The Rendering of Unstructured Grids Revisited

Rüdiger Westermann

Scientific Visualization and Imaging Group
University of Technology Aachen

Abstract. In this paper we propose a technique for resampling scalar fields given on unstructured tetrahedral grids. This technique takes advantage of hardware accelerated polygon rendering and 2D texture mapping and thus avoids any sorting of the tetrahedral elements. Using this technique, we have built a visualization tool that enables us to either resample the data onto arbitrarily sized Cartesian grids, or to directly render the data on a slice-by-slice basis. Since our approach does not rely on any pre-processing of the data, it can be utilized efficiently for the display of time-dependent unstructured grids where geometry as well as topology change over time.

1 Introduction and related work

Rendering unstructured volume data is still challenging because no existing algorithm allows for the accurate display of reasonably sized data sets at interactive frame rates. Although considerable efforts have been made during the last couple of years, frame rates are still not competitive with those that can be reached for high resolution Cartesian grids using specialized software solutions or dedicated graphics hardware. In addition, many of the proposed techniques fail in practical applications, where memory issues play a major concern and dynamic changes of geometry as well as topology happen frequently.

In particular two basic problems have to be addressed when developing rendering algorithms for unstructured grids. The first one is to determine the correct visibility ordering of elements. The second one is to implement an appropriate algorithm that allows one to render each element in the ascertained order.

Two different classes of algorithms exist to solve for the latter problem as illustrated in Figure 1. Image space techniques compute for each view ray the entry and the exit point for every element that is hit by that ray. At both points the data is interpolated between the values given at the elements vertices. Finally, taking into account an optical model the integration along the ray is performed. If pre-shaded samples are used for interpolation it can be exploited that the resampled signal along the ray depends linearly on the values at the entry and the exit point. In post-shading, however, the scalar field has to be reconstructed along the ray by taking an appropriate step size with respect to the selected transfer function.

Object space techniques, on the other hand, accomplish the rendering by projecting each element onto the viewing plane such as to approximate the visual stimulus of viewing the element with regard to the chosen optical model. Two principal methods have been shown to be very effective in performing this task: *slicing* and *cell projection*.

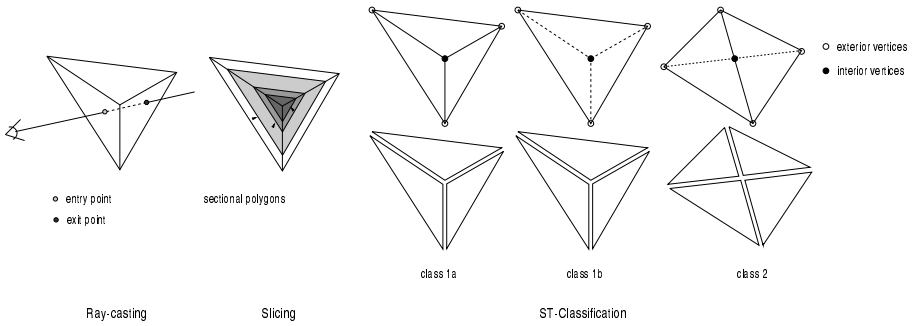

Fig. 1. *Different rendering techniques for tetrahedral elements are illustrated.*

Using the former method, each element gets projected on a slice-by-slice basis. Therefore, the cross sections of each element and the current slicing plane are computed. At each intersection point with one of the element edges the color value is interpolated from the pre-shaded samples at cell vertices. The cross sections are then rendered by letting the graphics hardware resample the color values onto the discrete pixel buffer during rasterization. The computation of the sectional polygons can either be done explicitly [19, 15], or implicitly on a per-pixel basis by taking advantage of dedicated graphics hardware providing per-fragment tests and operations [14].

A different object space approach is cell-projection [8]. Based on the current viewing parameters the projection of each tetrahedron is classified with respect to four different classes and decomposed into triangles correspondingly. For each vertex of the projected profiles color and opacity values are computed taking into account the underlying optical model. Although the bi-linear interpolation of these values across triangles doesn't give accurate results and is restricted to the rendering of pre-shaded samples, it has been established as the most prominent rendering technique for tetrahedral cells due to its efficiency. Different extensions to the cell-projection algorithm have been proposed in order to achieve better accuracy [12, 17] and to enable post-shading using arbitrary transfer functions [7]. The common method here is to pre-compute the volume rendering integral with regard to the current transfer function for a number of different parameter values and to code the results in a texture map. Then, rather than color samples the pre-computed values are interpolated and composited.

