Real-time Visualization of Unstructured Volumetric CFD Data Sets on GPUs

Mario Höfler*

Institute for Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

Real-time visualization and analysis of scalar and vector fields from massive volumetric data sets require efficient rendering techniques. Irregularly structured Computational Fluid Dynamics (CFD) cells have to be tetrahedralized prior to rendering on the Programmable Graphics Unit (GPU). The cells are organized within an octree providing fast search in the spatial, but also in the scalar function domain. Identification of the cells being part of the resulting isosurface is possible using the hierarchical data structure. Taking into account only relevant cells for rendering reduces time consumption and overhead caused by data transfers. The gained speedup depends on the distribution of values and cells in 3D space. Cross-section of the desired isosurface with a cell leads to the nodes being used for interpolation per polygon. Interpolation, pseudocoloring and shading is done on the GPU using vertex and fragment shaders. Shifting tasks from CPU to the GPU increases the framerate and enables customized shading. The spatial impression of the resulting isosurfaces is increased using Phong shading.

Keywords: Isosurface visualization, In section representation, Tetrahedralization, Octree, Shading languages

1 Introduction

Volume rendering deals with visualization of sampled scalar functions of three spatial dimensions [5, 14] and provides possibilities for visualization of normally unseen areas at high computational costs. Volume data often appears in medical applications or scientific simulations and may be produced by electromagnetic radiation sensor systems like Computer Tomography or comes from simulation results using mathematical CFD models. Indirect volume rendering methods like isosurface visualization or cross-section of the data with arbitrary functions allows real-time interaction even with huge data sets. Real-time interaction capabilities also enrich the impression for changes in the scalar domain by the observer. In section representations based on resampling the volume onto regular grid would cause discontinuities at the grid transitions which should be avoided. Fast and easy inspection of the scalar structure of data sets in VR/AR setups is supported by defining the cutting plane for the in section representation using a translucent board called personal interaction panel (PIP).

In this paper we present a method for rendering isosurfaces and cross-sections consisting of off-line preprocessing stages and on-line real-time rendering tasks. We convert the CFD input cells on irregular meshes into a mesh of tetrahedra in a preprocessing phase. A hierarchical data structure is used to organize the cells spatially on CPU side. To free cycles on the CPU we hand over interpolation and lighting tasks to programmable graphics hardware. GPU shaders allow performing arithmetic floating point operations on the graphics unit additional to traditional tasks to be done during rendering within the graphics pipeline.

In Section 2, we give an overview of the related work done in this area. In Section 3, we present necessary preprocessing of the given CFD data. In Section 4, we describe the tasks for identification of relevant data. In Section 5, an explanation of the differences between marching tetrahedra and marching cubes is given followed by Section 6, describing our implementation using shading languages. Results are shown and discussed in Section 7 and finally, conclusions and future work for improvements can be found in Section 8.

2 Related Work

Despite the fact, that several fast direct volume rendering methods like ray-tracing in the image domain, or projection methods working in object space, exist, the resulting images, showing details of the volume using particular transfer functions, are bought for high memory and time costs. Scharsach [10] describes an advanced ray-casting technique.

The second type of direct methods, cell projection, is based on projection of basic primitives like tetrahedra on unstructured grid onto view plane in back to front order. Shirley and Tuchman [11] called this approach Projected Tetrahedra (PT) algorithm therefore. Displaying multiple

*mario.hoefler@gmx.net

isosurfaces and combination of isosurfaces with projected volume cells is described by Röttger et al [9]. Textures are used to store exact color and opacity values instead of inaccurate value interpolation between vertices. The time complexity for visualization using 2D or 3D texture mapping is at most linear with the number of tetrahedral cells. A view-independent, hardware-based cell projection system was shown by Weiler et al [15]. The big advantage of this improved version extending the PT method is the possibility for doing all projections and scan conversions of tetrahedral cells directly on the GPU because of the homogeneous processing per cell independent of the camera position.

The well known Marching Cubes [7] algorithm used for level surface determination of volume data with scalar values on regular grid already delivers isosurfaces. Linear interpolation between grid points with values surrounding the desired value, and classification of the intersection situation into one out of 2^8 possible cases for a cube, leads to the isosurface. The problem with this algorithm and its optimization is the required regular 3D structure of the data. Nevertheless the basic idea of identifying data points used for linear interpolation through analysis of the cross-section of a cell with a plane approximating the isosurface is used also in our approach.

