What is GigaSpace / GigaVoxels ?

GigaVoxels

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

- In the most basic usage, you are already happy with what does one of the examples (good for you: nothing to code !)
- The most basic programming usage consists in writing the callback producer function that GigaVoxels will call at the last moment to obtain data for a given cube of space at a given resolution. This function has one part on CPU side and one part on GPU side. You might simply want to transfer data from CPU to GPU, or possibly to unpack or amplify it on GPU side. You might also generates it procedurally, or convert the requested chunk of data* from another shape structure already present in GPU memory (e.g., a VBO). footnote*: The basic use of Gigavoxels considers voxel grids since it is especially efficient to render complex data on a screen-oriented strategy: data are visited directly and in depth order without having to consider or sort any other data, while OpenGL - which is scene-oriented - has to visit and screen-project all existing elements, an especially inefficient strategy for highly complex scenes. LOD and antialiasing is just a piece of cake using voxels, and soft-shadows as well as depth-of-field are easy to obtain - and indeed, faster than sharp images. So, your producer might be a local voxelizer of a large VBO - BTW, we provide one.
- You probably also want to add a user-defined voxel shader to tweak the data on the fly or simply to tune the appearance. In particular, it is up to you to chose what should be generated on GPU, in the producer or in the voxel shader, depending on your preferred balance between storage, performances and needs of immediate change. E.g., if you want to let a user play with a color LUT or tune hypertexture parameters, this should be done in the voxel shader for immediate feedback. But if these are fixed, rendering will be faster if data is all-cooked in the producer and kept ready-to-use in cache.
- You might want to treat non-RGB values, or not on a simple way. Gigavoxels let you easily define the typing of voxel data. You can also customize pre and post-rendering tasks, various early tests, hints, and flag for preferred behaviors (e.g., priority strategy in strict-realtime).
- The basic Gigavoxels relies on octree dynamic partitionning of space, which nodes corresponds to voxel bricks - if space is not empty there. But indeed, you can prefer another space partitionning scheme, such as N^3-tree, k-d tree, or even BSP-tree. You can also prefer to store something else than voxels in the local data chunck, like e.g., a piece of mesh.

GigaSpace

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

- A multi-scale space-partitioning dynamic tree structure,
- A cache-manager storing constant-size data chunks corresponding to non-empty nodes of the space partition.
- A visitor function marching the data.
- A data producer called when missing data are encountered by the visitor. Datatypes can be anything, 'space' can represents any variety. The basic use of Gigavoxels use octrees as tree, voxels bricks as data chunk, and volume cone-tracing* as visitor. But we provide many other examples showing other choices. footnote*: volume cone-tracing is basically volume ray-marching with 3D MIPmapping.

Cuda vs OpenGL vs GLSL

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

- Zbuffer integration: You can do a first pass with OpenGL then call Gigavoxels providing the OpenGL color and depth buffer so that fragments are correctly integrated. Similarly, Gigavoxels can return its color and depth buffers to OpenGL, and so one with possibly other loops. Note that hidden voxels won't render at all: interoperability keeps the gold rule "pay only for what you really see".
- It is easy to tell Gigavoxels to render only behind or in front of a depth-buffer, which makes interactive volume clipping trivial.
- Volume objects instances and volume material inside meshes: You can use an openGL (bounding-)box or a proxy-geometry, use their objectview matrix to rotate the volume view-angle accordingly, provide only the visible fragments to Gigavoxels, and tell it to render only behind the front depth and (optionally) in front of the rear depth. This allows for the OpenGL display of transformed instances of a given volume (e.g., to render a forest) as well as shapes seeming carved into openwork 3D material.
- auxiliary Volumes for improved rendering effects; ray-tracing as second bounce: Gigavoxels is basically a ray-tracer so it can easily deal with reflexion, refraction, and fancy cameras such as fish-eyes. Moreover, it is especially good at blurry rendering and soft-shadows. Why not using it only to improve some bits of your classical OpenGL scene-graph rendering ? Binding Gigavoxels to a surface shader, you can easily launch the reflected, refracted or shadow rays within a voxelized version of the scene.

History and roots of Gigavoxels

Gigavoxels draws on:

- The idea that voxel ray-tracing is a very efficient and compact way to manage highly detailed surfaces (not necessarily fuzzy or transparent), as early shown in real-time by the game industry reference John Carmack (Id Software) http://www.pcper.com/reviews/Graphics-Cards/John-Carmack-id-Tech-6-Ray-Tracing-Consoles-Physics-and-more with Sparse Voxel Octrees https://www.google.fr/search?q=sparse+voxel+octree&tbm=isch (SVO), and for high quality rendering by Jim Kajiya http://www.icg.tugraz.at/courses/lv710.087/kajiyahair.pdf with Volumetric Textures https://www.google.fr/search?q=volumetric+textures&tbm=isch , first dedicated to furry Teddy bears, then generalized by Fabrice Neyret http://hal.inria.fr/index.php?action_todo=search&submit=1&s_type=advanced&submit=1&p_0=contained&v_0=volumetric%20textures&f_0=TITLE&c_0=&l_0=or&p_1=contained&v_1=texels&f_1=TITLE&c_1=&l_1=and&p_2=contained&v_2=&f_2=TITLE&c_2=&l_2=or&p_3=contained&v_3=&f_3=TITLE&c_3=&search_without_file=YES&search_in_typdoc[0]=ART_ACL&search_in_typdoc[1]=ART_SCL&search_in_typdoc[2]=COMM_ACT&search_in_t ypdoc[3]= COMM_SACT&search_in_typdoc[4]=THESE&orderby=DATEPROD&ascdesc=DESC .
- On the idea of differential cone-tracing http://en.wikipedia.org/wiki/Cone_tracing which allows for zero-cost antialiasing on volume data by relying on 3D MIP-mapping.
- On the technology of smart texture management determining visible parts on the fly then streaming or generating the data at the necessary resolution. Starting with SGI clipmaps and late variants for very large textured terrains in flight simulators, then generalized as a cache of multiscale texture tiles feeded by on demain production, by Sylvain Lefebvre and al. http://hal.inria.fr/inria-00070783
- Cyril Crassin connected all these ideas together during his PhD to found the basis of the Gigavoxels system [refs].

Final word

L'idée du code Gigavoxel est qu'on peut faire à la fois simplement les choses simples, mais aussi des choses plus complexes si voulu. Il y a donc un certain nombre de méthodes user-define en CUDA pour fabriquer des données (l'idée est que le moteur ne fabrique les données que quand il est sûr qu'on va les voir, c'est le secret pour avoir des scènes immenses et détaillées qui tiennent en mémoire et s'affichent en temps reel), pour faire le shading d'un voxel sur des données éventuellement custom (couleur+IR+UV, en astronomie), ... voire pour faire tout le rendu autrement (voire même sans voxels du tout !). Dans la doc developpeur, on veut fournir un certain nombre d'exemples de ce qu'on peut faire, de simple ou d'inattendu, pour illustrer les différents usages (i.e. ce n'est pas un soft de visu scientifique), faire envie... et parce que les codeurs ne lisent jamais les manuels mais copient-collent les exemples. On en a fait un certain nombre, mais il en manque d'importants. Par ailleurs, on a des pistes pour des usages "avancés" (par exemple, générer des VBO OpenGL avec Gigavoxel, qui servirait alors de structure de visibilité), et des améliorations (par exemple, une gestion des priorités et du temps disponible dans la fabrication "à la demande" des données, pour tenir le temps réel tout en ayant la meilleure qualité possible).