The more difficult problem, however, is to determine the correct visibility order of elements. Many algorithms have been proposed during the last couple of years trying to reach interactive frame rates in completely different ways. Probably the most efficient one has been used in [1, 4, 18]. These techniques exploit the fact that for tetrahedral meshes exhibiting a Delauney property the correct order can be found by sorting the tangential distances to circumscribing spheres using any customized algorithm. Although quite impressive frame rates can be obtained using this approach, the most serious drawback is that grids generated in practical applications are usually not Delauney meshes. This might lead to incorrect results and does not allow resolving topological cycles in the data.

A different alternative is the sweep-plane approach [3, 10, 9, 13]. In this approach the coherence within cutting planes in object space is exploited in order to determine the visibility ordering of the available primitives.

In addition, much work has been spent on accelerating the visibility ordering of unstructured elements. Usually, the data is first pre-processed in order to recover topological information, which is stored and used to accelerate the sorting procedure during rendering. Many different variants exist which exploit adjacency information within the grid and construct hierarchical data structures allowing for the efficient traversal of elements in the right order [16, 11, 2]

However, two significant problems are inherent to approaches that rely on pre-computed topological information. First, a considerable increase in memory might be introduced by the data structures necessary to allow for efficient visibility sorting. Second, as soon as changes in the geometry or the topology occur the pre-processing task has to be repeated. Particularly in numerical simulations where time-dependent sequences are generated quite commonly these limitations prohibit interactive frame rates.

In this paper, we present a novel approach for the resampling of scalar data given on unstructured tetrahedral grids. The goal of this approach is twofold: to emphasize the impact of state-of-the-art graphics hardware on current visualization techniques *and* to demonstrate how it can be used for the accurate rendering of large unstructured grids without the need for sorting the elements explicitly. As a direct implication the grid doesn't have to be pre-processed in order to determine topological information thus considerably reducing the memory overhead. Moreover, geometric and topological changes do not implicate expensive re-calculations. Particularly in practical applications where memory issues and topological changes of the grids are of major concern the real strength of our method comes into play.

The reminder of this paper is organized as follows. First, we describe the slicing procedure for tetrahedral cells our technique is based upon and we propose an algorithm in which hardware supported graphics operations are paramount. We then discuss implementation details and we outline two different alternatives for the rendering of unstructured grids taking advantage of our technique. Next, beneficial extensions using consumer graphics cards are demonstrated. We conclude the paper with a detailed discussion, and we show results and timings of our approach applied to real data sets.

2 Slicing tetrahedron

In the previous section we have outlined the general approach for slicing tetrahedra. We mentioned that only if pre-shaded samples are issued per vertex the generated fragments can be directly displayed and blended with pixel values already in the frame buffer. Unfortunately this strategy prohibits the use of arbitrary transfer functions to be applied to the original scalar data. Post-shading, on the other hand, allows the color distribution in the interior of each tetrahedron to be modified non-linearly. This can be achieved by interpreting the scalar material values as one-dimensional coordinates into a linear texture map. During rasterization texture coordinates are bi-linearly interpolated and the color value is finally looked up in a user defined texture lookup table.

The slicing of tetrahedral elements can also be viewed from a different perspective. As stated earlier, along each ray passing through a tetrahedron the material values depend linearly on the values reconstructed at the entry and the exit point. The reconstruction of the scalar field on a slicing plane parallel to the viewing plane thus involves computing the values at both the front and the back faces of the tetrahedron with respect to the current view, and to linearly combine them.

2.1 Hardware support

In order to avoid explicitly computing the sectional polygons and the scalar values used for shading we propose a method that takes advantage of hardware assisted polygon rendering and 2D texture mapping. Therefore let us assume that each tetrahedron to be projected has already been decomposed into triangles based on the ST-classification, named the ST-triangulation hereafter. We note that at the interior vertex of the triangulation two scalar values are stored: one for the point at the front facing edge and one for the point at the back facing edge. We can thus interpolate the data across the front faces and the back faces by scan-converting the ST-triangulation twice: In the second pass the function value at the interior vertex is exchanged (see Figure 2).