Livnat et al [6] presents span spaces and their organization in the scalar domain, providing an efficient method to search for cells intersecting the isosurface. A kd-tree is used to decompose the span space according to the minimum and maximum value per level. The Marching Cubes algorithm is used for visualization of the isosurface. We want to extract cells based on their location for the in section representation as well as extraction based on scalar values. Our approach based on an octree with the additional scalar value range hierarchy supports this requested dynamic behavior.

An alternative method for isosurface visualization based on point based rendering is presented by Co et al [2].

Several methods for visualization of CFD data are introduced by Ebert [3]. The importance of data structures in real time graphics systems, especially the advantages of octrees, are investigated by Wilhelms and Gelder [16].

Using a tetrahedral mesh as input to a volume rendering system simplifies the interpolation and per primitive drawings as a result as shown by Carneiro et al [1].

Handing over specific computations from CPU to the GPU in the context of interactive volume rendering is investigated by Wylie et al [17], Pascucci [8] and Klein et al [4]. The necessary processing of all cells of the volume per frame is the common main disadvantage of the previously mentioned solutions.

3 Preprocessing

The interaction with the data set in real-time requires an off-line preprocessing phase. One goal of the first stage is gathering appropriate information per vertex described in Section 3.1. A second stage, described in Section 3.2, is necessary for producing a hierarchical structure used in the following on-line phase.

3.1 Tetrahedralization

CFD simulation results may consist of scalar- and vector fields. The spatially unstructured volume mesh of CFD cells is difficult to analyze directly because there are multivariate per cell data and the number of nodes per cell may differ from cell to cell. Therefore the input data need to be preprocessed in advance to applying our algorithm to be able to provide homogeneous cell processing mechanisms. The tetgen library [1] is used for the tetrahedralization of the input data. As this process is constrained by the boundary of the CFD mesh (a piecewise linear complex), Steiner points are inserted as needed, and their associated scalar values are interpolated from the neighbors [12].

Interpolation of gradients, becoming normal vectors, and node values from neighboring cell values and resampling of the volume of nodes receiving a tetrahedral mesh of convex cells are the main steps of this first off-line stage.

The tetrahedron as a basic building block for the resampled CFD data has the property that the cross-section with a plane can only be a triangle or quadrangle. This fact allows us using the OpenGL `GL_QUAD` drawing mode for every polygon being part of the resulting surface. The big advantage of the `GL_QUAD` drawing mode is the fact, that using one vertex twice within one quad automatically forces OpenGL to treat the geometry as a triangle without any error. Furthermore we are able to build a homogeneous rendering system independent of the shape of the surface always expecting four interpolated corners per tetrahedra cross-section.

We approximate the gradient ∇f at a point $\mathbf{p}$ as

$$\nabla f = \sum_{i=1}^{N} (f(\mathbf{p}_i) - f(\mathbf{p})) \frac{\mathbf{p}_i - \mathbf{p}}{||\mathbf{p}_i - \mathbf{p}||^2},$$

where $\mathbf{p}_i$ are the centers of the N cells sharing a common vertex at $\mathbf{p}$, $f(\mathbf{p}_i)$ are the scalar values computed for these cells, and $f(\mathbf{p})$ is simply chosen as the average $f(\mathbf{p}) := \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{p}_i)$. However, this approximation is not suitable for highly curved regions. We are currently investigating other methods to compute the gradient.

3.2 Spatial Partitioning

Efficient identification of tetrahedra being used for visualization requires spatial subdivision of the volume for in section representation and a span space of the scalar- or vector field for the level- or isosurface extraction. We use an octree for the hierarchical structuring. The center of gravity per tetrahedron is used for assigning it to an octant. Each octant is either empty or consists of eight

[1] http://tetgen.berlios.de

sub-octants, or contains one tetrahedra. Attempts to insert another tetrahedron into an already occupied octant forces recursive splits into eight sub-octants. The octants do not overlap whereas the tetrahedra may overlap octant bounds. The covered space of each subtree is defined using a conservative bounding sphere of the corresponding parent node of the subtree. Bounding spheres are used because their volumetric capacity is more adequate for tetrahedra compared to axis aligned bounding boxes. All nodes of subtree tetrahedra define a common center of gravity which becomes the center of the bounding sphere. The radius of the used bounding sphere is equal to the distance of the farthest tetrahedron vertex with respect to the common center of gravity. Additionally to the spatially covered region, each tree item also stores the covered value range per scalar- and vector field of the tetrahedra lying in the covered spatial region of the subtree. In our implementation we export the octree nodes into a index-array as a last step of the preprocessing phase. Working only with the simple array increases the performance of the on-line cell search and decreases dynamic memory consumption.

