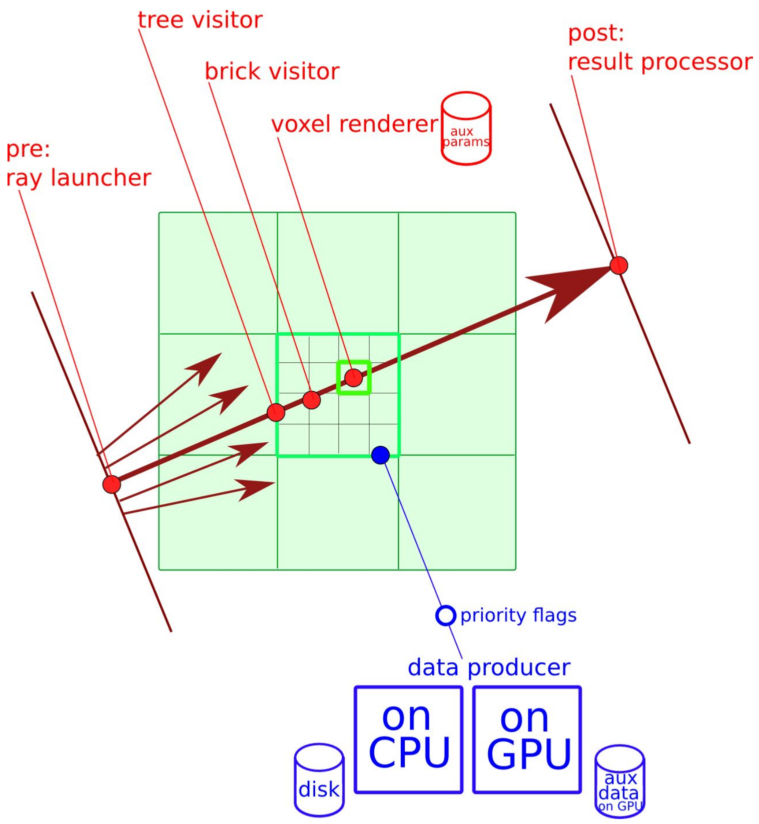
1.2.1. GigaVoxels

GigaVoxels is an open library made for GPU-based real-time quality rendering of very detailed and wide objects and scenes (encoded as SVO - Sparse Voxels Octree - but not necessarily fuzzy or transparent: see *History*). It can easily be mixed with ordinary OpenGL objects and scenes.
Its secret lies in *lazy evaluation*: chunks of data (i.e., bricks) 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 and aliasing if they are 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 the GigaVoxels renderer will call at the last moment to obtain data for a given cube of space at a given resolution. This function can have 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 generate it procedurally (e.g. volume Perlin noise), or convert the requested chunk of data[1] from another shape structure already present in GPU memory (e.g., a VBO mesh).

- You probably also want to add a user-defined voxel shader to tune the appearance (the view-dep BRDF, but probably also the material color rather than storing it in voxels) or even tweak the data on the fly. In facts, it is up to you to chose what should be generated (or pre-stored) on CPU, in the GPU producer, or on the fly 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 a density transfer function, or tune hypertexture parameters, this should be done in the voxel shader for immediate feedback - otherwise all the already produced bricks should be cancelled. But once these parameters 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 (multispectral, X-rays...), or not on a simple way (polarization, HDR…). GigaVoxels let you easily define the typing of voxel data. You can also customize pre and post-rendering tasks (e.g. to mix with a OpenGL scene), various early tests,

[1] 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 (they come naturally sorted), 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 (neighborhood is explicit), and soft-shadows as well as depth-of-field are easy to obtain - and indeed, they are faster than sharp images ! The implicit sorting permits the deferred loading or creation of the chunk that the GigaVoxels cache will keep for next frames. So, your producer might as complex as an iterative algorithm or even a local voxelizer of a large VBO - BTW, we do provide one.

hints, and flag for preferred behaviors (e.g., creation priority strategy in strict time-budget applications).
- The basic GigaVoxels relies on octree dynamic partitioning of space, which nodes corresponds to voxel bricks - if space is not empty there. But indeed, you can prefer another space partitioning 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 chunk, e.g., triangles, points, procedural parameters, internal data easing the update or subdivision to avoid a costly rebuilding from scratch...



[svg source]

1.2.2. GigaSpace

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 (indeed, three: a space-partitioning visitor, a chunk visitor, a point data accessor)
- A data producer called when missing data are encountered by the visitor.
Datatypes can be anything of your choice as long as chunk are fix size;
'space' can represents any variety as long as you know how to partition and visit it.

May be you got it: GigaVoxels and GigaSpace are one and same thing. GigaSpace is just the generalized way of seeing the tool. The basic use of GigaVoxels use octrees as tree, voxels bricks as data chunk, and volume cone-tracing[2] as visitor, so this naming is prefered for the SVO community. But we provide many other examples showing other choices, since the potential use is a lot larger than SVOs.

1.2.3. Cuda vs OpenGL vs GLSL

The GigaVoxels/GigaSpace data management is (now) a CUDA library - it used to be on CPU in our early stages.
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.
Still, 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 (and possibly, shadow-maps, light-maps and env-maps) so that fragments of both sources are correctly integrated. Similarly, GigaVoxels can return its color and depth buffers to OpenGL, and so on with possibly other loops. Note that hidden voxels won't render at all: interoperability keeps our gold rule "pay only for what you really see".
- *Volume clipping:*
It is easy to tell GigaVoxels to render only behind or in front of a depth-buffer, which makes interactive volume clipping trivial for volume visualization.
- *Volumetric objects instances and volumetric material inside meshes*:
To let OpenGL display cloned transformed instances of a given volume (e.g., to render a forest from a volumetric tree), or display shapes carved into some openwork 3D material, you can use an openGL bounding-box or a proxy-geometry, use their object-view matrix to rotate the volume view-angle accordingly, possibly provide a mask of visible fragments, and tell GigaVoxels to render only from the front depth and (optionally) stopping before the rear depth.
- *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, as a cone-tracer 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 classical GLSL surface shader, you can easily launch the reflected, refracted or shadow rays in gigaVoxels within a (probably crude) voxelized version of the scene. Real-time global illumination was achieved in this spirit in the GI-Voxels work.

[2] Volume cone-tracing is basically volume ray-marching with 3D MIPmapping.

1.2.4. 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) with *Sparse Voxel Octrees* (SVO), and for high quality rendering by Jim Kajiya with Volumetric Textures, first dedicated to furry Teddy bears, then generalized by Fabrice Neyret
- The idea of differential 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 demand production, by Sylvain Lefebvre and al .
- Cyril Crassin connected all these ideas together during his PhD to found the basis of the GigaVoxels system .
- Note that Proland, the quality real-time all-scale Earth renderer (also from our team), is based on the same spirit for its multi-scale 2D chunks of terrain.

2. Installation guides

For now, check the GigaVoxels web site ,
or the folder Documents plus the files REAME / HOWTO scattered in the source distribution.

2.1. Installation guide for demos

[copy-paste from distrib the appropriate README/HOWTO here]

2.2. Installation guide for developers

[copy-paste from distrib the appropriate README/HOWTO here]

2.3. Packaging guide for wizards

[copy-paste from distrib the appropriate README/HOWTO here]

3. First steps

3.1. Playing GigaVoxels demos

Two possibilities :

- browsing demos from our demo player:
  launch `gigavoxels-viewer` / `GvViewer` and load demos from the file menu (various tabs let you access demo or GigaVoxels parameters)

- launching standalone demos (for most demos)
  in the spirit of Cuda devkit demo, go to the target directory and launch the binary.

In the standalone demos, you will find only the minimal GUI strictly necessary for the demo.

In the player, the GUI let you play with generic and specific parameters, and also provide you with many exploration modes and informations that help you testing and understanding what happens.
For instance you can freeze the GigaVoxels cache then move the camera to observe what was stored, or visualize informations such as empty-space and age of bricks.

[add a complete presentation of the demos player ?]

Demos survival guide

(see below for GvViewer vs standalone demos specific issues)

- By default auto-subdivision of volumes is off, and max level is ultra-low (sorry for that...). -> see below how to tune them.
- GPU memory max usage quota is set by default at a small value to illustrate the caching/production behavior (red screen means cache full). Requested size seems hard-coded in each demo: you must recompile to change.
- Some key shortcuts are useful or crucial:
  - `h:` QGLviewer help panel (pure GvViewer keys are not listed; sorry for that...)
  - `+/-:` force subdivision level.
  - `t:` display octree and node status as superimposed wireframe.
  - `d:` freeze cache (then camera can explore it).
  - `c:` clear cache (force full rebuild).
  - `f:` display FPS.
  - `i,u:` perfmon toggles.
  - not that you can settle and play a camera path, capture image... see Help.
- Some demos are based on specially prepared data files, and will crash on others.
- Note that some tutos differ by the programming approach more than by what you see (typically the “simple sphere” meme). See dedicated sections in the doc, or in sources.
- Only the polished demos got to the binary set. SDK sources contain 18 more.
- Be sure your GPU preferences are configured so that a stucked CUDA code get recoverable.
  For Windows, see here.

GvViewer : survival guide

- To start, load a demo from file menu / open.
- On some OS the renderer tab (middle) width is 0: enlarge it with mouse.
- To activate shaders and tuning tabs, do click on the demo name in "pipeline browser" (top left).
- Take care of side effects: camera and lights position and direction from the last demo you watch might look away from the new scene :-)
- In case of doubt of other side effect, just quit and come back ;-)
- By default auto-subdivision of volumes is off, and max level is ultra-low (sorry for that...). -> Go to right tabs / renderer, set high max depth and clic dynamics update. Or use `+/-`.
- Various other quality/performance tunings are available: in right tabs / renderer, see request strategy and time budget.
- Some extra key shortcuts might be available beyond those mentioned in the previous section:
    - `l,r:` tune or reset light (in `GvSignedDistanceFieldVoxelization`).

Standalone demos & tutos : survival guide

- By default auto-subdivision of volumes is off, and max level is ultra-low (sorry for that...). -> tune max subdiv with `+/-`.
- Some extra key shortcuts might be available beyond those mentioned in the previous section:
    - `l,r:` tune or reset light.
    - `e:` open transfer function editor.
    - `pageup/down,home/end,insert/del` : demo-dependant tunings (in `DepthOfField, ClippingPlane, SimpleTriangles, MandelbrotSet`).
    - digits: demo-dependant tunings (in `SmartPerlinNoise`).
- Note that Voxelizer demo needs installation of ImageMagick.

3.2. Black-box use of GigaVoxels

[Alternative: be more basic, just how to blindly glue to API to (some part of) demos in your code and see result in Cuda or OpenGL. Then, keep the following below for "white box use" section]

You need to choose a *spacetype* class, a *datatype* class, a *producer*, a *ray-launcher*, a *renderer* (all these are detailed in the following sections).
Or if you start with an existing demo, to keep or change the ones that please you.
Note that most renderers we provide in the tutos implement the *tree visitor, brick visitor* and local treatment (*voxel renderer*) all in once.

