GPU-BASED CELL PROJECTION FOR LARGE STRUCTURED DATA SETS

André Maximo, Ricardo Marroquim, Ricardo Farias, Claudio Esperança
Universidade Federal do Rio de Janeiro - UFRJ / COPPE, Brazil
{*andre, ricardo, rfarias, esperanc*}*@lcg.ufrj.br*

Keywords: Direct volume rendering, gpu programming, cell projection, large volumetric data.

Abstract: We present a practical implementation of a cell projection algorithm for interactive visualization of large volumetric data sets using programmable graphics cards. By taking advantage of the data regularity, we can avoid computing some steps of the original algorithm with no quality loss. Furthermore, performance is increased since more than half the processing time is dedicated only for rendering. We also provide two tools for better user interactivity, one for transfer function editing and another for volume clipping. Our algorithm generates high quality images at a rendering speed of over 5.0 M Tet/s on current graphics hardware.

1 INTRODUCTION

A challenging problem in medical and scientific visualization is the interactive rendering of large 3D data sets. In medical imaging, for example, most data sets are regular (or structured) grids, usually ranging between 256^3 and 512^3 *voxels*. An interactive visualization system is very important when analysing these data sets. Efficiency is crucial for time-critical diagnostics, as well interactive manipulation of the volume data.

On the other hand, irregular (or unstructured) grids are typically used in scientific visualization such as geological inspection, fluid simulation, among others applications. Nevertheless, by applying a preprocessing step, irregular data can be adapted to a regular grid. This process, known as voxelization (Kaufman and Shimony, 1986) (Prakash and Manohar, 1995), typically undersamples or oversamples some portions of the volume. Better sampling strategies have been proposed by (LaMar et al., 1999) (I. Boada, 2001) to improve the quality of the output grid.

Conversely, approaches capable of handling irregular grids, such as the Projected Tetrahedra (PT) algorithm, can be easily applied to structured data, since uniformly sized voxels may be regarded a special case of the more general irregular grids. This, however, is generally a bad idea since much of the computation in such algorithms is dedicated to classifying and organizing voxels.

We present an implementation of the PT algorithm proposed by (Shirley and Tuchman, 1990) in a specific context, where projection and sorting computations costs are virtually eliminated. More than 60% of the time is dedicated only to the rendering pipeline.

The remainder of this paper is organized as follows: we discuss previous work in Section 2, followed by a quick overview of the PT algorithm in Section 3. We then describe our approach in Section 4. In Section 5 we present two tools for interactive manipulation of the volume data. Results are given in Section 6, and some conclusions are drawn in Section 7.

2 PREVIOUS WORK

There are many different approaches for direct volume rendering in the literature (Kaufman and Mueller, 2005): cell projection, ray casting and splatting. With the availability of programmable cards, new algorithms were proposed in order to exploit Graphics Processing Units (GPUs) for interactive volume rendering.

Many cell projection algorithms have been proposed for volume rendering on the GPU. Our ap-

proach is a variation of the GPU-based implementation of the Projected Tetrahedra technique (Marroquim et al., 2006).

The first PT implementation on GPU, called GATOR, was proposed by (Wylie et al., 2002). The GATOR algorithm is fast yet redundant, since for every vertex computation on the GPU all other tetrahedron's vertices must be made available.

One major drawback of the PT-based algorithms is the need of a visibility sorting of the cells. This problem has been specifically addressed on many works, e.g. (Williams, 1992) (Stein et al., 1994) (Comba et al., 1999), but it is not focused in this paper since sorting voxels of a regular grid is straightforward.

The Hardware-Based Ray Casting (HARC) algorithm (Weiler et al., 2003a) achieves high quality and fast volume rendering using the GPU. However, the memory consumption of the HARC algorithm is high compared to other cell projection algorithms. A hybrid solution between cell projection and ray casting is the View Independent Cell Projection (VICP) (Weiler et al., 2003b). By performing ray casting only inside each projected cell, the VICP achieves high quality image while consuming less memory than HARC.

The main idea of the splatting algorithms, introduced by (Westover, 1990), relies on the image area influenced by each 3D data set voxel. This area, know as *footprint*, is a 2D map of 3D reconstruction kernels of the volume. The extension of Elliptical Weighted Average (EWA) volume splatting (Zwicker et al., 2001) proposed by (Chen et al., 2004), accelerates the rendering by storing the splat and volume data on the GPU memory. The main drawbacks of this method are the aliasing and blurring effects in the final image.

