Animating ultra-complex voxel scenes through shell deformation

Daniel Rios Pavia, Cyril Crassin

Executive Summary Report

I did my TER stage in INRIA Grenoble which is one of the eight research centres run by INRIA, the French National Institute for Research in Computer Science and Control. During my stage I worked in ARTIS, a computer graphics team with many different projects that range from lighting simulation to expressive rendering. One of these projects is Gigavoxels.
Gigavoxels is a highly parallel voxel rendering engine by Cyril Crassin and Team ARTIS that has achieved real-time performance by using intelligent techniques in data streaming. My subject in the stage was to work on the animation for this engine.
For some years now triangles have been adopted as the main primitive in 3D real-time rendering. Recently with the advance of technologies, voxels, a 3D analogy of a 2D pixel, are gaining importance as they have become a viable option with great advantages on rendering and anti-aliasing. The problem in the past was that voxels were too heavy to handle (Easily voxel models can rise up to many GB in size). Furthermore, animation of a huge amount of voxels in real-time were non existent.
Nowadays, the Gigavoxels framework has been able to address the memory problem with a system of stream of data depending on visibility and occlusion information. The engine already has been able to achieve real-time rendering performance of billions of voxels and given the newly arised interest on characters made out of them, the framework is of great importance for future technologies in 3D rendering.
My job in the TER was to work on the animation to be added to the Gigavoxels frameworks. We decided to approach this problem by relying on a shell-based deformation technique to animate voxel data. The following report explains these concepts, as well as how they are going to be used to achieve our objective.
We begin with an introduction, followed by an explanation of our approach to animation. Afterwards we give the state of the art, concepts and techniques used on the work and then we explain with more detail the steps to produce the animation as well as a description of our implementation. Finally we conclude and mention future work on the subject.

Acknowledgements I'd like to thank Cyril for the opportunity and all the help he gave me, Fabrice for keeping an eye on me, Hedlena and Olivier for inspiring me to work hard, and Adrian and Franck for their wise words.

TER: Animating ultra-complex voxel scenes through shell deformation

Daniel Ríos Pavía[1] and Cyril Crassin[2]

[1] Université Joseph Fourier
[2] Université Joseph Fourier / INRIA / LJK

Abstract. Voxel representations have many advantages, such as ordered traversal during rendering and trivial very decent LOD through MIPmap. Special effect companies such Digital Domain or Rhythm&Hues now extensively use voxels engines, for semi-transparent objects such as clouds, avalanches, tornado or explosions, but also for complex solid objects. Several gaming companies are also looking into voxel engines to deal with ever more complex scenes but the main problem when dealing with voxel representations is the amount of data that has to be manipulated. This amount usually prevents animating in real time. To solve these issues, ARTIS team developed the Gigavoxels framework: a very powerful voxel engine based on GPU ray-casting, with advanced memory management, so that very complex scenes can be rendered in real-time. The purpose of the TER was to develop a solution for animating voxel objects in real-time, implement it and eventually integrate it to the Gigavoxels framework.

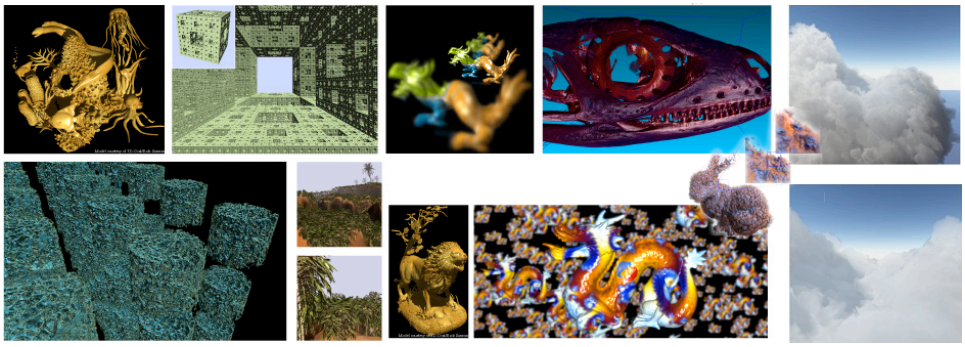

Fig. 1. Example images rendered using the GigaVoxels framework.