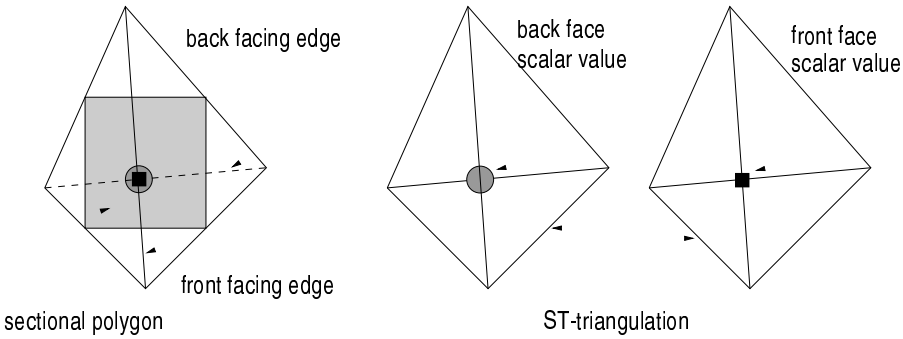

Fig. 2. *The scalar function can be reconstructed across the front and back faces by rendering the ST-triangulation twice with different scalar values at the interior vertex*

Although we already employ graphics hardware to reconstruct the scalar field at each pixel that is covered by the projection of the tetrahedron two problems still need to be addressed. The data should only be drawn to those pixels that are covered by the sectional region between the slicing plane and the tetrahedron *and* values need to be linearly combined in order to obtain the interpolated data samples. Both problems can be solved efficiently by means of a 2D texture map that stores pre-computed weights necessary to perform the linear interpolation. During rasterization the scalar values issued as polygon color are modulated with the texture color that represents the appropriate interpolation weights.

Let Nx, Ny be the size of the texture. Each texture element is described by a luminance and an alpha value. Then, by using $p = 2i/Nx - 1$ and $q = 2j/Ny - 1$ the

luminance values in the texture $T1$ are initialized as follows:

$$T1[i,j] = \begin{cases} \frac{p}{p+q} & : \quad i \geq Nx/2 \quad \& \quad j \geq Ny/2 \\ 0 & : \quad otherwise \end{cases}$$

Alpha values are set to one where the luminance values are different from zero, otherwise they are set to zero as well. In addition a second 2D texture map, $T2$, is created. It is similar to $T1$ but non-zero values $T1[i,j]$ are replaced by $1 - T1[i,j]$.

Before we analyze the assignment of texture coordinates to vertices of the ST-triangulation in more detail we should note that texture coordinates are going to be transformed to the range of (0,1) before rendering by means of the OpenGL texture matrix. As a consequence texture entry $(0.5, 0.5)$ is referenced by issuing a texture coordinate $(0,0)$. Thus only for positive coordinates non-zero texture values are mapped.

For every exterior vertex the u coordinate corresponds to the signed distance of that vertex to the current slicing plane. Values are positive if a vertex is located in front of the slicing plane with respect to the point of view. The v coordinate is equal to the u coordinate but its sign is flipped. At the interior vertex, however, the u coordinate equals the signed distance of the front face vertex to the slicing plane, while in the v coordinate the negated signed distance of the back face vertex to the slicing plane is kept. The principal assignment is illustrated in Figure 3.

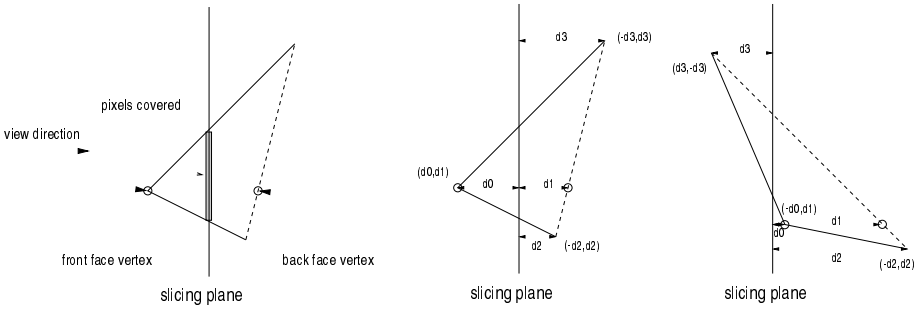

Fig. 3. *Sectional views of different ST-triangulations show the principal assignment of texture coordinates to vertices. Note that texture coordinates are properly transformed in order to address the appropriate texture values.*