Level-of-Detail (LOD) techniques may also be used in conjunction with volume rendering algorithms. The idea is to undersample the 3D data set in order to reach a specific frame rate. For instance, in the work of (Callahan et al., 2005), interactive frame rates are achieved by lowering the quality of the image.

The work of (Roettger et al., 2003) combines ray casting and the exploit of spatial coherence. Based on the idea of (Danskin and Hanrahan, 1992), performance is improved by early ray termination and space leaping. Pre-integration techniques are used to obtain high quality images.

In this paper, we use the partial pre-integration technique proposed by (Moreland and Angel, 2004) to obtain high quality images while allowing interactive transfer function editing. Additionally, we adapt the concept of footprint from splatting. In particular, we consider one unit of the volume data, i.e. a hexahedron, as a footprint. This means that the cell projection is only computed for one hexahedron and replicated to the rest of the volume. With this, the computational cost is concentrated on the actual rendering of the triangles.

3 PROJECTED TETRAHEDRA ALGORITHM

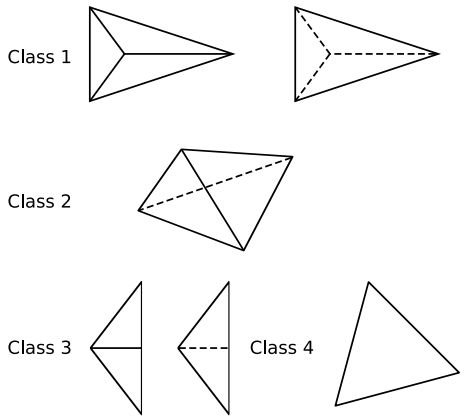

Figure 1: The different classifications of the projected tetrahedra.

The projection in our approach is based on the PT algorithm of (Shirley and Tuchman, 1990), where the tetrahedra are projected to screen space and composed in visibility order. The tetrahedra's projected shapes are classified depending on the different number of generated triangles (see Figure 1). With this classification, the colors and opacity values of the triangle vertices are evaluated. An approximation of the ray integral is then used to compute the color of each triangle fragment. The final pixel color is computed by summing all the fragment contributions in back-to-front order.

For each projection, the *thick* vertex is defined as the point of the ray segment that traverses the maximum distance through the tetrahedron. All other projected vertices are called *thin* vertices, as no distance is covered. For class 2 projections (see Figure 1), the thick vertex is computed as the intersection between the front and back edges, while for the other classes it is one of the projected vertices. The scalar values of the ray's entry and exit points are named as s_f and s_b (see Figure 2). Thin vertices have the same values for s_f and s_b, while for thick vertices these values may have to be interpolated from those of the thin vertices. The distance traversed by the ray segment is defined

as the thickness l of the cell.

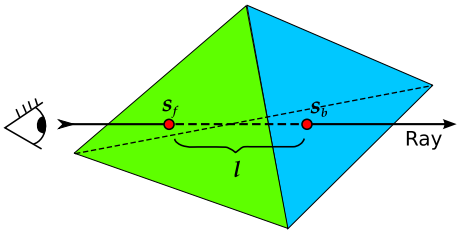

Figure 2: The ray integration parameters inside one tetrahedron.

Three values – s_f, s_b and l – are interpolated inside each triangle of the projected tetrahedron. To compute the fragment color, the original PT algorithm computes the average between s_f and s_b. The RGB color and τ are then retrieved from the transfer function by this average scalar. On the other hand, the opacity (α) value is evaluated as $\alpha = 1 - e^{-\tau l}$, where τ is the extinction coefficient and l is the interpolated thickness for that fragment.

Finally, the fragments are composed in back-to-front order. For each new color added to the frame buffer, the new final color is computed as $C_{fb} = \alpha C_{new} + (1-\alpha)C_{fb}$. C_{fb} is the color already in the frame buffer, while C_{new} and α are the new color and opacity values computed for the fragment.

4 ALGORITHM OVERVIEW

The main idea of this work is to use the PT algorithm taking advantage of the volume data regularity. The algorithm consists of four steps, where the first three steps take place in the CPU, whereas the last is performed by the GPU (see Figure 3). First, the projection of a single hexahedron is determined by splitting it into five tetrahedra. Then, a traversal order for the whole volume is determined in constant time. The remaining step on the CPU consists of allocating the volume information in a Vertex Array structure. Lastly, triangles corresponding to the projected tetrahedra are rendered and composited in back-to-front order using the GPU.