1 Introduction

Nowadays CG scenes are very complex, not only in special effects but also in games. On one side, scenes can be landscape-wide or more. On the other side, the details of CG models can go up to several dozen of triangles per pixel, and complex materials such as fur, hair, foliage, requires many triangles. To render them efficiently and avoid aliasing, voxels can be used.

Voxel: volume element representing a value on a regular grid in three dimensional space. This is could be seen as an analogy to a pixel which represent a value in two dimensional space. Common uses of voxels include volumetric imaging in medicine, representation of terrain in games and simulations. Recently there is newly arisen interest in them for adding volume details on characters given the voxel properties.

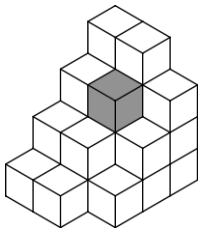

Fig. 2. Voxels

Voxels have many advantages, whe rendering they can be traversed in an orderly fashion, they are geometrically intuitive as well as it is possible to get a decent LOD with not much effort.
The problem with voxels is that the required memory is so huge that the rendering and the animation are still very slow. One character model can go up to 32 GB of memory. To solve these issues, the Gigavoxel framework was developed: a very powerful voxel engine based on GPU ray-casting, with advanced memory management, so that very complex scenes can be rendered in real-time. In this paper we are concerned on the animation problem. For this, it was necessary to prepare an approach that could be rendered in realtime and program it using CUDA [NVIDIA, 2010] and C++.

2 Our Approach to Animation

Our planned strategy for addressing the animation problem is by voxel deformations. A rough draft of our solution is having a voxel model and a coarse mesh of a fine detail model (see Figure 3) . We use shell maps [Porumbescu et al., 2005]

on the coarse mesh to generate a shell space where we can map voxels of our model and render it. This results in the possibility of moving the coarse mesh with current animation techniques and thus animation through voxel deformation. We consider the voxel model as a volumetric texture as it won't be modified at all during the procedure, instead the coarse mesh is modified and using the mapping we deform the voxels.

Fig. 3. Mesh model with very fine details [YECK - angle6308@hotmail.com]

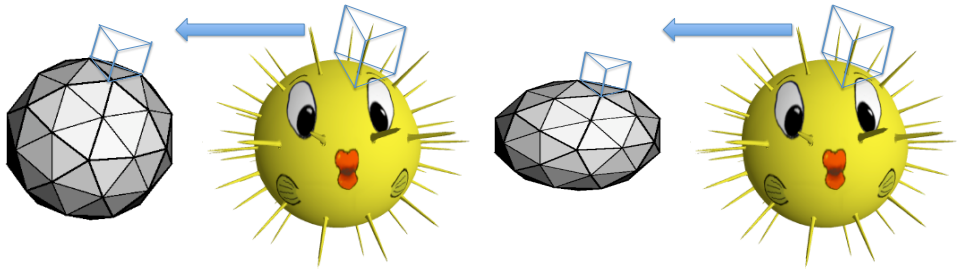

Fig. 4. On the left we map a prism from the object voxels to a prism on the coarse mesh. On the right we change the coarse mesh, the mapping is still valid so we deform our voxels thus obtaining animation.Credits to [Jeschke et al., 2007] for fish image.

3 State of the Art

3.1 Volume Deformations

There is recent work on volume deformations by Pixar Animation. They build a cage around the volume making it possible to deform it with harmonic coordinates [Joshi et al., 2007]. Our approach is different as we start from a coarse mesh of the voxel model but it served as inspiration to this work.

3.2 Voxels

Voxel have often been used for scientific data visualization and for producing great special effects. There are companies such as Digital domain, Cinesite, or Rhythm & Hues that use these voxel engines [Kisacikoglu, 1998], [La Barge et al., 2003], [Kapler, 2003] and [Krall and Harrington, 2005] to render very complex scenes. We can see some examples in recent films like XXX, Lord of the Rings, The Day After Tomorrow, Pirates of the Caribbean, The Mummy 3 where such effect are used. Clouds, smoke, foam, and even extremely detailed geometric data (e.g., boats in Pirates of the Caribbean) are all represented with voxels and rendered via volume rendering. Also, voxels can be found in video games as it gives the possibility of scenarios where the height field cannot help (e.g. caves).