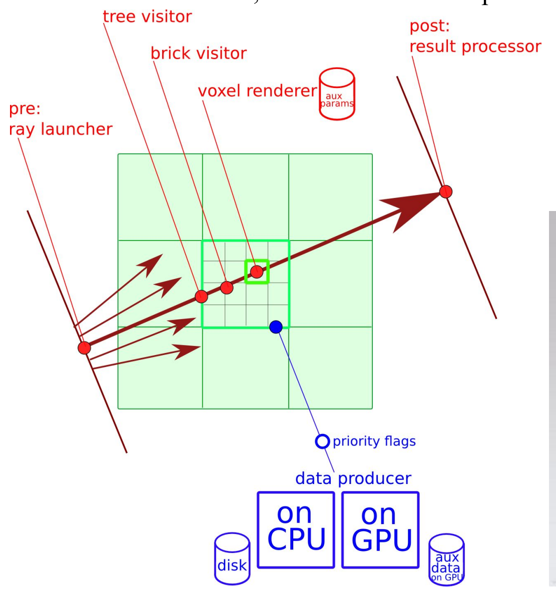
In Cuda:
[ToDo: "tune the settings this way in this file"], then launch the viewer ["that way"]
Simple sphere example: ...

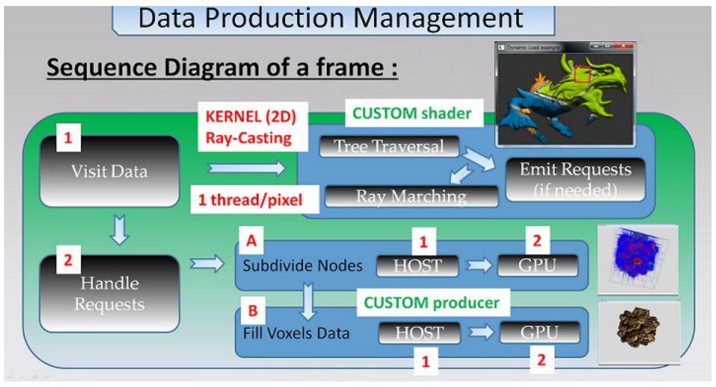
In OpenGL:
[Toso: "do that way"]
Simple sphere example: ...

3.3. White-box use of GigaVoxels: overview

[here, we explain more the API, based on detailing the first *simple sphere* example.]

As a reminder, here are the various pieces involved in GigaVoxels:

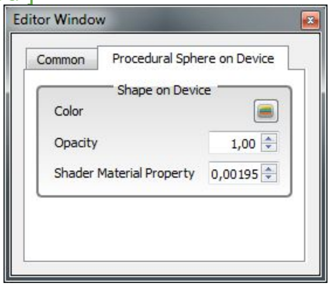




[svg source]

Not all elements need to be present. For instance there might be no CPU-side producer, or even no producer at all. The ray launcher can be the canonical volume box marcher, with no post-process.

In this example, "color" and "normal"are generated by the GPU Producer on demand. The Renderer is a classical ray-marcher, and is done brick by brick. [detail the coding and API involved]


Editor Window

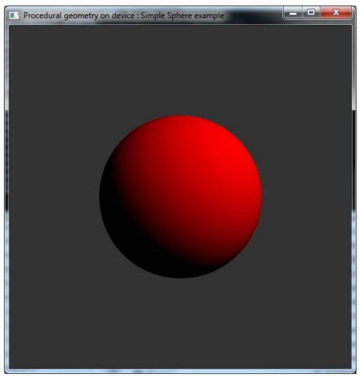
Common | Procedural Sphere on Device

Shape on Device

Color

Opacity 1,00

Shader Material Property 0,00195



4. Choosing a *data type*, a *node type*, a *brick type*

4.1. Data type

The *simple sphere* example above already illustrates some of the choice you have concerning the type of local data to store and compute with in voxels:

- Storing only the density ? how ? (opacity, density, coverage, optical depth or something more cheesy ? scale-independent or not ?)
- Or storing also the gradients (i.e., normals) to avoid costly and inaccurate finite differences calculation at rendering time ? But this requires more memory so the cache might forget and request recalculation more often : the arbitration is your's !
  Our *simple sphere* example stores the normals, while our *Mandelbulb* example compute finite differences in the renderer.
- Storing also the colors ? well, specular is view-dependant and thus can only be computed in renderer, but for complex appearance some parts could be precomputed and stored. Or you might prefer to store less and compute the whole appearance in the renderer - plus fetching more data can be costly on GPU.
- At low level, do you want floats, doubles, halves ? log-like HDR encoding or flat ? RGB or some color space, or extra spectrum samples ?
- All these are for voxels, of the usual kind. But you might want to store something else like 3D points (see section N.N), triangles (see section N.N), or your fancy custom representation.

To settle a data-type, define `[ instructions ]`.
Then tell your producer, renderer, ray launcher to use is by doing `[ instructions ]`

4.2. Node type

What to store in your tree nodes ?
If you want an octree, it is simply 8 nodes pointers, i.e., a 2x2x2 brick of node ids.

4.3. Brick type

How do you want your bricks ? Assuming you want to store voxels, how many of them per brick ?
Gigavoxels use trees of bricks of voxels rather than trees of voxels to benefit interpolation in space and LOD and highly coherent local access.
Voxels tracing is algorithmically efficient and cache-friendly and the larger the bricks the shallowest the tree so it can be interesting to have a bunch of voxels in bricks. But it wastes some space as empty voxels or inside voxels, and wasting is both in memory and marching - empty voxels are computational evil for volume ray-tracing.

Also you probably want a border duplicating the brick neighborhood for efficient seamless interpolation through bricks, likely on every sides for ease and even numbers - but one per axe would be sufficient. It's probably interesting to amortize this extra storage on not-too-small bricks.

Still, some person do prefer 2x2x2 bricks, others 8x8x8 or more, and optimal is probably application dependant. (But more work on how to optimize tunings would be very welcomed :-)).

5. Making a *Data Producer*

The role of the producer is to provide the scene content on demand for a given region of space at a given resolution, in a form usable by your renderer. This data is requested by the renderer when it needs it, and will be kept in an LRU cache for next frames. So at each frame only a few bricks of small size (e.g. 10x10x10 voxels) should be produced.

Attention: for correct and efficient spatial interpolation, voxels bricks generally include a margin duplicating neighbor data. An easy error is to forget accounting for the margin !

Also, keep in mind that voxels do not correspond to a fix scale: If you chose to store local opacity, then a filled voxel should not have the same value for a cubic meter or micrometer.

5.1. Full-GPU producers (procedural)

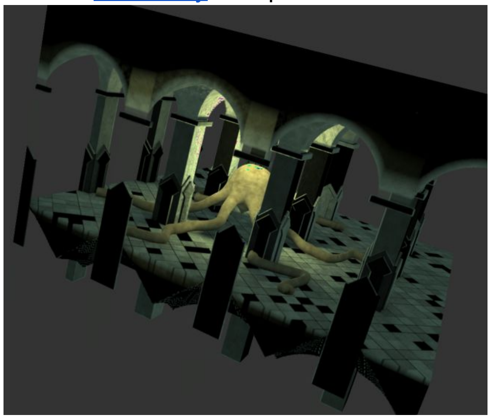
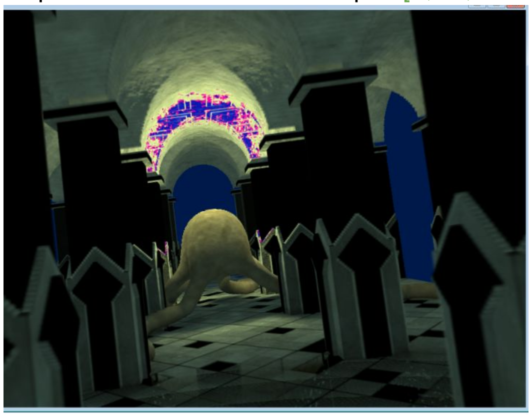
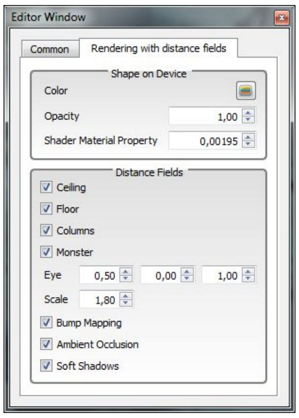
In some cases the shapes can be defined implicitly (e.g. equations, fractals) with no need of big raw data like meshes. So location and resolution is all you need to generate the data.
We provide 4 examples: a simple sphere, a Mandelbulb fractal, a Perlin noise hypertexture, a CSG scene.

5.1.1. From shape equation: demo "simple sphere 1 (GPU)"

The sphere equation tells us where is the matter and where is the empty space.
So we just have to loop on the brick voxels and settle either a 'in' or 'out' density.
In fact we can do it smarter, with anti-aliasing: for this we use for equation the signed distance to the sphere. If the voxel center is at distance more than a voxel radius inside or outside, the voxel is filled or empty. Otherwise we can estimate the coverage of the voxel by linear interpolation.

5.1.2. From distance field: demo "Procedural Constructive Solid Geometry"

Indeed, all shapes for which a distance function can be derived, can use the same scheme than for the *simple sphere* above.
A lot of basic shapes like spheres, cylinders, boxes, torus can be defined that way, but also their combination: union (min of distances), intersection (max of distances), subtraction (intersection with the negated distance)… Inigo Quilez provide a full tutorial here, which can be tested interactively in shadertoy. We provide one of his complexe demo scene as example: [`(Gv)SliseSix`]

Editor Window

Common | Rendering with distance fields

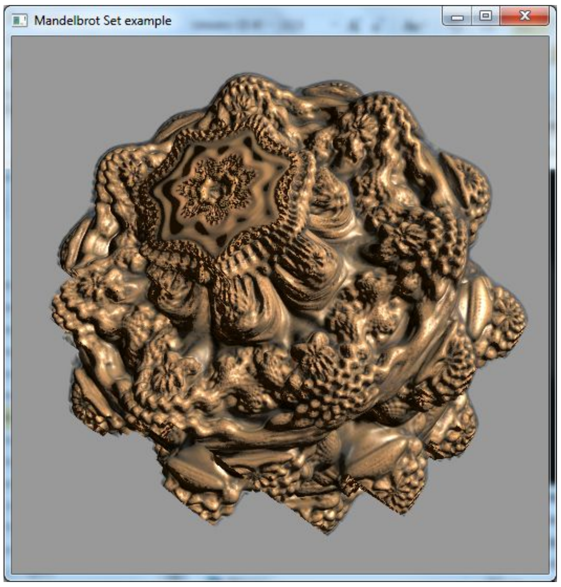
Shape on Device
Color
Opacity 1,00
Shader Material Property 0,00195

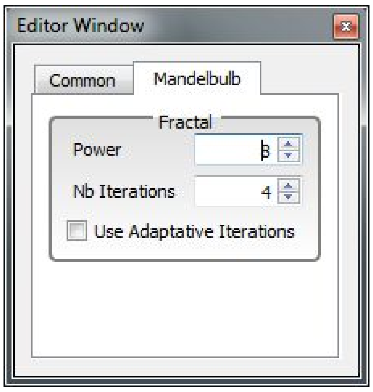
Distance Fields
☑ Ceiling
☑ Floor
☑ Columns
☑ Monster
Eye 0,50 0,00 1,00
Scale 1,80
☑ Bump Mapping
☑ Ambient Occlusion
☑ Soft Shadows

In this example, "color" are generated by the GPU Producer on-demand. Shading is also done at Production time (shadow, noise, etc…). Note that an editor window enables you to modify scene elements (invalidating the whole scene data so that the visible part of the scene must be entirely re-produced). [detail the coding and API involved]

5.1.3. From iterative distance field: demo “Mandelbulb fractal”

Distances functions might even be iterative or approximate. For instance interesting fractals such as the Mandelbrot set, or more recently, the Mandelbulb set, have known functions giving the distance to their shape. We can thus use it to create the object. We provide as example a Mandelbulb set: [strangely named `(Gv)MandelbrotSet`]





Power: 8
Nb Iterations: 4
Use Adaptative Iterations: ☐

In this example “color” are generated by the GPU Producer on-demand. The Voxel Renderer evaluates shading and normals (by central finite differences) on the fly during the brick-marching.
Note that an editor window enables you to modify object attributes (invalidating the whole scene data, so that the visible part of the scene must be entirely re-produced) . [detail the coding and API involved]

5.1.4. From noise function: demo “hypertexture 1”

Perlin noise is a fractal-based function of more practical use for CG. It is known as *hypertexture* when the target is volumetric densities enriching a base shape. Here, we have a potential function between 0 and 1 that we can directly sample at voxels centers.