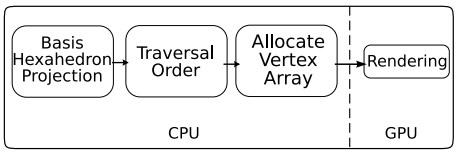

Figure 3: Algorithm overview.

4.1 Basis Hexahedron Projection

Since the PT algorithm requires a tetrahedral mesh as input, the volume data must be preprocessed by subdividing each hexahedron into five tetrahedra. We term each such set of tetrahedra a *volume unit* or *volunit* for short. A key observation is that, by rendering the volume in orthographic projection, all *volunits* are projected to the screen in exactly the same way. Thus, to avoid redundant computation, the projection parameters are computed only once for a *basis hexahedron*.

Each *basis tetrahedron* is projected to screen coordinates and its projection class is determined by means of four cross-product tests, refer to (Marroquim et al., 2006) for more details. Then, the thick vertex and the front and back scalar values can be computed with at the most two segment intersections (depending on the class).

For each of the five *basis tetrahedra* the following projection values are computed and stored:

- *basis projection class*,
- *basis projected vertices coordinates*,
- *basis thick vertex coordinates*,
- *basis intersection parameters* for computing the front and back scalar values,
- *basis rendering order*.

4.2 Rendering

To render the volume data, each tetrahedron is rendered as a triangle fan. The primitives are drawn in back-to-front order and the resulting pixel fragments are composed using a blend function. The *basis hexahedron* is iteratively displaced to the position of each *volunit*, and the stored *basis projected vertices* are used to compose the tetrahedra triangles. Each triangle fan is rendered with the number of triangles relative to its *basis projection class* following the *basis rendering order*. The first vertex of the fan, its central node, is the *basis thick vertex*.

Even though the geometry can be resolved by just displacing the *basis hexahedron*, the colors and opacity values are unique for each vertex and must be computed on the fly for each *volunit*. In fact, the final color is computed only in the GPU's fragment shader. The values passed as the vertex color to the rendering pipeline are its front and back scalar values and its thickness.

For every vertex other than the thick vertex, the scalar front and back values are the same (the original scalar of the volume data). Furthermore, their thickness value is always zero since the ray traverses no length of the tetrahedron at these vertices. On

the other hand, the scalar values for the thick vertex are calculated using the *basis intersection parameters*, while the *basis thickness* was already computed for each *basis tetrahedron*. Figure 4 illustrates a rendered triangle with color values.

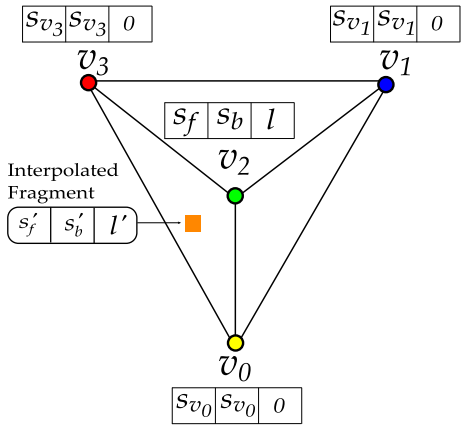

Figure 4: The color values for a class 1 projection case.

4.2.1 Vertex and Fragment Shaders

Vertex coordinates are computed on the fly by the vertex shader based on the 3 integer indices of the vertex in the lattice. This computation requires several parameters previously determined for the *basis hexahedron* and passed to the shader as global values, or "uniforms" in GLSL parlance.

Each grid vertex is rendered multiple times, once for each incident tetrahedron. Thus, along with the vertex grid coordinates, additional information is coded in the vertex normal, namely the tet_{id} and the $vert_{id}$. The tet_{id} is stored in the x normal coordinate and identifies which of the five tetrahedra is being rendered. The $vert_{id}$ is stored in the y normal coordinate and identifies the vertex of the tetrahedron. It should be noted that this coding scheme shifts most of the computational load from the CPU to the GPU.

The fragment shader receives trilinearly interpolated vertex colors (sf, sb, l) for each triangle fragment. The interpolated scalar values are used to lookup the chromaticity and opacity values in a transfer function table. This table is stored in a 1D texture with 256 positions. For each scalar value the transfer function texture is accessed to determine its correspondent RGBA color.