Figure 3 tells us that u and v sum up to the thickness of the element along the viewing direction. Since u is just the thickness minus the distance from the slicing plane to the back faces, factors $1 - u/(u+v)$ and $u/(u+v)$ exactly correspond to the weights needed to modulate the scalar values at the front and at the back faces, respectively. These are exactly the factors that are pre-computed in the texture maps. As long as one of the texture coordinates is less than zero, both entry point and exit point lie either behind or in front of the slicing plane. Since negative texture coordinates index alpha values not equal to one, the generated fragments can be discarded before they affect the color buffer by means of the OpenGL alpha test. As a result only those fragments are going to be displayed and blended that are covered by the slicing plane.

In order to obtain for each pixel the accurately interpolated scalar values the ST-triangulation is rendered twice. In the first pass the scalar value at the front facing edge is issued at the interior vertex and fragment color is modulated with the texture $T2$. Blending is disabled in this pass. In the second pass the triangles are rendered using the scalar value at the back facing edge at the interior vertex and texture $T1$ is activated. The blending function is chosen such as to add incoming fragments to pixels, which finally results in the accurately interpolated scalar values.

We should note here the similarity of our method to the one proposed in [7], where a 2D texture was employed to code the interpolation weights needed to render shaded iso-surfaces on a per-pixel basis. In [14] a different alternative using the stencil and the depth test was demonstrated. Using this approach, however, it is necessary to read the color buffer for every single slicing in order to compute the correctly interpolated values. In the current implementation these values are directly obtained by hardware accelerated texture mapping and blending.

3 Tetrahedral grids

Now that we know how to exploit hardware assisted graphics operations for the resampling of scalar values on an arbitrary slice through a single tetrahedron the procedure can easily be extended to tetrahedral grids. First of all, for each slice those elements have to be determined that may intersect that slice. For this purpose we take advantage of an active element data structure [3, 19] that considerably limits the number of visited and processed elements per slice.

3.1 Implementation details

In order to perform the slicing of tetrahedral grids as efficient as possible some beneficial extensions have been integrated into our approach. Since in the proposed scenario elements might get drawn several times depending on the number of slices they overlap, one important requirement is to optimize the rendering of each of the constructed triangle sets. Therefore, in the current implementation we directly create triangle fans for each ST-triangulation [18] thus greatly improving the overall performance.

In addition we exploit coherences between slicing planes thus minimizing the number of numerical calculations to be performed. Although for a certain element the distance of each vertex to the current slice has to be computed, the geometry of the triangulation as well as the scalar values remain unchanged. As a matter of fact the ST-triangulation for every tetrahedron only has to be determined once. As soon as an element is inserted into the active element list the triangle fan and the distance of each vertex to the current slice are computed and stored. For every new slice the distances can now be updated incrementally.

In order to determine the ST-classification and to calculate the resulting triangulations we strictly avoid storing additional information. In his way we minimize the memory overhead, but even more importantly we guarantee that the delay is as short as possible once the grid is temporally modified.

3.2 Resampling vs. rendering

In order to visualize the resampled data we can read the pixel data into main memory and render it by means of any known volume rendering technique. However, in order to avoid the expensive memory transfer we use OpenGL to directly convert the pixel data into a 2D texture map or into a particular slice of a 3D texture map. This allows us to successively construct a stack of 2D textures or one single 3D texture that can be used later on for rendering purposes. Although in the current approach we exclusively exploit 3D texture maps, impressive image quality and performance can be achieved by means of 2D textures [6]. In this case, however, the data has to be resampled three times in order to generate the 2D texture stacks necessary to account for arbitrary changes of the viewing direction.