Still, as a “texture” we need to apply it to a shape to be altered or amplified. Let first see the simplest example based on the simple sphere which interior is replaced by hypertexture (i.e., voxel density = sphere * 3D noise). [example 1: `GvHypertexture`]

Another classical use of the noise - corresponding to the historical hypertexture model - is to alter the distance function of the main shape. Since the initial bounding box must be respected (otherwise we can’t guess where the producer must be called), The main shape is considered to have a “blurry skin” of coverage ranging from 0 (transparent) to 1 (opaque) between 2 distance levels, e.g. distance=0 and distance= -skinThickess, which bounds the region to enrich with details. [example 2: `(Gv)Amplified*`]

We also propose more complex tutorials based on hypertextures:

- smart hypertextures [anchor: `(Gv)NoiseInheritance`], reusing computations done for coarser bricks when more resolution is requested;
- lazy hypertextures [anchor: `GvLazyHyperTexture, SmartPerlinNoise?`], trying to avoid useless calculations (like details in empty regions)
- interactive hypertextures [anchor: `GvDynamicHypertexture`], done on the fly in the renderer instead of in the producer to let the user dynamically tune the texture parameters.

[some of these demos are not in the precompiled set]

5.2. CPU- and mix producers

The producer can also be implemented on CPU side. Beside the fact that the application engine might better know what and how to produce, this side benefit for large memory, access to file system and network.

There is always a GPU side, that is at least an empty shell that receive the data produced on CPU and store it in the cache. But then it is easy to process the data on the way, having a mix-producer.

5.2.1. CPU producer: demo “simple sphere 2 (CPU)”

We provide another *Simpler Sphere* example where the same job than above [anchor] is done on the CPU. The GPU producer is then an empty shell streaming the content from CPU.
[detail the coding and API involved for CPU and GPU parts]

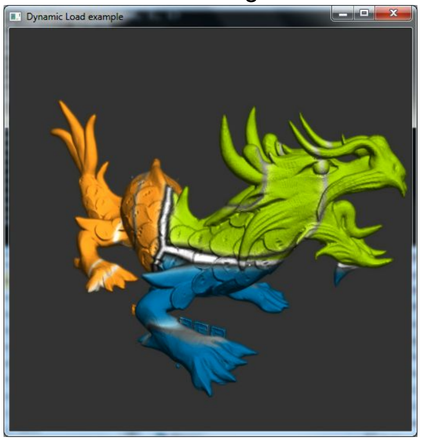
In this example, “color” (and eventually “normal”) are generated by the CPU Producer on-demand. Data is written to a buffer shared by the GPU (memory mapped). The GPU Producer is responsible to read data and write it in GigaSpace cache.

To speed transfer, user has the option to load all files data in a CPU buffer. This enables to use memcpy() instead of using stream IO API like fopen(), fread(), etc… [this assumes that previously we were loading from disk, which is not at all what is written !]

The Renderer is a classical ray-marcher, and is done brick by brick. If normals were not stored in 3D model files, a finite difference is used during brick-marching on-the-fly. [why “if” ? what does the demo ?]

5.2.2. CPU producer: demo “loaded volume (1)”

CPU-side producing is costly due to memory transfers from outside the GPU. It is generally done when the data can’t be generated, typically, for visualizing existing data such as volumes of density. CPU have a lot more memory than GPU and can thus be used as “cache for the GPU”. A basic optimisation is to pre-organize the volume data in bricks, at least to mark the empty areas as such - otherwise the producer should analyze the content to avoid filling the GigaVoxels cache with empty voxels, and it should better be done on CPU side to avoid transferring these as well.



In this example [(Gv) `DynamicLoad` ?], “color” and “normal” are generated by the CPU Producer on-demand. Data is written to a buffer shared by the GPU (memory mapped). The GPU Producer is responsible to read data and write them in GigaSpace cache.[detail the coding and API involved]

It might even be that huge data don’t even fit the CPU memory and is streamed from disk by the CPU producer - it makes sense if we only display a small part of it at a time, either because most is out of view, is masked by front parts, or is at coarse resolution due to distance or depth of field.
In such a case the data on disk should be pre-tiled into octree blocks pre-filtered at appropriate resolution, that the CPU producer just have to fetch.

In that purpose we provide a file format and a converter [prog name, explain how to use.]
[ADD here the demos with raw and with Dicom for now in 9.2 ? or just the producer part ?]

5.2.3. GPU/CPU mix producer: tuto “compressed transfert”

Some typical late treatments that can be done on GPU side is decoding, interpolating, decompressing the data sent by the CPU on a way trying to minimize the Bus transfer or the storage on CPU side. Note that being smart in a GPU massive parallelism frame is not always intuitive (tests or lesser coherency can make counterproductive “optimizations”).

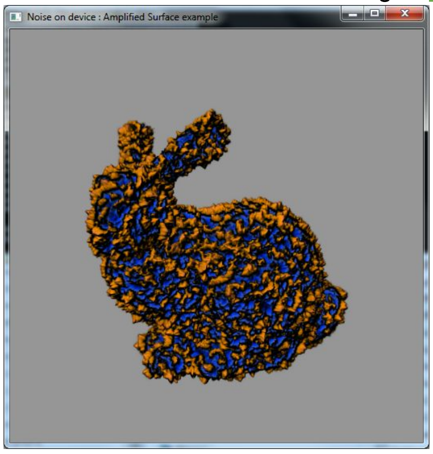
5.2.4. GPU/CPU mix producer: demo “amplified volumes and surfaces”

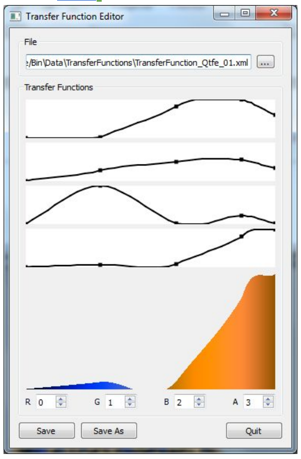
It is easy to let the GPU side of the producer modify the data provided by the CPU side, for instance by adding noise as in [example above]. The main interest is to let the CPU manage the shape at low resolution, and to let the GPU taking care of the high resolution details.

We provide a tuto where the volume provided by the CPU producer corresponds to the signed distance to a mesh surface (here it is precalculated and stored. In another tuto [anchor: 5.2.5] we show how to produce it directly in the CPU producer. In tuto [anchor: 5.2.5], we even do the voxelization directly in the GPU producer).

Then the GPU producer “amplifies” the data with hypertextures adding details by modifying the distance, as in [anchor to example before]. [`(Gv)Amplified*` . Which is what ?]
In the demo, we let the user tune parameter to shape the aspect built by the GPU producer: the color, but also the isovalue of the distance field that will correspond to the surface.
To access such parameters in the producers, [do that way].

Note that this is not the appropriate way to make a real interactive tuning tool, since the whole produced data has to be erased at each user change. This would make sense for static tuning. We deal with correct interactive tuning in [tuto coming later: 9.A]





In this example the Data consists of a pre-computed “signed distance field” voxel version of 3D mesh of a bunny (at each 3D points, we have the distance to surface + the normals stored in a float4 read-only texture). The CPU producer streams it to the associated GPU Producer. The GPU producer modulates this data on-demand using Perlin noise so as to “amplify” if to high-resolution hypertexture.

Note that the user can modify noise appearance by means of a LUT. In this example, the LUT is applied at Production time, so modifications makes existing data invalid and the Cache is thus cleared at each modification. In other examples [anchor] the LUT is applied on-the-fly in the Voxel Renderer, which allows huge modification while keeping all the cached data valid (at the price of extra rendering cost and some extra storage since all could-become-visible bricks must be stored).
“*Amplified Surface*” demo modifies surface, “*Amplified Volume*” add 3D volumetric effects using alpha.

5.2.5. GPU producer: from VBO mesh

5.2.5.1. Voxelization in producer: demo "simple bunny"

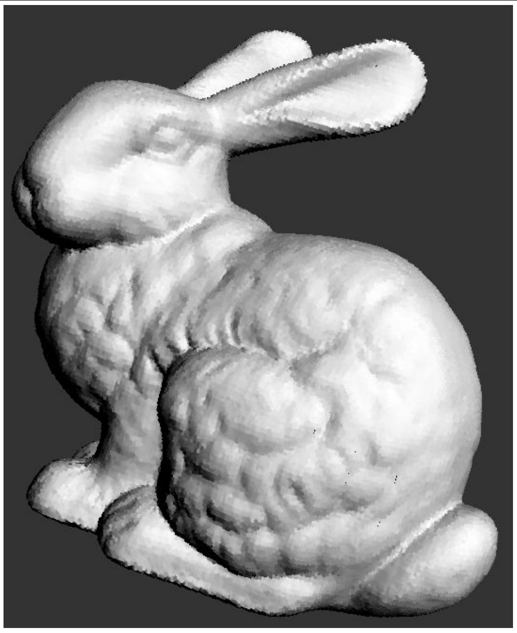
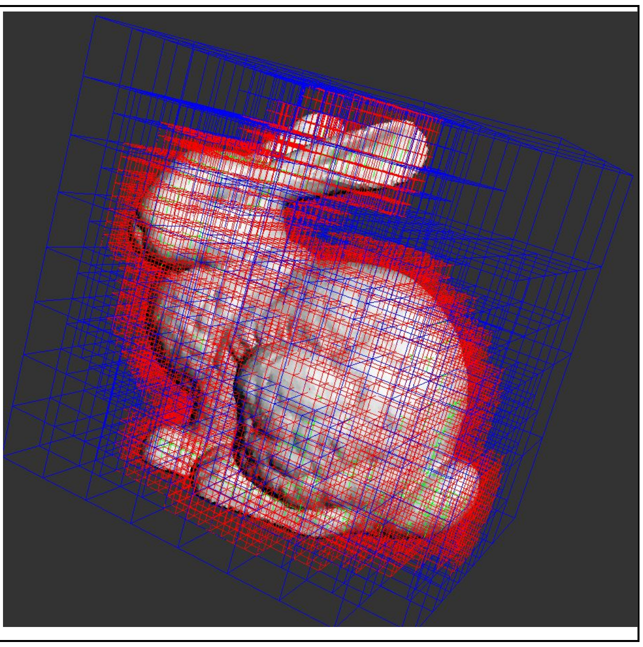
CPU bricks producer is not the only way to provide arbitrary shapes. Since meshes are compact data (when kept coarse), we can have VBO stored on the GPU.
So it is possible to produced mesh-amplification data entirely on GPU by letting the GPU producer process the voxelization of the mesh for the requested region of space (then possibly adding noise as above). This is what we propose in [this tuto].