The Figure 5 summarizes the algorithm pipeline in the GPU. Each volume hexahedron is sent to the GPU as five triangle fans, while each fan corresponds to one tetrahedron projection. The vertex shader uses the *basis hexahedron* information, stored globally as uniform variables, to correctly displace each vertex. The rasterization process interpolates the color information inside each triangle (see Figure 4), and the final fragment color is computed.

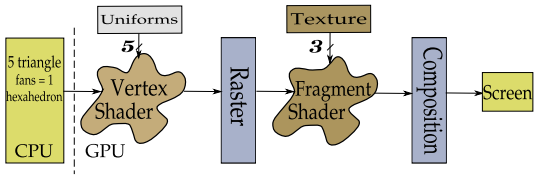

Figure 5: Algorithm pipeline. The 5 projected basis vertices, including the thick vertex, are read in the vertex shader. While the fragment shader reads 3 textures: exponential, transfer function, and partial pre-integration.

4.2.2 Fragment Color Computation

Given the front and back colors and thickness values, there are some options for determining the final fragment color. The simplest and fastest way is to compute the chromaticity value as the average between the front and back colors. The opacity is computed as $\alpha = 1 - e^{-\tau l}$, where τ is the mean alpha value, and l is the thickness. Instead of computing the exponential value on the fly, a faster way is to use another 1D texture, namely exponential texture, as a lookup table for e^{-u}, with u sampled over interval $[0,1]$.

Another way to compute the final color is by actually integrating the colors along the ray traversing the tetrahedron. However, in practical terms, this is infeasible. The pre-integration method (Roettger et al., 2000) is a way to achieve this result by storing the integral values for different $\{s_f, s_b, l\}$ in a table. A single lookup operation returns the final fragment color. Nevertheless, this table is computed using the transfer function implicitly, and must be recomputed every time it changes. Since generating the pre-integration table is an expensive operation, this cannot be done in interactive times.

To overcome this disadvantage, the partial pre-integration approach introduced by (Moreland and Angel, 2004) computes a table independent of the transfer function. The so-called ψ table does not depend on any attribute of the visualization, therefore it is pre-compiled within our implementation.

In the fragment shader, the colors associated with s_f and s_b are retrieved from the transfer function texture. The C_f and C_b together with the thickness value l are used to compute the indices of the ψ table, stored in a 2D texture. The value retrieved from this table is then used to compute the final fragment color.

In contrast with the original PT method, the partial method is slower than computing the final color using

the scalar front and back average. Moreover, the use of partial pre-integration, instead of pre-integration, allows interactive transfer function editing. Every time the function is updated, the texture is reloaded into the GPU memory.

4.2.3 Optimizing with Vertex Arrays

To achieve even higher frame rates, we make use of the optimized OpenGL function *glMultiDrawElements*. The CPU–GPU transfer load is reduced by keeping some common data among *volunits* stored in the GPU memory. For each *volunit* three arrays are read by this function: a vertex, a color, and a normal array. Each array has fixed size and contains five vertices for each of the five tetrahedra (4 vertices plus the thick vertex).

All coordinates, colors and normals are passed exactly the same way as explained in the last section. However, the vertex and color arrays must be recomputed for each *volunit* while the normal array can be computed only once since it is valid for all *volume units*.

The order of the vertices of the triangle fans must be passed as an array argument to the function. Again, five arrays of indices are created once for all *volunits* with the *basis rendering order*.

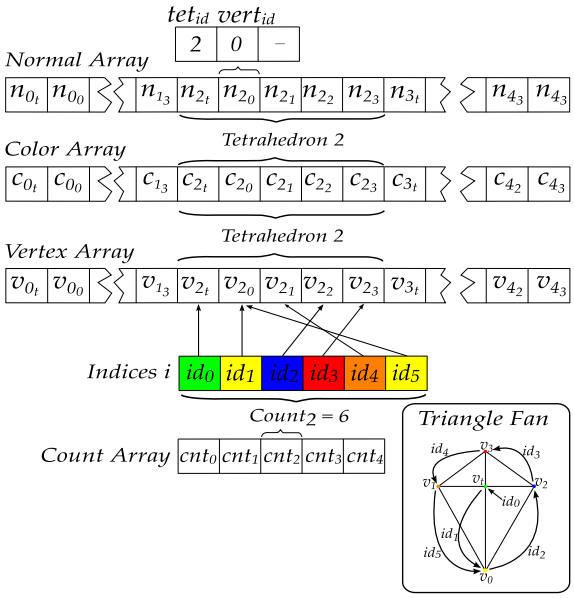

Figure 6: The vertex, color, and normal arrays structure.