3.3 Gigavoxels

Gigavoxels rendering system is a realtime voxel engine introduced by [Crassin et al., 2009]. GigaVoxels is based on a lightwise sparse voxel octree data structure, a fully GPU voxel ray-caster that provides very high quality and real time rendering and a data streaming strategy based visibility informations provided directly by ray-casting. 3D Mip-Mapping is used to provide very-high filtering quality and the out-of-core algorithm allows virtually unlimited resolution scenes rendering. GigaVoxels is entirely implemented on the GPU using the CUDA programming model [NVIDIA, 2010]. CUDA allows direct access to the GPU computing

Fig. 5. Voxel effect used on recent films. On the left 'The Day After Tomorrow'. On the right 'Lord of the Rings'.

ressources and offers the flexibility to implement any parallel algorithm. One of the future works of Gigavoxels is on animation and in this paper we approach the problem using voxel deformation.

3.4 Shellmaps

'Shell map is a bijective mapping between shell space and texture space that can be used to generate small-scale features on surfaces using a variety of modeling techniques'[Porumbescu et al., 2005]. The shell maps consist on the generation of an offset surface from an original mesh creating a shell space where texture coordinates are mapped. The generality of them allows texture space to contain geometric objects, procedural volume textures, scalar fields, or other shell-mapped objects.
The problem with shell maps is that ray tracing linearly through the shell space would map to a curved ray in texture space. There are different approaches to solve this rendering problem like using a tetrahedral mesh as approximation to the curved ray [Porumbescu et al., 2005] or smooth mapping [Jeschke et al., 2007].

We will use shell maps to map and deform the voxels.

3.5 A-buffer

In order to efficiently implement shell maps, one of the need is to be able te retrieve the list of tetrahedron visible on each pixel of teh screen. To do that efficiently, our approach relies on rasterisation of the triangle faces of the tetrahadron, and on an A-Buffer [Carpenter, 1984] keeping information for each triangle.
Basically an A-buffer is a simple list of fragments per pixel [Carpenter 1984].

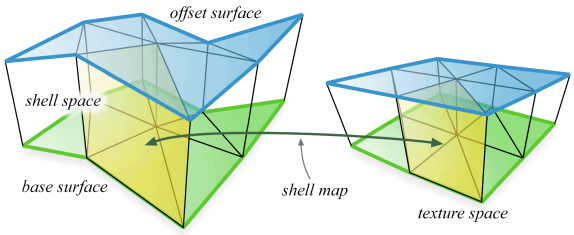

Fig. 6. Shell maps, mapping from shell space to texture space (the object's voxels). [Porumbescu et al., 2005]

Previous methods to implement it on DX10 generation hardware required multiple passes to capture an interesting number of fragments per pixel. They where essentially based on depth-peeling, with enhancements allowing to capture more than one layer per geometric pass, like the k-buffer, stencil routed k-buffer. Bucket sort depth peeling allows to capture up to 32 fragments per geometry pass but with only 32 bits per fragment (just a depth) and at the cost of potential collisions. All these techniques were complex and especially limited by the maximum of 8 render targets that were writable by the fragment shader.
Since our approach relies entirely on the CUDA pipeline, we relied on [**?**] to implement the triangle rasterisation and A-buffer.