In practice we don't want to voxelize the whole world, but only its not-empty part: in the object bounding box. In this example the mesh fit the displayed volume, otherwise we should first test whether the requested region intersect the bounding box to not generate bricks of empty voxels.

[`GvVowelizor, GvSignedDistanceFieldVoxelization`, …
which demo shows what ? where is the GPU-producer Cuda voxelizor ?]

A. Basic Voxelization [what does it mean ?]

In this demo, a 3D mesh is voxelized. At each pass, the ray-casting stage requests the content production for missing 3D regions of space. At production pass, we voxelize the mesh node by node in sequence.

This demo is based on the basic "simple sphere CPU" data streaming tutorial where a CPU Producer is used to fill a buffer shared with GPU memory. The GPU Producer read data and fill the associated GigaSpace data cache.

To fill the memory shared buffer during at CPU Producer stage, this time we choose to rely on GLSL shaders to voxelize the mesh. The GigaVoxels API enables to use CUDA/OpenGL Interoperability to be able to use its elements in OpenGL/GLSL.
We do brick by brick 3D rasterization storing normals [?] data with the use of GLSL `imageAtomicAdd()`[for what?]. At GPU Producer stage, data is read and placed in GigaSpace cache. Then, at shader rendering stage, normal is sampled and normalized [?] and a default color is applied. It's kind of binary voxelization.
[very difficult to understand. to be rephrase].

B. Signed Distance Field Voxelization [what does it mean ?]

Here, we do brick by brick 3D rasterization storing a“signed distance field”representation of the mesh. We store the final closest distance to the mesh at a given voxel.
Then, at GPU Producer stage, data is read and “normals” are computed by central finite differences and stored in GigaSpace cache. Then, at shader rendering stage, normal is sampled and normalize and a default color is applied.

5.2.6. Producer with *Oracle* function: managing a Scene-Graph

Mesh voxelization is easy to generalize to a scene graph made of a hierarchy of bounding boxes each associated to a VBO. Then, what we need is an *oracle function* called by the rendered on cache miss. This function will either immediately return “empty area” or take in charge the data production request that will be asynchronously processed by the GPU voxelizer producer.
[do we have an example of oracle ? what is the (real or emulated) API, BTW ?]

Note that here, the idea is to have GigaVoxels managing the full volume of the scene. Another strategy is to let OpenGL manage it and use GigaVoxels only inside proxy objects, possibly instanced several time with no extra memory - which also permit animation. This is treated section N.N.

6. Making a Renderer

The role of the renderer is to march the tree nodes through space and LOD along the cone ray, then the bricks, then to access the voxel data, process some calculation on it (e.g., shading), and accumulate it so as to provide the final results for the ray. It also tune the cones - in case they are less trivial than just pixel footprint aperture. Pre & post rendering phases allow to prepare the ray and process the result, possibly in the scope of interoperability with OpenGL.
All these should in theory be separate functions, but in most demo they are often merged.

6.1. Tuning the voxel renderer

6.1.1. Colors, material and normals: demo “simple sphere 3”

Storing more data in the voxels is very memory consuming, which can alter cache capacity, data fetching time, and thus performances. When it is possible, it is better to store only what is strictly necessary (e.g. density or occupancy, distance field...). Of course this come with an extra cost at rendering time, so the right balance must be evaluated for your very applications and requirements.

Typically, colors are 3 extra values that are often constant for a given object material. It is thus better to introduce them directly in the renderer, either hard-coded or as a static data provided to GigaVoxel. Note that this solution also allows to change the color without invalidating and re-producing the data as in *simple sphere* tuto in section N.N.

Specular BRDF like Phong shading is view-dependent so it is better to compute it in the renderer than to precompute and store it in the voxels.

Storing normals in the voxels is very memory consuming. Moreover, most subsequent transform of the density values also alter the normals. So it can be convenient to compute the normals in the renderer by finite differences.
Note that this come with some possible pitfalls: finite differences are less precise than analytical derivatives (when the shape equation is known) and can be ill-conditioned for low values of density or density gradient. In particular, one should avoid constant density close to the surface for not totally opaque objects since these should better be shaded “like a surface” than based on a zero gradient. Anyway it is always better to ensure at least a 2 voxels-wide density gradient at borders for Nyquist–Shannon reasons (i.e. to avoid geometric aliasing due to spatial frequencies larger than the voxel sampling frequency).

We illustrate these render-time evaluation in [this tuto]

6.1.2. Demo “interactive volume visualization”

A classical process done in volume visualization is tuning the *transfer function* telling which range of densities should be displayed as hard, semi-transparent or invisible, and possibly, in which colors. That way, the user can stress bones, cartilage or soft tissues. In the tuto “loaded volume” in section N.N we did it in the producer, which was forcing the entire regeneration of the GigaVoxels cache at every change. For interactivity, it is way more advantageous to do that in the renderer: the same data keeps valid in the cache, and is simply displayed differently when marching the voxels. [do we have a tuto for volumes ? - beside Perlin noise]

The inconvenient is that we may store, march and shade many invisible voxels. Empty voxels have huge impact on performances since while they cost as much as matter, they don’t stop the rays as matter would. Marching 100 invisible voxels before hitting opaque data is roughly 100 times costlier to render. Still, it probably makes sense for interactive tuning. Note that a smart compromise would be too loosely re-produce the bricks as the user change the opacity transfer function: only a few would be produced at each frame, and the stored voxel occupancy would be closer to the amount of marched voxels. When the user explore without changing the tuning, the performance would be nominal.

6.1.3. Demo “on the fly hypertextures (2)”

Tuning hypertexture parameters is exactly the same problem as tuning transfer functions in the tuto above:
Invalidating and re-building the whole cache at every frame is not smart (and surely not interactive). So we can apply amplification by hypertextures in the renderer rather than in the producer as we did above to tune density.

As above, the advantage is obvious for the interactive tuning of hypertexture parameters. We illustrate this in [this tuto: 9.A `GvDynamicHyperTextures`]. As above, we could apply a smart mix-approach relying on producer or renderer hypertextures depending whether the user is currently tuning the parameters or not.
Tuning is not the only possible deferred transform: as for material color this value could be change while instancing several copies (see OpenGL interoperability) to introduce variations, or fluctuate with time.

Note that on-the-fly evaluation of hypertexture (plus their normals) might be costly, but this can also have advantages:
First, for very opaque or hidden areas every voxels of a brick must be computed at production, while most will never be marched. For fast motions the brick will be valid for only a few frame, so it might be that on average it’s less costly to evaluate on the fly.
Second, a key idea of amplification is that the memory-costly large-scale shape be used through a less memory consuming configuration, at coarse resolution. The normal use of GigaVoxels is to sample rays once or twice per voxel, with voxel footprint matching the pixel size. But we could use bigger voxels - since stored density has smooth variations - , higher details being implicitly created by the unstored hypertexture still sampled at thin scale. This decrease a lot the pressure on the cache. This can be crucial when semi-transparency or many holes prevent the background data to be hidden, which could overwhelm the cache.

6.2. Tuning the tree visitor: Easy rendering effects with GigaVoxels

6.2.1. Soft shadows

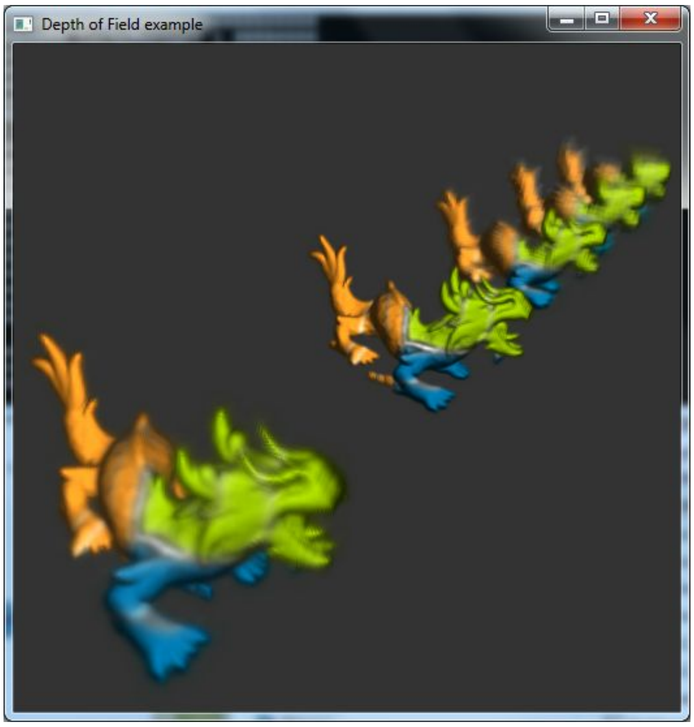
The GigaVoxels renderer is a cone tracer: its aperture corresponds to the back-projection of a pixel in space with distance. But we can change this tuning ! If we make the cone larger, the image will be blurrier. It is not that interesting as such for the rendering of colors, But it is more for computing shadows, for instance to render a shadow map to be applied on the floor (e.g. in an OpenGL application). The soft shadow casted by our scene on the floor from a large source approximated as a sphere is obtained by launching cones from the floor map pixels, with aperture matching the sphere. This will work even for source at the middle of the scene. Demo [`GvCastShadow` ?] illustrate this effect. [see also paper/video].

Note that this effect usually considered as complex and costly in Computer Graphics indeed *improves* performances in GigaVoxels, since we rely on coarser and coarser voxels ! This is an illustration that with classical algorithms we are very far to our “*pay only for what you see*”: with them you have to pay more for less visual information.

6.2.2. Depth of Field

Indeed there is a similar situation for direct rendering: Depth of field can be modeled by a generalized cone which “apex” is at focal plane distance and which enlarge in front and behind it. We just have to setup our cone that way in the renderer and we have depth of field ! We illustrate this `(Gv)DepthOfField` +paper & videos]. In practice we don’t want an infinite precision at focal distance, but rather the max between pixel footprint and pure depth of field footprint.

And here too, this nice effect will also improve the performances, contrarily to usual algorithms.