Figure 6 illustrates an example of the arrays structure. In this case the third tetrahedron is of class 2 (six vertices).

4.3 Sorting

One disadvantage of the PT algorithm is that the primitives must be rendered in an ordered way, typically back-to-front. Sorting millions of tetrahedra can be achieved in at most O(n) using auxiliary data structures and by performing more costly preprocessing operations (Williams, 1992) (Stein et al., 1994) (Comba et al., 1999). However, we can again profit from the fact that the data is regularly spaced and thus implicitly sorted as a 3D array. All that remains is determining a traversal rule for this data, which can be done in constant and negligible time. In fact, only 8 possible traversal orders are sufficient to render the volume from any possible angle.

Let $\vec{v} = \{v_x, v_y, v_z\}$ be the viewing vector, i.e, a vector pointing to the observer. Then, a positive v_x indicates that the *volunits* must be traversed in ascending order of x indices, whereas a negative v_x would indicate a descending order. A similar rationale may be used for the other axes. Note that the relative order in which the axes are traversed does not matter for regular data, that is, iterating in the order x, y and z will produce the same result as iterating in z, y and x or any other permutation.

5 VOLUME INTERACTION

The interaction with the volume data is improved with two manipulation features: interactive transfer function editing and volume clipping.

The first allows the user to interactively manipulate control points of the transfer function. Every time a change occurs, the transfer function texture is recomputed and uploaded again to the fragment shader. The brightness of the image can also be adjusted and acts as a global opacity factor.

The clipping tool acts directly on the volume image. By selecting rectangular areas, the volume can be trimmed and smaller features enlarged. The only limitation is that the clipping must be parallel to one of the volume's bounding box sides to ensure that the regular properties are maintained.

6 RESULTS

In the previous sections, we have described an implementation of the PT algorithm for regular data using vertex and fragment shaders. Our prototype was programmed in C++ using OpenGL 2.0 with GLSL under Linux. Performance measurements were made on a Intel Pentium IV 3.6 GHz, 2 GB RAM, with a

nVidia GeForce 6800 256 MB graphics card and a PCI Express 16x bus interface.

Some results for different data sets are shown in Table 1. All timings were taken with a 512^2 viewport and considering that the model is constantly rotating. The rotation procedure is important to change between different projection classes, since parallel rendering generates less triangles.

The number of vertices (# Verts) and tetrahedra (# Tet) depends on the dimension of the original regular data set. Performance measures are given in frames per second (fps) and millions of tetrahedra per second (M tet/s) for each data set. In fact, this last column contains two values: the first is the nominal number of tetrahedra per second, including those which do not contribute to the final image, while the second is the effective number, i.e., the tetrahedra actually rendered.

Table 1: Average frames and tetrahedra per second.

Data set	*# Verts*	*# Tet*	*fps*	*M tet/s*
Fuel	262 K	1.2 M	70.78	88.5/5.99
ToothC	1 M	5 M	1.22	13.1/6.30
Tooth	10 M	52 M	0.24	12.7/6.61
Foot	16 M	83 M	0.81	67.7/6.23
Skull	16 M	83 M	0.61	51.7/6.25
Aneurism	16 M	83 M	2.42	201/5.35

We use five different data sets to measure our algorithm (VolVis, 2006). The simulation of fuel injection, computed tomography of a tooth, the x-ray scan of a human foot, skull and aneurism. The row labeled ToothC corresponds to the original tooth tomography clipped with our tool.

The value for tetrahedra per second (*tet/s*) given in Table 1 only counts the tetrahedra which were actually rendered, since *volunits* with zero opacity are discarded. The models rendered with our algorithm are shown in Figure 7.

The timing for the vertex array setup and rendering are given in Table 2. In our algorithm, the average time spent in rendering is more than 60% of the total time. It should be noted that as the time spent in rendering the volume becomes closer to 100%, the algorithm times will be bound by the graphics card performance (Roettger and Ertl, 2003).

Table 2: Setup and render times.