Additionally our method can be employed to directly render the volume data. Each slice is first rendered into a temporary buffer. Then the results are copied into the color buffer where they are blended appropriately. Pre-shading or post-shading works equally well using this approach. Pre-shaded colors are directly issued as polygon color at the triangle vertices. Post-shading, on the other hand, is achieved by selecting an appropriate lookup table that affects pixel values once they are copied into the color buffer.

Note that perspective projections can not be realized since texture coordinates are computed with respect to an orthographic projection. Even if two different ST-triangulations are computed representing the front faces and the back faces, respectively, and perspectively corrected texture mapping is performed, wrong interpolation weights will be computed due to the perspective distortion.

4 Texture combiners

Our approach can be improved considerably by exploiting the functionality of current PC graphics hardware, i.e. the Nvidia GeForce family GPUs. On these chips programmable fragment arithmetic is available that allows one to compute combinations between the color of incoming fragments and texture samples during rasterization [5]. The hardware is capable of simultaneously performing component-wise products and dot-products between the fragment color, texture samples or user-defined constant RGB values. One of the key features of this chip is that two texture units are available that allow for the simultaneous mapping of different textures to one single triangle. At first glance this functionality doesn't seem to be different to what we can achieve with multitextures as provided in the OpenGL 1.2 function set. In addition, however, texture combiners offer the possibility to multiply texture samples with different polygon colors before they are combined.

In order to take advantage of this functionality textures $T1$ and $T2$ are becoming the multitextures to be combined during rasterization. At each vertex of the ST-triangulation a primary and a secondary color is issued which only differ at the interior vertex. At this vertex the scalar value at the front facing edge and at the back facing edge is coded. In the first register combiner texture samples are simply modulated with the primary and the secondary color, respectively, and the results are added in order to obtain the interpolated scalar values. The correctly interpolated scalar values are thus generated and rendered into the color buffer in one single rendering pass.

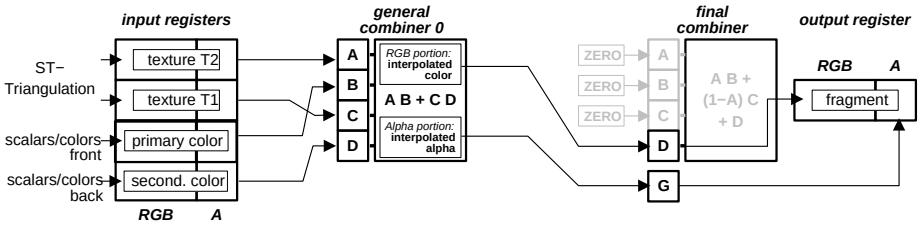

Fig. 4. *The register combiner setup for one pass data resampling.*

Obviously the same procedure can be employed using pre-shaded color samples as primary and secondary color. Since the interpolation is performed in the texture combiner results can be directly blended with pixels already in the color buffer. Neither does the pixel data has to be copied nor do we have to use any additional buffer.

5 Results

In this section we analyze the main modules and features of our system. All tests were run on a SGI Onyx IR2 equipped with one R12000, 300 MHz processor, 64 MB texture memory and 256 MB main memory.

Two tetrahedral data sets were used: A finite-element data set of 180K elements and the bluntfin data set converted to 225K elements. In our first test both data sets were resampled onto differently sized Cartesian grids.

We distinguish between the time needed to initialize and setup the active element list (**Ael**), the elapsed time consumed by the graphics subsystem (**Rnd**), the time it took to read the pixel data into main memory (**ToR**) and the time used to read and build the 3D texture (**ToT**). Since in all our examples we exclusively use 3D texture based rendering the total time is the sum of **Ael**, **Rnd** and **ToT**. Note that by using our approach for the direct rendering of unstructured grids the overall times will be slightly faster since the color buffer only has to be copied and the pixel data doesn't have to be converted into a texture map.