This example shows the usage of a GPU Shader used to mimics a depth of field effect. The goal here is to reproduce the “circle of confusion” mechanism in depth-of-field, which is done by mimicking the ray pencil: the conic ray - corresponding to pin-hole camera - is replaced by a bi-cone with apex at focus distance. User customizable API provides access to a cone aperture function which indeed controls the size of the pencil with depth, and can thus be used to model lens effect.The shader is responsible for modifying the cone-tracing behavior [no, probably not this guy, since LOD determination is involved !]. Apart from this only point, all the rest is based on the same tutorial as the “dynamic load” (data streaming of 3D pre-voxelized 3D model),

6.3. Pre & post rendering

In the simplest cases the scene is limited to the GigaVoxels volume, cone-rays start from the virtual screen assumed to be in front of the volume, and the resulting color buffer is displayed. But many other situation exists, where the volume interacts with other scene elements or user controls, where the geometry of rays is altered, or where calculated values are not directly displayable. Pre & post rendering phases allow to tackle these cases.

6.3.1. Tuto: Volume visualization with clipping planes

In volume visualization, for instance when a doctor analyses some scanner data, it is sometime convenient to introduce virtual cuts in the data, that can be interactively displaced. Cutting along a slice means hiding the volume data in front (or behind). Of course erasing the stored content is an option, but pretty inefficient in an interactive context since many bricks should be modified or restored at every frame. A simpler solution consists in making the rays start (or stop) at the cutting plane.
Modify the starting ray is a simple initialization in the pre-rendering function.
Forcing the farthest ray end consists in computing the max distance for the the ray, and testing this variable during marching.

For a single cutting plane it is easy to modify the initialization manually. For several intersecting planes in different directions, or for both a front and a back plane, an easy way is to rely on OpenGL to rasterize the planes, then to grab the Z-buffer - or possibly a min and a max Z-buffer - and provide them to the GigaVoxels renderer to be used as start distance and max distance for each ray. [details ?]

[`GvClippingPlane` where is the demo ? grab it from 9.2 ?]

6.3.2. Tuto: Volume visualization with multiple transfer functions

Similarly, the user might be interested to see different depictions - rather than display on or off - between the limit planes, for instance with different color lookup and opacity transfer function.
For this, we just have to launch GigaVoxels several times between the appropriate pair of Z-buffers using the appropriate appearance settings. The blending of the partial layers can be programmed either on GigaVoxels side - providing initial values for the colors - or on OpenGL side - displaying the results as an RGBA texture.
[details ?]

6.3.3. Tuto: Lense effects

The trivial tuning of rays direction and aperture corresponds to the simple Pythagorician model of pin-hole camera. But we are free to tune them as we wish. For instance, as they would be passed a lense. [demo ? `GvRayMapGenerator` ?]

6.3.4. Tuto: Special color spaces, multispectral rendering

The data type of colors cumulated through the marching could be anything else than RGB encoded as a triple of floats. For instance the samples might not be taken in the visible spectrum, or in more or less than 3 bands, or special encodings (double, log like optical depth, ...).
The GigaVoxels programmer is free to manipulate whatever variable during rendering as it is free to store whatever variable in bricks. The post-renderer function is then typically used to convert the result in some displayable format.

7. Interoperability with OpenGL

In OpenGL applicative world, you may need more effects difficult to obtain, or you may have efficiency issues. In GigaVoxels world, some things are more easy and efficients, but sometime it's the contrary.
Or you may have a huge existing base of OpenGL code and first want to improve small parts at a time using volumes without having to dangerously jump the big gap of totally changing everything.
Or you play specifically with volumes, but as we have seen above OpenGL can help.
Why not taking the best of both worlds ? Or any intermediate balance that will best ease your life.

7.1. OpenGL and GigaVoxels share the world

Both OpenGL and GigaVoxels know how to do a Z-test and how to blend colors. In particular, both can be provided with an existing Z-buffer and account on it to render only visible fragments (for OpenGL) or voxels (for GigaVoxels).
On GigaVoxels side, this corresponds to the pre/post rendering: we launch rays only if not masked, start at Z=nearPlane, and pre-set the max Z at which the ray should stop. After rendering, we return the color and Z buffers to OpenGL.

It is generally interesting to do first the OpenGL pass - typically with opaque objects - then the GigaVoxels pass - better than OpenGL for semi-transparent objects. Note also that the more the scene is hidden by OpenGL objects, the less GigaVoxels will be costly (remember GigaVoxels motto: "*pay only for what you see*"). But you could also render GigaVoxels first - efficiently managing a very large and complex environment - , then rely on occlusion queries (see also here) to test which of your OpenGL character or object will be visible.

7.2. OpenGL features in GigaVoxels

Indeed it is easy to store and provide to GigaVoxels environment elements of the OpenGL scene, in order to benefit of existing GUI or to ease the integration with an OpenGL scene.

At voxel shading one can easily combine the effect of several lights, acces a shadow- , light- or environment maps, projected textures… We might even store in bricks or procedurally compute u,v texture coordinates if we wish and fetch a texture map at rendering.

7.3. GigaVoxels in an OpenGL proxy geometry

7.3.1. Proxy bounding box

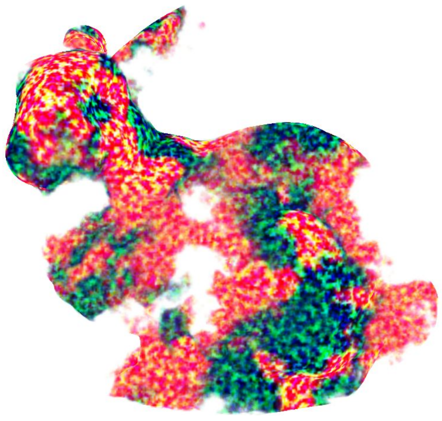
The idea is that within the OpenGL scene, one object (or more) is fake, replaced by a stub box: at this location we will included color and Z coming from another renderer - here, GigaVoxels. On GigaVoxels side, we just have to lauch rays starting at the front face and ending and the rear face, much like for the volume clipping above (section N.N). Assuming our GigaVoxels volume is meant to be align with the proxy box, we can apply the model-view matrix on rays in the pre-rendering, and mask all the pixels not covered by the pixel footprint of the box - another obvious optimization is to clamp the GigaVoxels viewport).
This is illustrated in tuto [`GvProxyGeometry`]. [more to say ?]

Note that the proxy box could move, turn, zoom: this is a convenient way to animate GigaVoxels scenes made of rigid objects without having to trash the whole cache. We will see soon that it also permit to instantiate them many time at same memory cost.

7.3.2. Proxy shape

In some situations, isolating the alternate renderer via a box is too naive. For instance, very un-convex shapes in crowded environments where parts of other objects may interleave. In such case you need a more precise delimitation of the volume of alternate rendering.
Another application is when using volume as a pure 3D material, i.e., hypertexture, to be carved by an OpenGL object. This is what we illustrate in demo [(Gv)DepthPeeling GvShellNoise? ProxyGeometryManager?].
A classical solution to generate such interfacial shapes is *depth peeling* : several algorithms exists to produce the multiple Z-buffer encapsulating the shape. GigaVoxels is then launched between the "inside" segments. A poor-man approximation is to get only the closest and farthest Z-buffers; then the pre/post is the same as for boxes.

Here, OpenGL is used to rasterized front and back faces in order to obtain a shell representation of the 3D mesh. The front and back Z-buffer [or more layers ?] are sent to GigaVoxels on Cuda side that use then to start rays from the front shell and stop them to the rear shell, instead of using the default GigaSpace Bounding Box.
In this demo, Perlin noise hypertexture is added in the GPU Producer. [how ugly is this image. tune nicer !] We will see later [anchor: 7.5] that we can also achieve the same look in one all-integrated pass, by using GLSL GigaVoxels directly in a GLSL shader.



7.3.3. Instancing

In a number of Computer Graphics complex scenes a same object is duplicated many time, possibly with some appearance changes. Typically, for trees in a landscape or in a forest we could store only one volumetric model of tree in GigaVoxels, and *instantiate* it many times by inserting as many proxy bounding box at different locations. Then it is easy to also modify the orientation and size, and even to introduce symmetries: it just changes the mapping of GigaVoxel rays to pixels on the proxy surfaces.
[first basic demo. GvInstancing .Where is it ? What to say about sorting ? and naive vs smart ways (limiting the context switch) ? perfs ?]

To add variations, we can store only intermediate data without color, and varies colors or any other parameter on the fly in the Voxel Renderer (even density, or adding some noise). For this we just have to associate to each proxy cube a small record of shading parameter sent to GigaVoxels when rendering the box. [details ? where is the demo ?]

Variations are not always sufficient, we might also want to mix a few different model of trees. The next section explains how to store several independant volume in GigaVoxels, which is used in combination with instancing to implement a rich landscape using little memory [where's the demo ? OpenGLScene ?].

7.4. Multiple objects in the same GigaVoxels cache

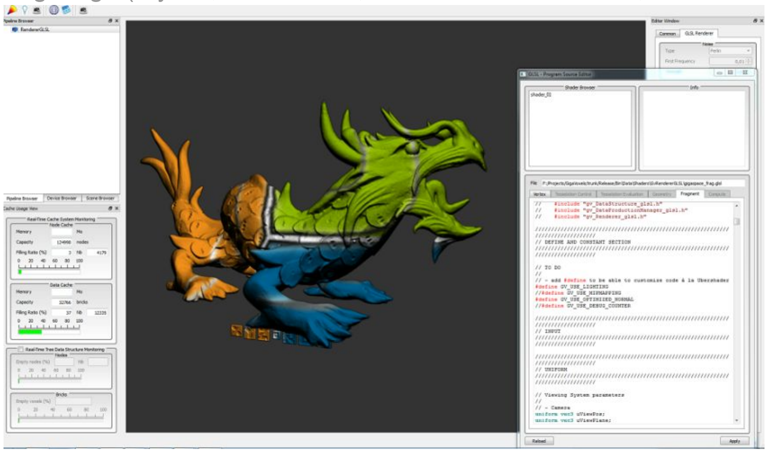
GigaVoxels is firstly a cache, for tree nodes and bricks. As such, there is no problem in storing many different objects in the same cache: the semantic of these objects lies in the producer and the renderer.
When rendering objects of one kind or the other, one simply have to settle the appropriate one.
[detail how !] [MultiObject]
Of course for efficiency reason it may be interested to groups instances of the same kind.
[is it true ? maybe only in the case of smart instancing limiting the context switch ?]

7.5. GLSL GigaVoxels renderer (bye bye Cuda !)

You might not want to cope at all with Cuda for the rendering in your application. As a choice, or to avoid cumulating context switch between Cuda and OpenGL. Also, CUDA is not yet very convenient during debug and code tuning since everything must be recompiled.
For all these reasons we also provide a pure GLSL version of the renderer `[ (Gv)*GLSL ]`. It is particularly convenient for direct insertion in pixels shaders, avoiding multipass and context switch, easing dynamic code tweaking (which is also convenient for learning).
But it's *only* the renderer: it cannot handle memory allocation and production. So you will either use it with data entirely in GPU memory, or still use a pass of Cuda at every frame to account for possible data production requests.
Note also that the early implementation of GigaVoxels (see I3D'09 GigaVoxels paper) were in pure GLSL, with the data allocation and management handle on CPU. So this is also an option… (but we don't provide this old piece of code :-)).