4 On-line Tetrahedra Extraction

The static octree representation has to be searched for cells being used for the visualization starting afterwards. Searching for k cells being part of the resulting surface of totally n tetrahedra in the octree takes $O(k+k\cdot\log(\frac{n}{k}))$ time [6]. Isosurface visualization requires a cell search based on value ranges whereas in section representation requires spatial intersection tests. Anyway the result of each individual search described in sections 4.1 and 4.2 is an array of extracted tetrahedra indices. Constant isovalues or functions for the in section representation require only one extraction at the beginning of the visualization independent of the viewing parameters. Each update of the desired isovalue or plane forces a new tetrahedra extraction.

4.1 Level Surface Visualization

Identification of tetrahedra with contributions to the isosurface is the first step after receiving a new isovalue. Equation 1 shows the necessary condition for points on the isosurface (see Figure 1) with $f(\mathbf{x})$ being a spatial function such as temperature, velocity or pressure.

$$f(\mathbf{x}) - isovalue = 0 \tag{1}$$

The required informations to interpolate an isosurface are vertex positions, their associated values and the isovalue.

4.2 In Section Representation

The difference to the previously described mode is the independence of node values. In section representation (see

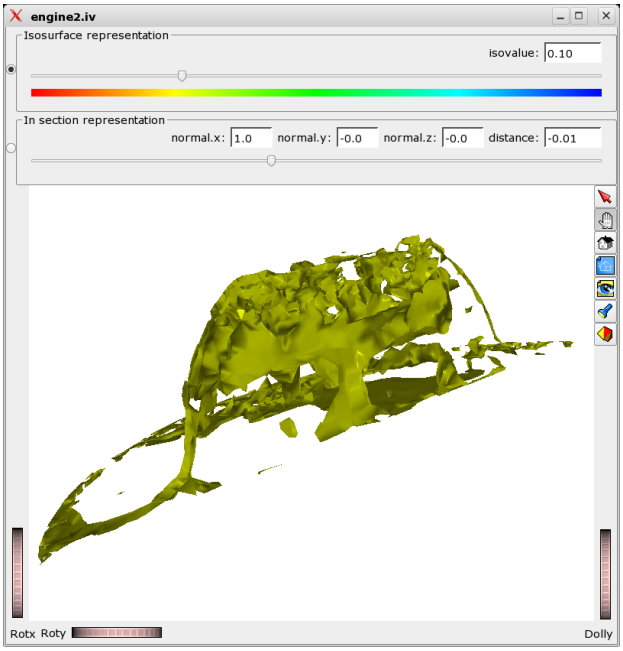


Figure 1: The isosurface visualization of the temperature during combustion within an engine (see data set 2 in Table 2). The slider at the top of the image is used for changing the isovalue interactively. A big advantage of this kind of visualization is the fast and concise overview of the structure of scalar fields in 3D space.

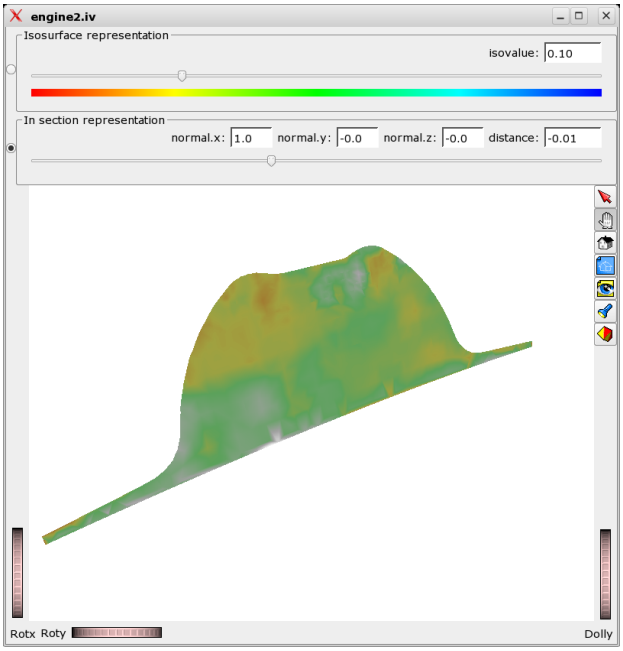


Figure 2: In section representation using a cutting plane requires the user to enter the orientation of the cutting plane together with the perpendicular distance from origin. The color bar gives a hint for color/temperature encoding. This kind of representation can be used if the cross-section of volumes with cutting planes may give valuable information about flows or the structure of scalar fields.