Data set	*Setup*	*Render*	*% Total*
Fuel	0.005 s	0.009 s	64.28 %
Tooth	1.377 s	6.484 s	82.47 %
Foot	4.556 s	9.074 s	66.57 %
Skull	4.606 s	8.593 s	65.10 %
Aneurism	0.199 s	0.210 s	51.34 %

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented a direct volume rendering algorithm, based on the PT method, that takes advantage of the data regularity. The graphics hardware is also explored to increase frame rates up to 6.6 M Tets/s while generating high quality images.

No extra data structures are created other than the volume data itself. The only limitation is that the data set must fit in main memory to avoid swapping. We've also created a clipping interface to easily cut away undesirable parts of the volume, increasing the frames rates and allowing better visualization and interactivity.

Currently, each hexahedron is divided into five tetrahedra, each of which is rendered in the worst case as 4 triangles or a maximum of 20 triangles per *volunit*. As future work we are investigating ideas to render less primitives by *volume units*.

Another improvement being considered consists of enhancing the visualization by mixing in a Phong lighting model (Max, 1995) where the gradient field is used to estimate normal vectors.

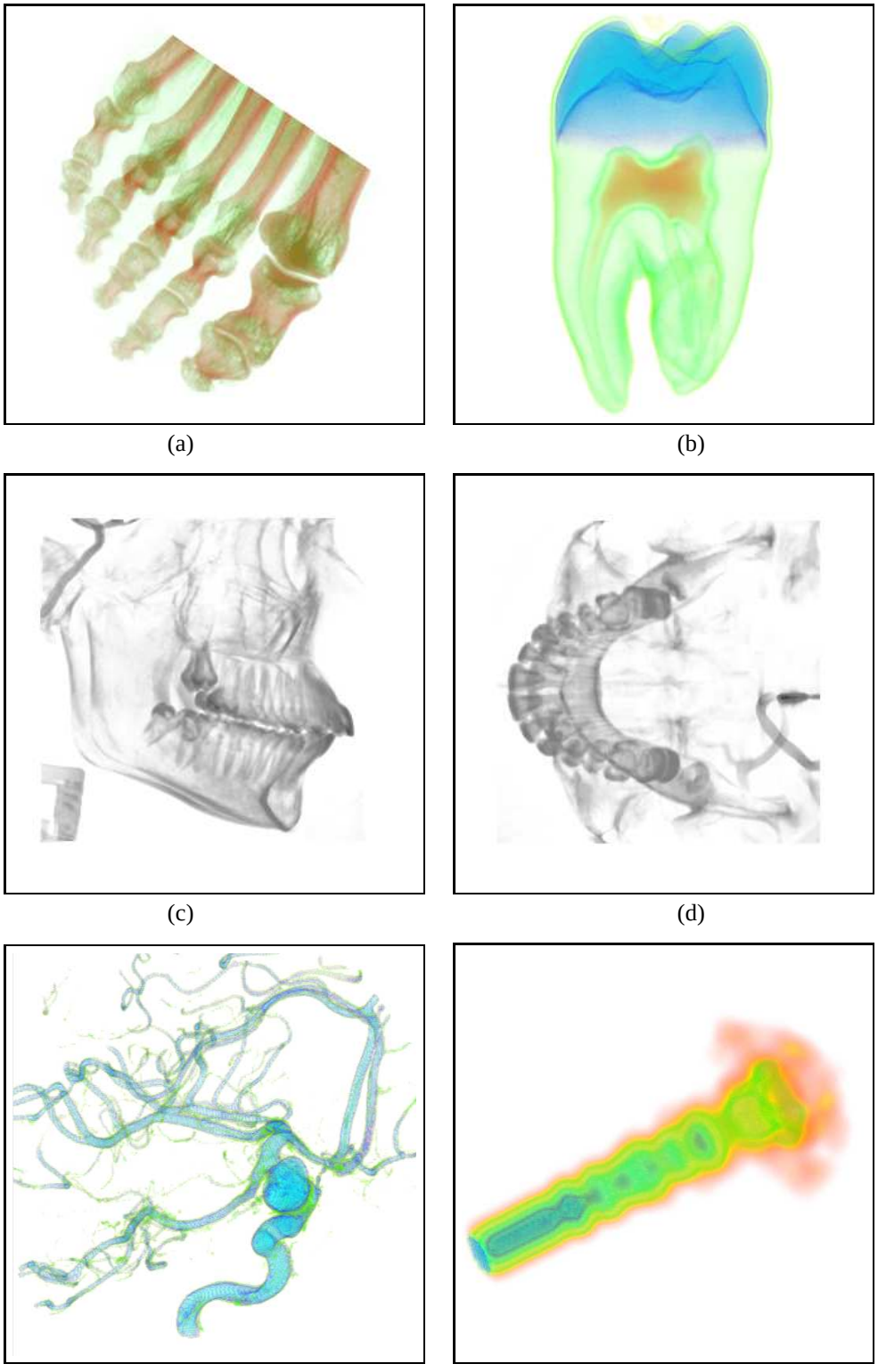

Figure 7: Data sets : Foot (a), Tooth (clipped) (b), Skull (side view) (c), Skull (Bottom view) (d), Aneurism (e), Fuel injection (f).

REFERENCES

Callahan, S., Comba, J., Shirley, P., and Silva, C. (2005). Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05*, ISBN 0-7803-9462-3, pages 199–206.

Chen, W., Ren, L., Zwicker, M., and Pfister, H. (2004). Hardware-accelerated adaptive ewa volume splatting. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 67–74, Washington, DC, USA. IEEE Computer Society.

Comba, J., Klosowski, J. T., Max, N. L., Mitchell, J. S. B., Silva, C. T., and Williams, P. L. (1999). Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):369–376.

Danskin, J. and Hanrahan, P. (1992). Fast algorithms for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, New York, NY, USA. ACM Press.

I. Boada, I. Navazo, R. S. (2001). Multiresolution volume visualization with a texture-based octree. In *The Visual Computer*, pages 185–197. Springer International.

Kaufman, A. and Mueller, K. (2005). Overview of volume rendering. *Chapter for The Visualization Handbook*.

Kaufman, A. and Shimony, E. (1986). 3d scan-conversion algorithms for voxel-based graphics. In *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics*, pages 45–75, New York, NY, USA. ACM Press.