4 Implementation

The implementation can be separated and explained in different steps:

1. Shell maps and Tetrahedral generation
2. Per-pixel texture coordinates extraction by using an A-Buffer
3. Ray tracing the volumetric texture
4. Animation of the coarse mesh

4.1 Shell maps and Tetrahedral generation

Having a coarse mesh of our voxel model to animate we can use Shell maps to do a mapping between the voxels and the generated shell space.
For rendering, a launched linear ray on shell space will map to a curved ray on texture space. To solve this at the moment there are two approaches. The first one is the construction of a tetrahedral mesh that fills the space in between the original and offset surface[Porumbescu et al., 2005]. Afterwards, for each tetrahedra on the mesh we identify a corresponding one in texture space and with straightforward barycentric-coordinates, we can do a mapping between them thus obtaining shell maps. This approach leads to artifacts at the border of the tetrahedras. The second approach happens directly to the prisms and it

consists on marching the ray stepwise linearly in texture space and correct it's direction each time[Jeschke et al., 2007]. This will give a better approximation to the curved ray eliminating the artifacts but it is slower. We chose the tetrahedral mesh as we believe it is more suited for real time rendering and not such a complicated solution for a prototype.

Another intrinsic problem of shell maps is when deforming or animating the

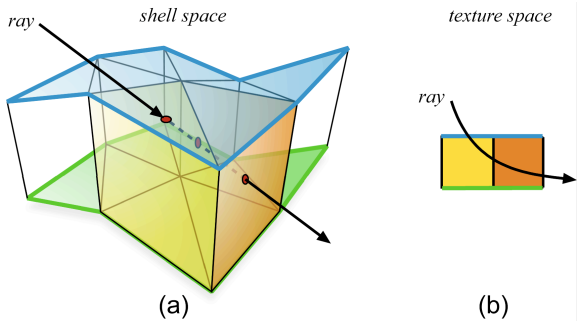

Fig. 7. Linear ray in shell space is a curved ray on texture space [Porumbescu et al., 2005].

coarse mesh, it will arise concavities that will make the shell space smaller. This is because the offset surface will change to prevent self intersections and preserve the bijectivity. For the prototyping purposes we don't need the bijection thus we decided to use non bijective shell maps, meaning we will use constant offset and self intersections are allowed.

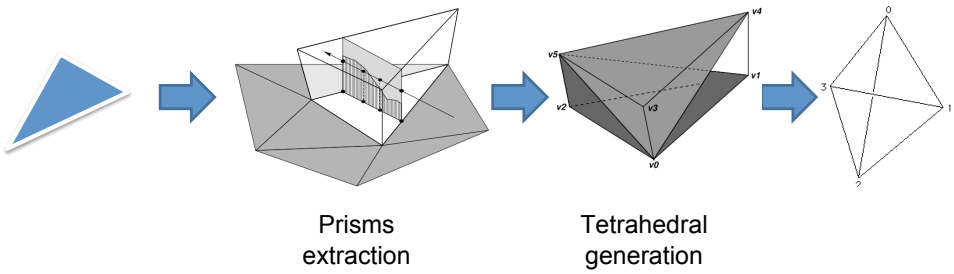

Fig. 8. Pipeline to obtain the tetrahedra. Images from [Hirche et al., 2004].

4.2 Per-pixel texture coordinates extraction by using an A-buffer

After obtaining the tetrahedral mesh, the next step is obtaining the texture coordinates of ray intersection with the tretrahedras to be used at each pixel of the screen. For this step there were two options, a ray intersecting method or a rasterization A-buffer method. The ray intersection requires to test all triangles for each pixel while the rasterization would only need to process each triangle once thus we've chosen the later to obtain the texture coordinates.

An A-buffer used for depth peeling consist of rasterizing the primitives of a mesh and storing all the depth values of the generated fragments in a buffer per pixel, afterwards depending on the depth the buffer is sorted and thus a transparency effect in rendering can be obtained.

Fig. 9. Transparency using the A-Buffer [Liu et al., 2009].

We used a modified version of the CUDA algorithm proposed for A-buffer single-pass depth peeling in [Liu et al., 2009], in which we rasterize each triangle of the tetrahedras, store the depth and texture coordinates of the fragments in the A-buffer and then sort them using the depth. This will allow us to obtain the ordered list of texture coordinates for each pixel that we will use to render the scene by using the optimized Gigavoxels framework ray tracer.

4.3 Ray casting the volumetric texture

Now it is possible to do ray casting for each pixel of the screen by using the A-buffer with the ordered stored texture coordinates.
It consists on launching a linear ray per pixel between each stored texture coordinate of it's corresponding buffer.