In all experiments the z-coordinate of the Cartesian grids correspond to the number of slices that were processed. No connectivity information was used and space was partitioned into 40 slabs in order to minimize the number of elements to be visited per slice. Table 1 shows explicit timings using hardware assisted color interpolation and 2D texture mapping. All images on the color page below show the resampled data sets rendered via 3D textures. In all cases the frame rate was faster than 8 fps. As can be seen, the time needed to render the tetrahedral elements dominates the overall performance. Since each element has to be rendered twice for every slice it overlaps the overall polygon count increases considerably. In total, the number of triangles rendered for both data sets was of about a factor of 24-39 times higher than the number of elements. On the other hand, as our timings show the additional overhead that is introduced can be absorbed very effectively by the graphics hardware. Since we swap expensive calculations into the graphics subsystem we reach impressive frame rates that are competitive

Resolution	Ael	Rnd	ToR	ToT
64x64x64	0.2	1.4	0.2	0.03
128x128x128	0.24	2.3	0.4	0.1
256x256x256	0.3	3.1	1.1	0.4

Resolution	Ael	Rnd	ToR	ToT
256x128x64	0.15	2.2	0.3	0.1
512x256x128	0.23	3.3	1.0	0.33

Fig. 5. *Timings (seconds) for the heat-sink (left) and the bluntfin (right) data set and differently sized Cartesian grids.*

to those presented in the literature. Although our timings are slower than those proposed in [18], our method keeps the memory overhead low and allows for topologically correct rendering of unstructured grids.

Compared to the most recently published approach [2] we are faster and we need less memory and entirely avoid expensive pre-processing of the data. Particularly for time-dependent data sets we expect our method to be superior to others because constant frame rates are guaranteed even if large parts of the grid are modified.

We have also implemented the proposed method on the nVidia GTS 2 graphics GPU. Using pre-shaded samples it took 2.3 seconds to directly render the heat-sink data set on a 512x512 pixel raster using 256 slices. We didn't realize the resampling into 2D textures due to the immense amount of texture memory necessary to store three copies of the volume.

6 Conclusion

In this paper we have emphasized a novel approach to achieve interactive display of unstructured grids. The major contribution here is that we efficiently exploit standard APIs like OpenGL to perform hardware assisted resampling and direct volume rendering.

In particular we have shown that our resampling approach in combination with texture based volume rendering allows for interactive exploration of large unstructured grids. For direct volume rendering arbitrary transfer functions can be applied to the scalar field.

Our results have shown that the presented method is as fast as any other method that allows for the topologically correct rendering of unstructured grids. Particularly for time-dependent data sets we expect our method to be superior because it does not rely on any pre-computed topological information. Any update of the grid can simply be realized by inserting new elements into the edge list and by removing non-valid elements from this list.

Furthermore, the user can specify arbitrarily sized regions to be resampled in any desired resolution. This allows one to select the resolution that can just be resampled at interactive rates. Once the data has been converted into a 3D Cartesian grid interactive rendering can be achieved using any known algorithm.

We have also shown how to efficiently exploit consumer graphics cards for the rendering of unstructured grids. Taking advantage of per-fragment arithmetic and multitextures direct rendering of tetrahedral grids using pre-shaded color samples can be achieved. Our timings have shown that we come close to the performance that can be reached using *CellFast* [18]. Although we considerably raise the load in the geometry

unit, at the same time we reduce the load in the CPU and minimize memory access. Overall, this leads to a very efficient alternative for the topologically correct rendering of large tetrahedral grids.