7.5.1. GLSL Renderer : data streaming (3D model) [what does it mean ?]

Based on the *basic Dynamic Load* tutorial to stream data from CPU (a pre-voxelized 3D model), this demo shows how to use the OpenGL GigaSpace renderer and its associated GLSL API to be able to customize on-the-fly the Ray-Casting stage (ray init, intersection, data structure traversal, brick visiting, etc…)

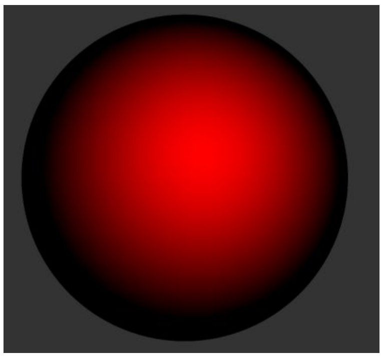


File: F:/Projects/GigaVoxels/trunk/Release/Bin/Data/Shaders/GvRendererGLSL/gigaspace_frag.glsl

```
//    #include "gv_DataStructure_glsl.h"
//    #include "gv_DataProductionManager_glsl.h"
//    #include "gv_Renderer_glsl.h"

//////////////////////////////////////////////////////////////////////
///////////////////
// DEFINE AND CONSTANT SECTION
//////////////////////////////////////////////////////////////////////
///////////////////

// TO DO
//
// - add #define to be able to customize code à la Ubershader
#define GV_USE_LIGHTING
//#define GV_USE_MIPMAPPING
#define GV_USE_OPTIMIZED_NORMAL
//#define GV_USE_DEBUG_COUNTER

//////////////////////////////////////////////////////////////////////
///////////////////
// INPUT
//////////////////////////////////////////////////////////////////////
///////////////////

//////////////////////////////////////////////////////////////////////
///////////////////
// UNIFORM
//////////////////////////////////////////////////////////////////////
///////////////////

// Viewing System parameters
//
// - Camera
uniform vec3 uViewPos;
uniform vec3 uViewPlane;
```

7.5.2. GLSL Renderer : procedural shape on device (GPU) [what does it mean ?]

Same idea than the previous tutorial (GLSL Ray-Caster), but based on the basic Simple Sphere tutorial to produce data on-the-fly with the help of a GPU Producer.



7.6. GigaVoxels as effects booster for OpenGL

We already mentioned some direct motivations for inserting GigaVoxels objects in OpenGL scenes: volumetric or semi-transparent objects such as trees, or complex-looking 3D material such as hypertexture.

in section N.N we also shown that we could use a coarsely voxelized version of the scene to compute dynamic soft shadows on OpenGL surfaces: we just have to launch GigaVoxels shadow rays from fragment shaders (the GLSL version would be the simplest) when rendering OpenGL objects.

Indeed, exactly the same way that we computed shadows, we can add reflections and refractions to OpenGL objects by setting the reflected or refracted ray and rendering GigaVoxels from the fragment shader.
It is especially interesting for glossy reflection and refraction since crude voxelization can be sufficient, which permit re-voxelization of dynamic scenes or deforming objects at every frames.

[we need to give details ! for GLSL case, and for Cuda case !]
[where are the demo ? `GlossySurface, RefractingLayer, GvEnvironmentMapping`]

In the PG'11 paper "*Interactive Indirect Illumination Using Voxel Cone Tracing*", it's the whole global illumination, comprising glossy multiple reflections, that were evaluated in voxel space while objects were classically rasterized in OpenGL.

8. Beyond *GigaVoxels: GigaSpace*

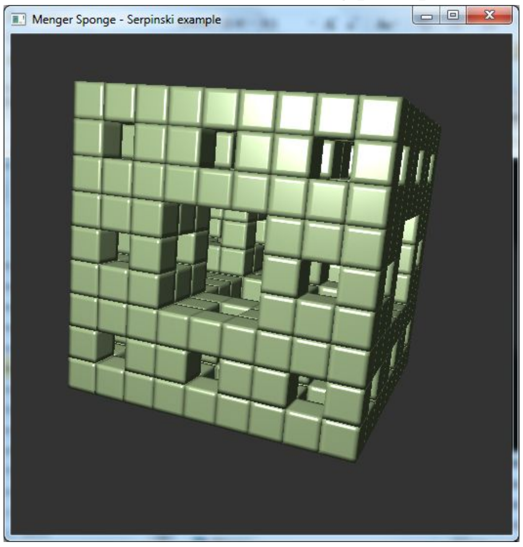
As we said in the initial presentation in section N.N, GigaVoxels is so modular that it can do a lot of other things than just Sparse Voxel Octrees. From variants, to totally different representation of space and matter, to tasks other than rendering.

8.1. Beyond SVO: Alternatives to octrees

In GigaSpace you can define your own space partitioning hierarchy, as long as you know how to create and march it. We propose a few demos to illustrate some possibilities.

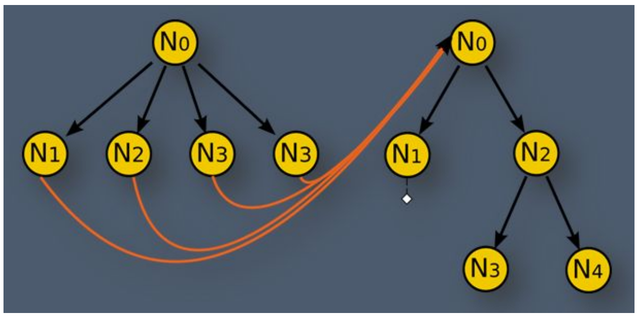
8.1.1. "Menger Sponge" tutorial : 27-tree with loops

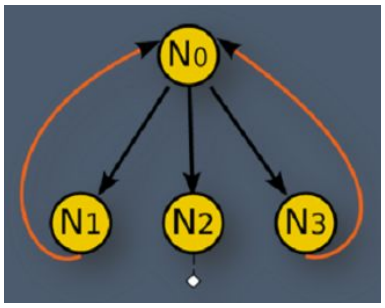
[`(Gv)MengerSponge`] In the Menger sponge fractal there is a 3x3x3 brick pattern repeating at all scales and locations. It is thus tempting to factor it, so as to store only one tree brick and one voxel brick.
Factoring it means making a virtual tree structure, where all children link to the same unique parent instance. At marching we simply switch from the tree descent to the voxel brick when the pixel scale LOD is reached.
Another particularity of 27 tree is MIPmapping: we must replace power-of-2 logic by power-of-3.
Since there is indeed no production and no storage, this makes a good demo to bench rendering up to very deep levels in the tree - indeed the bottleneck is the floating point precision. [link to video]



In this strange example, only 1 brick of voxels is produced but the node data structure loop infinitely on itself : nodes are subdivided but points on the same brick.

Loops in node based data structure allows to create complex scenes / objects with repeated or fractal patterns (but constrained by grid and size alignment, contrary to OpenGL instantiation).





8.1.2. N3-trees, kd-trees, BVH

There are many other useful cubic-hierarchy subdivisions, and all the one resulting of constant size-blocks are convenient for GigaSpace. For instance,

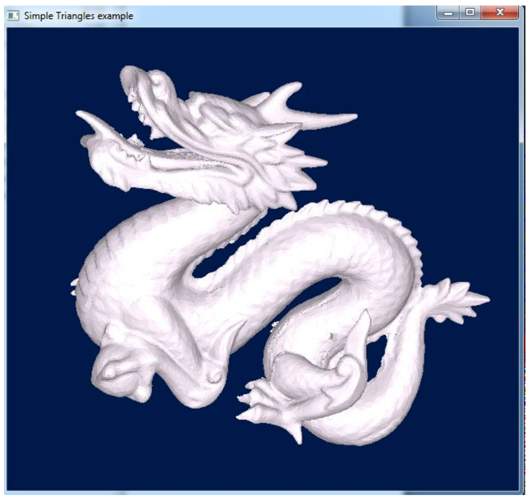
- N^3-trees: splitting regularly the sides by N rather than 2 for octrees or 3 for 27-trees. Having "large" trees allow to limit the depth of the descent.
- k-d trees: binary tree where each level subdivides either in x, y or z, at a given ordinate value.
- BSP-trees: binary tree where each level subdivides along an arbitrary plane (or hyperplane).
- Bounding Volume Hierarchy: for instance, bounding boxes of bounding boxes, either axis-aligned or not. The demo "*Simple Triangles*" below uses such a BVH tree.

8.2. Beyond SVO: Alternatives to voxels

The GigaSpace "brick" is basically a data chunk of fixed size: indeed, it's up to you to use this storage to describe the content of this region of space, as long as you know how to produce it and to march it.

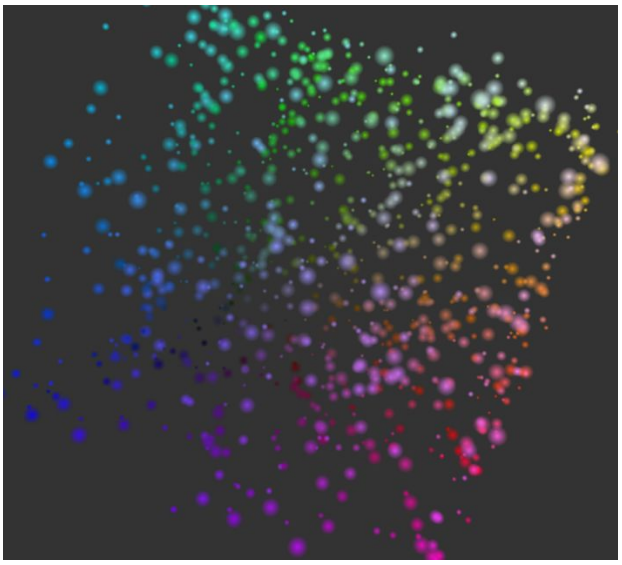
8.2.1. Data chunk = triangles: Demo "Simple Triangles"

In this demo, the bricks contain geometric triangles that the renderer will ray-trace. In order to balance the storage so as to fit constant-size data chunks, we rely on a BVH hierarchy (it is more "large" and thus less "deep" than a BSP hierarchy). In this simple "proof of concept" demo this partitioning is precomputed on CPU and loaded on demand by the CPU producer. Apart from this deferred loading, the renderer is very similar to an ordinary ray-tracer based on meshes.
[`GvSimpleTriangles` ? what about the LOD levels ? still there ? Where is the "factory" demo ?]
Based on the same model as the simple sphere CPU, this example shows how to handle BVH. This demo is more a proof of concept, implementation is far from optimal. Mesh is loaded and pre-process to store a list of triangles in a BVH way.This list of triangles is then traversed at rendering stage. Rendering directly use triangles data, there are no octree or voxel at all in this demo.



8.2.2. Data chunk = geometric primitives: Demo "Ray-Tracing of Spheres"

In this demo, the bricks contain a list of spheres, i.e. of positions, radius and colors. At brick rendering, local ray-sphere intersections are computed and sorted, then shaded. The hierarchy is mostly used to insure that bricks cat fit the local amount of spheres (but rendering is indeed more efficient is bricks contain very few spheres) and to isolate empty area (so that no test at all occurs in these regions). [GvSphereRayTracing +GvInfiniteSky? What about LOD levels ? still there ?]