The idea is to integrate the optimized Gigavoxels framework ray casting with the generated A-buffer to render the scene.

Our base prototype relies on a simple volume ray-casting based on a sample from the NVIDIA CUDA SDK [NVIDIA, 2010].

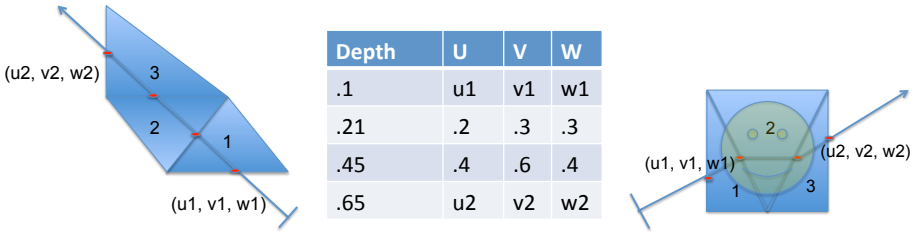

Fig. 10. On the left a ray/projection from a pixel to get the an A-buffer (Center) with the texture coordinates. On the right we use the A-buffer for approximating the curved ray on texture space.

4.4 Animation of a coarse mesh

Finally, at each keyframe of the animation of the coarse mesh it is needed to extrude the vertices using their normals to generate the offset surface. Supposing the order and number of vertices for each keyframe of the animation remain the same, it is not necessary to recompute the tetrahedras as the stored indices and information can remain the same. If this condition is not met then tetrahedras need to be regenerated each keyframe.

5 Implementation and Results

5.1 Tetrahedral generation

For this I used Nvidia nvModel loader for OBJ files which contain mesh information, then extrude the vertices using their normals to generate the offset surface, and then computing the tetrahedras with the ripple algorithm described with detail in [Erleben et al., 2005]. Until now the implementation is using OpenGL and C++.

5.2 A-buffer

My contribution in this part was modifying the implementation in CUDA of the algorithm proposed by [Liu et al., 2009] to get an ordered list of texture coordinates per pixel as well as correcting the bugs from the original version: memory overflow of the A-buffer and wrong fixed point precision for the razterizer. I also

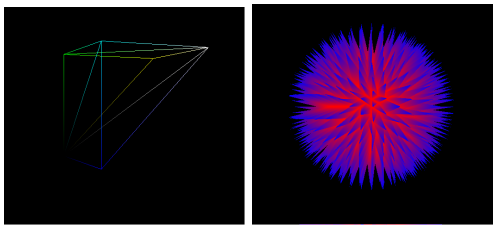

Fig. 11. On the left, tetrahedra generation for a triangle. On the right, bottom tetrahedra generation for a sphere.

had to implement perspective correct interpolation of the texture coordinates for the razterizer. I Integrated as well the tetrahedral generation to this part of the code

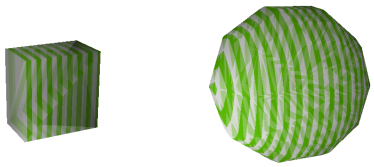

Fig. 12. On the left, A-buffer for a quad tetrahedra. On the right, A-buffer for sphere tetrahedra.

5.3 Ray casting

For testing purposes I implemented a simple parallel ray caster in CUDA that used the A-buffer to render the scene. I also integrated with the A-buffer and the tetrahedral generation implementations.

5.4 Animation of a coarse mesh

For this part of the procedure, I investigated different methods for animating a coarse mesh to test the deformations. FBX and Collada methods would take time and effort to get results so for the moment I opted for a simple array of .OBJ files that would load and give the effect of animation. This part is yet to be integrated to the rest of the code.

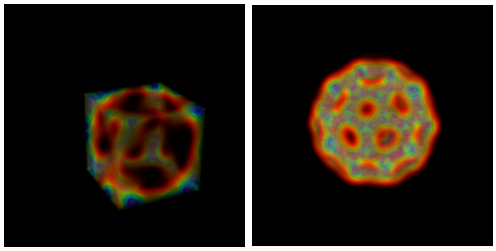

Fig. 13. On the left, raycasting for quad. On the right, raycasting for sphere.