References

1. P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In *EG Workshop, Scientific Visualization in Scientific Computing*, pages 59–71, 1995.
2. J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. In *Computer Graphics Forum (Proc. EUROGRAPHICS '99)*, pages 369–376, 1999.
3. C. Giertsen. Volume Visualization of Sparse Irregular Meshes. *Computer Graphics and Applications*, 12(2):40–48, 1992.
4. M. Karasick, D. Lieber, L. Nackman, and V. Rajan. Visualization of three-dimensional delaunay meshes. *Algorithmica*, 19(1-2):114–128, 1997.
5. D. Kirk. From Multitexture to Register Combiners to Per-Pixel Shading. http://www.nvidia.com/Developer.
6. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and Ertl. T. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures And Multi-Stage Rasterization. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 109–119, 2000.
7. S. Roettger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings IEEE Visualization 2000*, pages 109–116, 2000.
8. P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '90*, 24(5):63–70, 1990.
9. C. Silva and J. Mitchell. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *Transactions on Visualization and Computer Graphics*, 4(2), June 1997.
10. C. Silva, J. Mitchell, and A. Kaufman. Fast Rendering of Irregular Grids. In *ACM Symposium on Volume Visualization '96*, pages 15–23, 1996.
11. C. Silva, J. Mitchell, and P. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proceedings ACM/IEEE Symposium on Volume Visualization 98*, pages 87–94, 1998.
12. C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *ACM Symposium on Volume Visualization '94*, pages 83–90, 1994.
13. R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering unstructured Volume Primitives by Polygon Drawing. In *IEEE Visualization '97*, pages 35–43, 1997.
14. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *ACM Computer Graphics (Proc. SIGGRAPH '98)*, pages 291–294, 1998.
15. J. Wilhelms, A. van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids. In *IEEE Visualization 1996*, pages 57–65, 1996.
16. P. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):102–126, 1992.
17. P. Williams, N. Max, and C. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, 1998.
18. C. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In *IEEE Visualization '99 Late Breaking Hot Topics*, pages 21–24, 1999.
19. R. Yagel, D. Reed, A. Law, P. Shih, and N. Shareef. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. In *ACM Symposium on Volume Visualization '96*, pages 55–63, 1996.

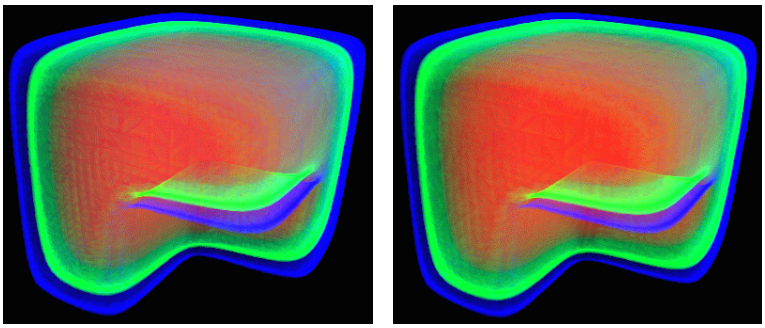

Fig. 6. *The heat-sink data set was resampled on a* 128^3 *(left) and on a* 256^3 *(right) Cartesian grid and rendered via 3D textures.*

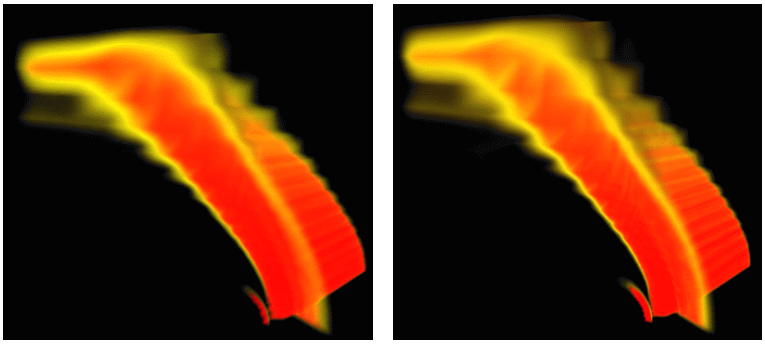

Fig. 7. *Both images show the bluntfin data set resampled on a 256x128x64 (left) and a 512x256x128 (right) Cartesian grid and rendered via 3D textures.*

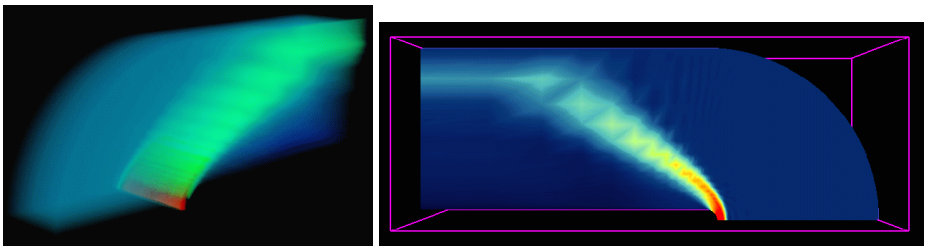

Fig. 8. *On the left the bluntfin data set was resampled on a 512x256x128 grid. On the right our method was employed to resample the data on an arbitrary slice. The resampling time was 0.18 seconds.*