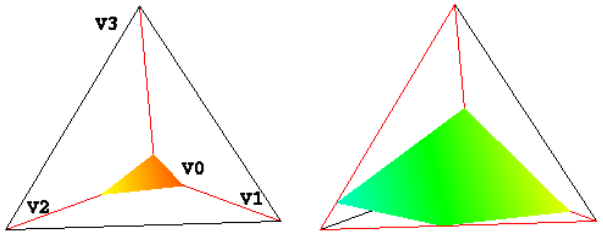
Figure 2) of the volume data set for a given plane is composed by connecting the plane-edge intersections of edges between tetrahedra nodes lying in different half spaces with respect to the given cutting plane.

$$< \mathbf{x}, \mathbf{n} > -d = 0 \tag{2}$$

Equation 2 shows the necessary condition for points on the in section surface with vertex $\mathbf{x}$, plane normal $\mathbf{n}$ and perpendicular distance d from plane to the origin in the world coordinate system. Again only tetrahedra surrounding the cutting plane have to be taken into account. The required informations for this representation are vertex positions, their associated values and the cutting plane parameters.

5 Marching Tetrahedra

Marching tetrahedra is a method, similar to the well-known marching cubes algorithm, leading to $2^4 = 16$ possible in section regions in case of simple cutting planes including the two trivial cases, all nodes in front or back of the plane. The main advantage of working with tetrahedral cells instead of cubes are less cross-section cases leading to simpler algorithms with less ambiguity configurations. Another important criterion is the convexity of the tetrahedral mesh which cannot be guaranteed by the original CFD cells. The intersection of a tetrahedron with a plane can only result in a triangle or quadrangle (see Figure 3).



(a) Code e.g. '$1110_B = 14_D$' or '$0001_B = 1_D$'

(b) Code e.g. '$1010_B = 10_D$' or '$0101_B = 5_D$'

Figure 3: Tetrahedron in section representations. The vertex arrangement is the same over all images. The code or bit mask is true (positive logic) for a vertex with a higher value compared to the isovalue, or for a vertex lying within the cutting plane half-space. The bit corresponding to V0 is at the LSB position in the bit mask unlike V3 which affects the MSB. The edges to be used for interpolation are colored red. Subscript B refers to a binary notation, whereas D refers to the decimal notation.

Figure 3 also shows different triangle or quadrangle slices together with their hit codes. A hit code represents the intersection situation of the tetrahedron with a plane. The four bits of the code stands for tetrahedron nodes and their relative position given a plane. The equations 1 and 2 are used to determine the bit value during tetrahedra extraction. For the isosurface extraction, node values higher than the isovalue lead to a one for the corresponding node bit in the bit mask. In the in section representation case, the one is assigned to nodes within the half-space of the given cutting plane. Using the hit code all vertices used for interpolation can be identified uniquely. We use lookup tables returning edge indices for a given hit code.

6 Surface Extraction and Shading

We use the C for graphics (Cg) toolkit from NVIDIA as shading language because of its high-level programming language capabilities and its portability through platform profiles. Interpolation of polygon vertices per polygon representing a subsurface of the desired isosurface, or the in section surface, is done by different vertex shaders bound at runtime. The shader selection depends on the visualization mode selected by the user.

Instead of transferring single vertices with their normal vectors from CPU to GPU, a Vertex Buffer Object (VBO) is used. After extraction of relevant tetrahedra, end point informations per edge, along to interpolate using node values or vertex positions, are written into the VBO. Table 1 shows the layout for a single VBO edge entry. Transferring edges between CPU and GPU instead of vertices causes a little memory overhead because several edges may be connected to a single vertex dependent on the topology of the tetrahedral mesh. The advantage of this method is that the VBO is initialized once at start-up and rendering only requires providing edge indices. The OpenGL `glMultiDrawElements` mode is used to trigger rendering the resulting surface. The binding between OpenGL and Cg for the $2 \cdot 4 + 2 \cdot 3 = 14$ variables is done using standard client states for vertices, texture coordinates, normals and colors.

Listing 1 shows the vertex shader for the position interpolation whereas Listing 2 is used to interpolate scalar values given the intersection. The vertex shaders are executed per edge delivering one of maximal four end points of the quadrangle representing the tetrahedron cross-section. `v1` is the first vertex of the edge, `v2` the second, with `g1` and `g2` as the corresponding normal (gradient) vectors.

```
vert2frag main(
{
  float4 v1   : POSITION,
  float4 v2   : TEXCOORD0,
  float3 g1   : NORMAL,
  float3 g2   : COLOR,
  uniform float4x4 mod_view,
  uniform float4x4 mod_view_it,
  uniform float4x4 mod_view_proj,
  uniform float isovalue,
  uniform float max_value
)
{
  vert2frag OUT;
  float alpha =
```