8.2.3. Data chunk = VBO

In the demo "*Dynamic Frustum Culling*" below (section N.N), GigaVoxels is not directly used by rendering, but to manage points organised as a VBO to be later render as OpenGL GlPoints. Bricks thus contain a chunk of VBO. [GvVBOGenerator. more to say ?] [split between here and 8.3.2]

8.3. Beyond rendering

GigaSpace is at first a smart cache organizing spatial content. It can thus be of use for other tasks than rendering. Here we give a few illustrations.

8.3.1. Querying the data structure

This is the most trivial use: for any kind of usage, using GigaVoxel as a simple storage, which the only visitor consists in fetching a value at a given location in space. And possibly at a given resolution, which gives information about neighborhood. Note that this could be adapted to any number of dimensions.

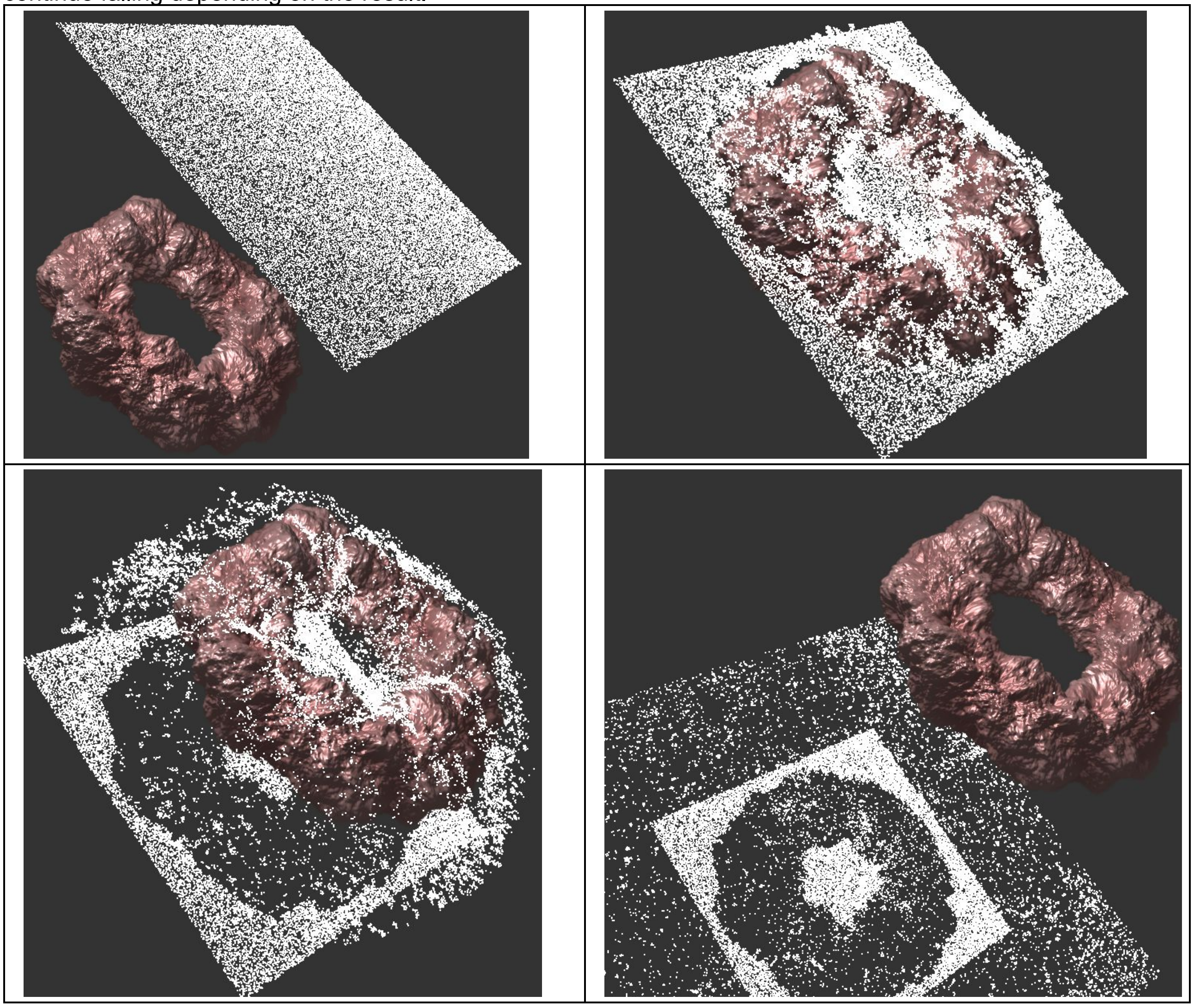
8.3.1.1. Solid textures, cached procedural textures

In graphics - e.g. OpenGL rendering via a GLSL shader - this can be used to encode texture content: for parameter-free textures like [octree textures] or solid textures (e.g. for wood or stone material into which the surface is carved), but also for ordinary texture-coordinates based textures - GigaVoxels will then rely on the uv or uvw space. This can be advantageous for multi-resolution textures, for storing data only where it is visible (and only at required resolution), and for procedural textures which costly evaluation would be cached for next frames.

8.3.1.2. Collision detection

[`GvCollision, GvCollisionBBox` What about the second demo ?]

Vector data (i.e., Lagrangian structures) such as meshes don't allow direct access to spatial content at given coordinates. That's why collision detection is a costly complicated task. Raster data (i.e., Eulerian structures) such as grids and octrees naturally offer a direct access to spatial content at given coordinate. So GigaSpace can be used to test whether a given location or neighborhood of space is empty or occupied, and possibly provide extra information such as the coverage or the local gradient - i.e., the normal. For most applications a coarse voxelization of the object should be sufficient.
In the following demo, a OpenGL VBO of particles is launched against a GigaSpace object (noise-based torus). The idea is to query the data structure for collision at each particle location, and to make them bounce or continue falling depending on the result.

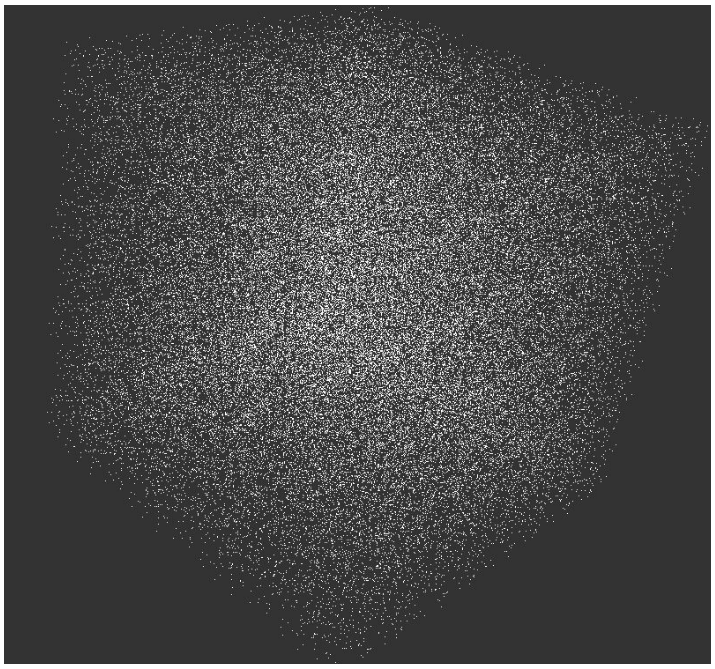


The demo is based on the same tutorial as the simple sphere on GPU, with the add of Perlin noise (a fully procedural GPU Producer). At each frame , a CUDA kernel is launched to update VBO particles by mapping them to CUDA memory (CUDA Graphics Interoperability) and querying the GigaSpace Data Structure.

8.3.2. GigaVoxels as a dynamic VBO culler and sorter : Demo "dynamic Frustum Culling"

The problem with projective rendering (like OpenGL) is that there is no native information about whether objects are visible or not, and at which resolution. They must be pre-organised in hierarchy, tested at rendering time. But for moving objects this hierarchy must be reorganised at every frame. Another problem is that semi-transparent objects must be sorted - most games can't afford this for massive data such as foliage, that's why foliage is so often aliased).
An especially bad case is particles, rendered as textured GlPoints: they are numerous, dynamics, and semi-transparent.

GigaVoxels can be use to select what is visible - i.e., in the view frustum, and not hidden by the environment. Moreover it naturally visits from front to back, which can be used for sorting. We just have to write a producer and a marcher selecting chunks of points VBO, the result of this "rendering" being a VBO contain the subset of visible points in depth order. [explain ! `GvVBOGenerator`]



[comments ? comparative perfs ?]

9. Around Rendering : animation and loading

9.1. Animation in GigaVoxels

Animation is not easy in voxel world. Still, there are several handles to make them move.

9.1.1. Animated scene graph

As we have seen in section N.N, GigaVoxels embedded into proxy geometry can be translated, rotated or scaled through the Object-View matrix like any object.

9.1.2. Animated deformation

Non-rigid transformations require more care since rendering is done in the deformed parametric space:

- inverse transpose jacobian of deformation should be applied to normals,
- rays direction should be transformed in GigaVoxels pre-renderer
- rays could be curved, so the marching should accounts for that (or approximate it).

Note that this can be used to simulate FFD deformation within a shell map, as in EGCAS'95 "Animates Texels" article.

9.1.3. Animated volume (at production)

Trashing the whole cache and re-producing the necessary bricks at every frame is sure a costly solution, but it can be affordable for limited objects and resolution. A moving hand was re-rasterized at every frame in in the interactive demo (see video) of the article "*Interactive Ambient Occlusion Using Voxel Cone Tracing*".

9.1.4. Animated volume (at rendering)

We have seen than we can animate parameters of procedural noise or material (comprising opacity). [anchor to A]

For very limited animations, simple other tricks can be used:

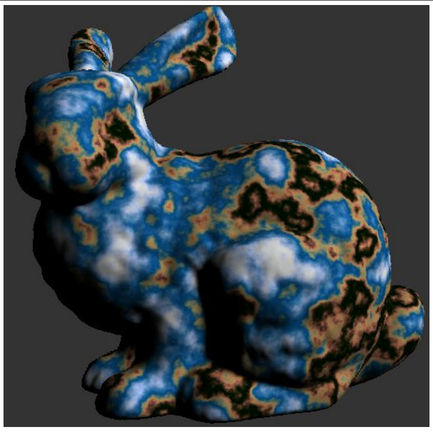
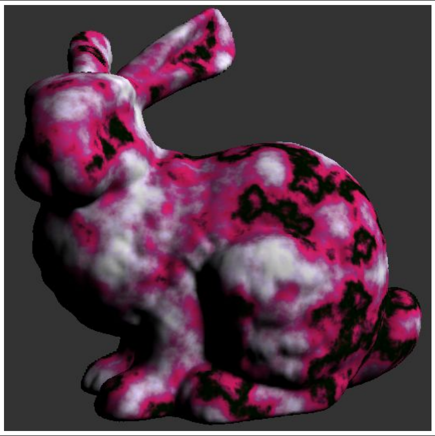
- Storing a few bricks for the same location to be cycled in time as cartoons do, but this can be memory consuming. [anchor to B or C ?]