LaMar, E., Hamann, B., and Joy, K. I. (1999). Multiresolution techniques for interactive texture-based volume visualization. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 355–361, Los Alamitos, CA, USA. IEEE Computer Society Press.

Marroquim, R., Maximo, A., Farias, R., and Esperanca, C. (2006). Gpu-based cell projection for interactive volume rendering. *sibgrapi*, 0:147–154.

Max, N. (1995). Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108.

Moreland, K. and Angel, E. (2004). A fast high accuracy volume renderer for unstructured data. In *VVS '04: Proceedings of the 2004 IEEE Symposium on Volume visualization and graphics*, pages 13–22, Piscataway, NJ, USA. IEEE Press.

Prakash, C. and Manohar, S. (1995). Volume rendering of unstructured grid – a voxelization approach. In *Computers Graphics*, pages 711–726.

Roettger, S. and Ertl, T. (2003). Cell projection of convex polyhedra. In *VG '03: Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 103–107, New York, NY, USA. ACM Press.

Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., and Strasser, W. (2003). Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 231–238, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Roettger, S., Kraus, M., and Ertl, T. (2000). Hardware-accelerated volume and isosurface rendering based on cell-projection. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 109–116, Los Alamitos, CA, USA. IEEE Computer Society Press.

Shirley, P. and Tuchman, A. A. (1990). Polygonal approximation to direct scalar volume rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24(5), pages 63–70.

Stein, C., Becker, B., and Max, N. (1994). Sorting and hardware assisted rendering for volume visualization. In Kaufman, A. and Krueger, W., editors, *1994 Symposium on Volume Visualization*, pages 83–90.

VolVis (2006). Volume Visualization. http://www.volvis.org/.

Weiler, M., Kraus, M., Merz, M., and Ertl, T. (2003a). Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE conference on Visualization '03*, pages 333–340.

Weiler, M., Kraus, M., Merz, M., and Ertl, T. (2003b). Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175.

Westover, L. (1990). Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA. ACM Press.

Williams, P. L. (1992). Visibility-ordering meshed polyhedra. *ACM Trans. Graph.*, 11(2):103–126.

Wylie, B., Moreland, K., Fisk, L. A., and Crossno, P. (2002). Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics*, pages 7–12, Piscataway, NJ, USA. IEEE Press.

Zwicker, M., Pfister, H., VanBaar, J., and Gross, M. (2001). Ewa volume splatting. In *IEEE Visualization 2001*.