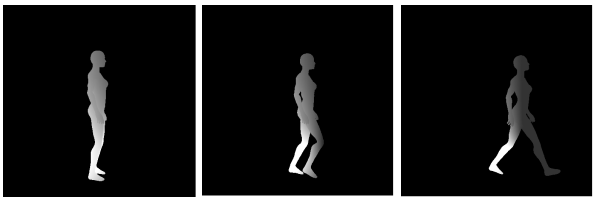

Fig. 14. Animation of OBJ files.

6 Conclusion and Future Improvements

Until now the current ideas for animation by voxel deformation look like a viable option that could lead to an article in the future. The implementation is close to be finished as a demonstration to the procedure described in this paper, although there are still lots of optimizations to be done:
The tetrahedral generation could be done in CUDA. Also the A-buffer is not as fast in CUDA for obtaining the texture coordinates as it could be in OpenGL, the dedicated rasterization hardware from the GPU is not used here. It is now possible with the next generation of NVIDIA GPUs to make random read/write access into arbitrary buffers, which allow to take advantage of the optimized rasterization of the video card and to obtain a faster A-buffer [Crassin, 2010]. We have an implementation of A-buffer using these new features which will serve to compare with our CUDA implementation. For the ray casting part, it needs to be integrated into the Gigavoxels and the Animation, there are many other better ways to implement it other than using a series of OBJ files like collada and FBX.

Acknowledgments We would like to thank Meng-Cheng Huang for the CUDA rasterizer and Brian Budge for the tetrahedra generation base codes they provided us.

References

[Crassin, 2010] Crassin, C. (2010). Icare3d. fast and accurate single pass buffer. `http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html`.

[Crassin et al., 2009] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. (2009). Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22, New York, NY, USA. ACM.

[Erleben et al., 2005] Erleben, K., Dohlmann, H., and Sporring, J. (2005). The adaptive thin shell tetrahedral mesh. In *WSCG (Journal Papers)*, pages 17–24.

[Hirche et al., 2004] Hirche, J., Ehlert, A., Guthe, S., and Doggett, M. (2004). Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of Graphics Interface 2004*, pages 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. Canadian Human-Computer Communications Society.

[Jeschke et al., 2007] Jeschke, S., Mantler, S., and Wimmer, M. (2007). Interactive smooth and curved shell mapping.

[Joshi et al., 2007] Joshi, P., Meyer, M., DeRose, T., Green, B., and Sanocki, T. (2007). Harmonic coordinates for character articulation. *ACM Trans. Graph.*, 26(3):71.

[Kapler, 2003] Kapler, A. (2003). Avalanche! snowy fx for xxx. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1, New York, NY, USA. ACM.

[Kisacikoglu, 1998] Kisacikoglu, G. (1998). The making of black-hole and nebula clouds for the motion picture "sphere" with volumetric rendering and the f-rep of solids. In *SIGGRAPH '98: ACM SIGGRAPH 98 Conference abstracts and applications*, page 289, New York, NY, USA. ACM.

[Krall and Harrington, 2005] Krall, J. and Harrington, C. (2005). Modeling and rendering of clouds on "stealth". In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 85, New York, NY, USA. ACM.

[La Barge et al., 2003] La Barge, B., Tessendorf, J., and Gaddipati, V. (2003). Tetrad volume and particle rendering in x2. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1, New York, NY, USA. ACM.

[Liu et al., 2009] Liu, F., Huang, M.-C., Liu, X.-H., and Wu, E.-H. (2009). Single pass depth peeling via cuda rasterizer. In *SIGGRAPH '09: SIGGRAPH 2009: Talks*, pages 1–1, New York, NY, USA. ACM.

[NVIDIA, 2010] NVIDIA (2010). *CUDA Programming guide 3.0*.

[Porumbescu et al., 2005] Porumbescu, S. D., Budge, B., Feng, L., and Joy, K. I. (2005). Shell maps. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 626–633, New York, NY, USA. ACM.