- Using ColorMap faked animation, mapping indexed values as opaque/transparent or colors with a cycling mapping. . [anchor to B or C ? + `GvAnimatedLUT`]

[From here, big mess to sort out !]

A. Dynamic hypertexture

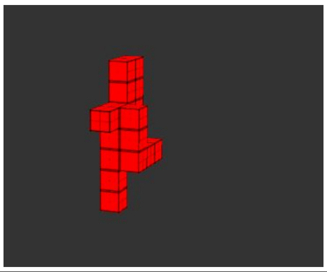
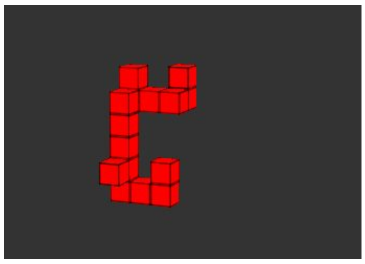
This demo is based on the Amplified Surface demo where a GPU Producer was perturbating a 3D model surface with Perlin noise (model is represented by a Signed Distance Field : distance to surface + normals at each points). But this time, Noise is applied at Rendering stage where shader apply animation with LUT to access texels.

B. Animated snake

BEWARE : not sure of J. work => should have been like old-style Amiga games with LUT, but here ?

The demo [`GvnimatedSnake`] is based on a simple Procedural GPU Producer. Rendering creates animation on-the-fly.

C. Animated fan

BEWARE : not sure of J. work => should have been like old-style Amiga games with LUT, but here ?

The demo [`GvnimatedFan`] is based on a Data Streamer Producer like the Dynamic Load (loading files from CPU from a pre-voxelized model of Fan). Rendering creates animation on-the-fly.

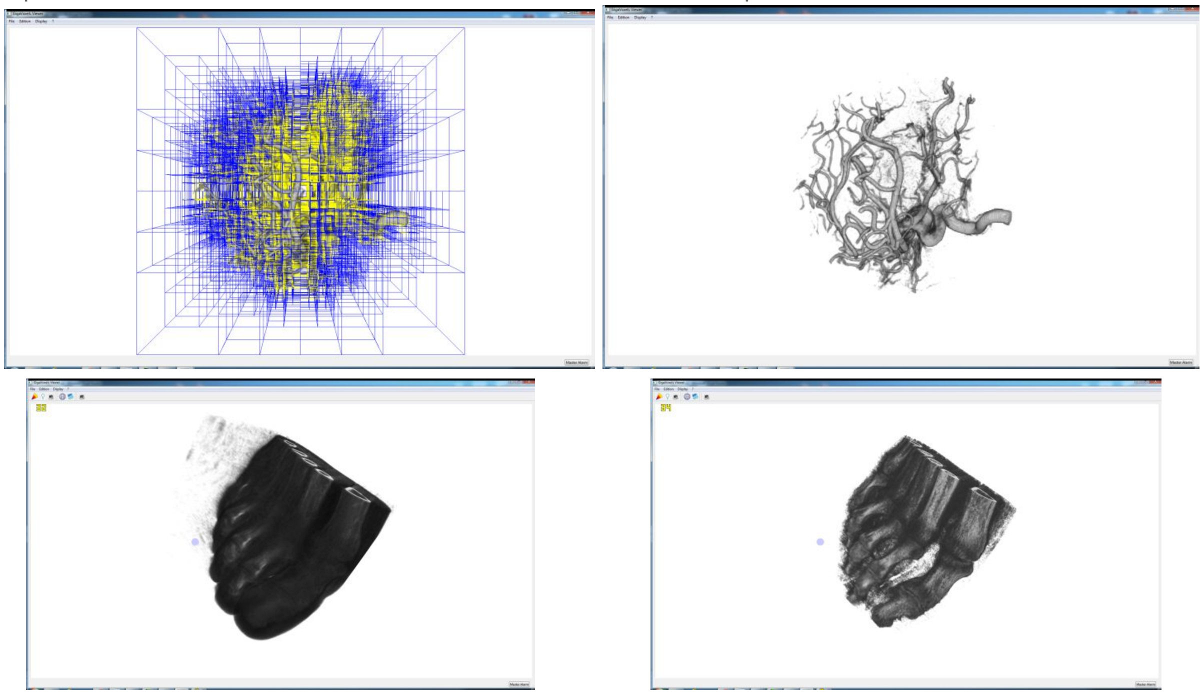
9.2. More about volume visualization

[to be distributed elsewhere. 6.1.2 or 6.3.1 ? or 10 ?]

[GvDicomViewer, GvDataConverter, GvRawDataLoader ,..]

A. Raw data

The demo is based on a Data Stream Producer like the Dynamic Load (loading files from CPU from a pre-voxelized model of Fan [?]). This demo shows how to customize the voxelization tool to create voxelized representation of scientific data stored in a basic RAW file. Example : scanner dataset of scalar data.



LUT and threshold can be used to remove noise at production and/or rendering time.
Thresholding at production time enables to better fit the octree [?] and accelerate the data structure traversal at rendering stage.

B. DICOM data

Same as before, for using a dedicated pre-voxelizer tool [?] to handle DICOM data files

10. Extra demos

[If we want to push here demos that are too complex / buggy / ugly to be "tutos" . Like, all the ones that are in sources but not in the pre-compiled set ?]

11. Tuning performances

GPU programming is uneasy and often not intuitive to optimize. Of course usage of memory and computation costs, but many other things impacts the huge difference between the *fast path Grâal* and the slow path often there (either you know it or not). In addition, GigaVoxels is a huge piece of code and data structure with many handles. Also, there are different requirements and modalities of use. Here, we give some tools, advices and tricks to better optimize your use.

11.1. Flags, modes and priority settings

11.1.1. Modalities of use

There are 2 main different ways of using GigaVoxels; speed-oriented or quality-oriented:

11.1.1.1. Speed oriented

You may prefer the fastest results, possibly with temporary artifacts, and get the nice image when you stop moving. This is the modality generally chosen in visualization.

In this case, the batch of production requests is honored only once per frame. So some bricks might be temporarily missing or too low-resolution at one frame, then progressively updated at the next frames. For this, do [explain]

11.1.1.2. Quality oriented

Or you may prefer to have the correct result at any frame, artefact-free, even if it might be some time slower. This is the modality generally chosen in realistic applications.

In this case a rendering pass ends when all rays are either finished or waiting for data, ray distance is stored in a buffer, then a production pass allows to relaunch the rendering or rays where it stopped, up to complete rendition.
See the demo "*quality rendering*". [more that one demo, I guess ?]

11.1.2. Flags and preference setup

In fact we can also mitigate intermediate behaviors:

11.1.2.1. Time budget

We can allocate a time budget, and interrupt production or quality iterations when it is passed.
[show example, or explain how]

11.1.2.2. Priority flags for production

We can settle priorities in brick requests, so as to produce first the most important according to some user-defined criterion. This is typically used together with time budget.

For instance you might want to give priority to bricks for which no parent is already in cache - using the parent whenever present instead of the missing child just give an area slightly more blurry. Or to front stage bricks.
[show example, or explain how]

11.2. Strategy choices

- Will you define the shape in CPU producer, in GPU producer, or on-the-fly in the renderer ? All of it, or part of it ?
- ‘Part of it’ includes attribute: will you store normals, colors, etc, or just the minimum ?
- Can you separate 2 scales having 2 different characteristics and thus different best treatment, as coarse explicit shape vs procedural amplification ?
- What will you do in Gigavoxels or in OpenGL ? Will you use Cuda or the GLSL renderer ?
- Will you use progressive rendering or quality rendering ? or time budget and priorities ?

11.3. Perf killers and optimization

Too slow.
- Is it in GigaVoxel (Cuda phase), or in OpenGL ? or in an iterative scheme ? (check base cost vs overhead).
- Is it due to the production or to the rendering ? Per pass or overall (check overhead) ?
- CPU production: evaluate the transfer bandwidth. Is it reasonable ?
- Is the cache swapping ?

Overhead:
- Are all your synchronisation point necessary ? and do you know all of them ?
- Are some batches too small ? May you group bigger batches ?
- Are buffer conversions or transfer costly ?

Is the cache swapping ?
- Too much memory per brick, because too many attributes stored per voxel.
- Too many bricks required at the same time. For opaque scenes it shouldn’t be, since this number should be the order of magnitude of bricks required to tile the 2D screen. Verify your brick transparency, your blending equation, and your opacity threshold.
- For ‘texture’ or shape yielding a lot of empty voxels per brick, try smaller bricks.

Rendering is too slow.
- Estimate the average number of bricks and voxels visited per ray. Is it reasonable ?
  Idem for the worst case (since worst pixel can totally kill the parallelism).
  If too many bricks/voxels visited: Too much transparency.
- Is there situations where you can avoid the full calculation ? Typically, try to do no calculation at all in empty voxels (no normals, shading, blending, proceduralism, …).
- Check the various GPU programming tricks, typically concerning the patterns of poor parallelism (if, etc).
- Might you store some costly terms in the brick ? (e.g, normals).

Too much transparency: the algorithmic transparency might be higher than you think.
- due to a bug: in the blending, or in production (empty bricks still producing voxels).
- or to the fact that 99% opacity let the rays continue marching the volume. You might change this threshold.
- or to bricks containing many empty voxels.
- One often have a thin semi-transparent skin on object. Is it thin ? (You don’t want to go deeper than 2-3 voxels at front hit). How many voxels do you visit per pixel ? empty, vs filled voxels ?
- How much silhouetting do you have ? (especially as non-empty bricks). Transparent voxels close to the silhouettes are evil, and might represent most of the rendering cost.

Production is slow:
- is it the number or bricks to produce, each brick that cost much, or the number of batches ?
- Are your voxel production scheme well parallelized ?
- Are your batches big enough ? Or can huge batches be balanced at previous (or next) frames ?
- Proceduralism: can you avoid some calculations ?
    - In transparent areas
    - In masked areas (e.g. inside cores)
    - Factoring in the algorithm.
- Might you reuse data of the parent ? (a possibility is to store internal data + renderable data to ease reuse. Still, control that the overheads don’t ruin this “optimization”.)

11.4. How to save the day

- better stop criterion for rays. filled. low freq.
- change LOD (and thus stepping) depending of current opacity.
- You don’t want progressive rendering because of artifacts: try priorities.
- Less volume or screen resolution can be a lot cheaper and not always visible. See our “adaptive resolution” demo. [where is it ? anchor]
- more in producer / in renderer.
- More lazy. (but verify optimization really optimize)
- Avoid storing the invisible parts (inside of objects).
- Treat constant density bricks the same way that empty bricks: one dataset per node, not per voxel.
- Full calculation only for large visibility (not for deep voxels).
- “Teleportation problem” (or “first image problem”): rebuilding the totality of bricks for the same frame is very costly. This often happen in predictable situations: at start, when clicking a “go there” option. A possibility is to store on CPU the cache state for these locations, and to upload it all in once.

12. Notes about the future and ideas

[copy-paste the TodoList]

13. GigaVoxels / GigaSpace API

[copy-paste the Doxygen]