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Edge	Start node							End node						
	Vertex			Normal			Scalar	Vertex			Normal			Scalar
	x	y	z	x	y	z		x	y	z	x	y	z	

Table 1: VBO edge entry layout

```
    (isovalue-v1.w)/(v2.w-v1.w);
  float3 position =
    lerp(v1.xyz, v2.xyz, alpha);
  float3 gradient =
    normalize(lerp(g1, g2, alpha));
  OUT.position_cam = mul(
    mod_view, float4(position, 1));
  OUT.normal_cam = mul(
    mod_view_it, float4(gradient, 0)).xyz;
  OUT.color = isovalue/max_value;
  OUT.position = mul(
    mod_view_proj, float4(position, 1));
  return OUT;
}
```

Listing 1: Vertex shader for the isosurface visualization

```
vert2frag main(
  float4 v1  : POSITION,
  float4 v2  : TEXCOORD0,
  uniform float4x4 mod_view,
  uniform float4x4 mod_view_it,
  uniform float4x4 mod_view_proj,
  uniform float3 plane_normal,
  uniform float plane_distance,
  uniform float max_value
  )
{
  vert2frag OUT;
  float distance = -plane_distance;
  float3 edge_direction = v2.xyz - v1.xyz;
  float denominator = dot(
    plane_normal, edge_direction);
  float alpha = (-distance - dot(
    plane_normal, v1.xyz))/denominator;
  float3 position =
    v1.xyz + alpha*edge_direction;
  float value = lerp(v1.w, v2.w, alpha);
  OUT.position_cam = mul(
    mod_view, float4(position, 1)).xyz;
  OUT.normal_cam = mul(mod_view_it,
    float4(plane_normal, 0)).xyz;
  OUT.color = value/max_value;
  OUT.position = mul(
    mod_view_proj, float4(position, 1));
  return OUT;
```

Listing 2: Vertex shader for the cutting plane visualization

For lighting reasons the gradient or normal vectors are of great interest. The fragment shader is used to map the scalar value to a color using a one-dimensional transfer function. Local Phong illumination and shading is implemented to visualize specular effects. In the Phong reflection model the pixel color is determined using the interpolated normal vector. One can get depth cues using this kind of shading and spatial impressions through specular highlights if $(\mathbf{R} \cdot \mathbf{V})$ is high (see Figures 7 and 9). During in section representation of the data, specular lighting is disabled because a 2D cutting plane does not require spatial impression and the color of the interpolated values per pixel are of interest independent of the camera position (see Figures 8 and 10). View dependent lighting does not make sense on a plane since high values for $(\mathbf{R} \cdot \mathbf{V})$ results in whitening the hole plane, which is of no help during analysis of volumetric data sets.

7 Results

The presented method for real-time visualization of unstructured CFD data sets is capable of calculating and rendering 30k plane/tetrahedron intersections with more than 50 frames per second and performing Phong shading on each pixel based on normal vector or gradient direction and the direction of light. Due to the hierarchical data structure the total number of tetrahedra within the tetrahedral mesh might be much higher since the rendered tetrahedra are extracted prior to rendering identified to be a part of the resulting surface. We use the data sets listed in Table 2 for testing on a Intel Pentium 4 3GHz CPU with 2MB cache, 1GB RAM and NVIDIA GeForce 7800 GPU.

Id	Tetrahedra	Nodes	Edges	Octree depth
1	625k	132.6k	772.6k	8
2	44.1k	8.7k	56.7k	10

Table 2: The data sets used for testing. Data set 1 is created synthetically (see Figure 9). Data set 2 represents the CFD simulation result of the temperature during combustion in a two-stroke engine.

Figure 4 shows the separate timings for cell extraction and isosurface visualization for data set 1. Searching for cells is significantly faster than pure rendering, therefore our combined approach is balancing the workload between the CPU and the GPU quite well. Figure 5 shows the separate timings for cell extraction and in section representation for the same data set. The linear relationship between the number of extracted cells and the framerate is not as easy to see as in the previous Figure 4 because the

number of cells, in section with the cutting plane, is the same for the most cases. Either a minimum or a maximum in case of axis parallel cuts with respect to the volume. Therefore the number of samples between this minimum and maximum number of hit cells is too small in order to be representative. Finally, Figure 6 shows the total timings for tetrahedra extraction and surface visualization for both cases, isosurface visualizations and in section representations. The framerate for the combination might be higher than in the cell extraction only case, because here cell search is only executed if changes of the isovalue or the cutting plane parameters occur. Note that the framerate is above 50 frames per second in most cases even if the number of extracted tetrahedra is above $32 \cdot 10^3$ given $625 \cdot 10^3$ total tetrahedra.

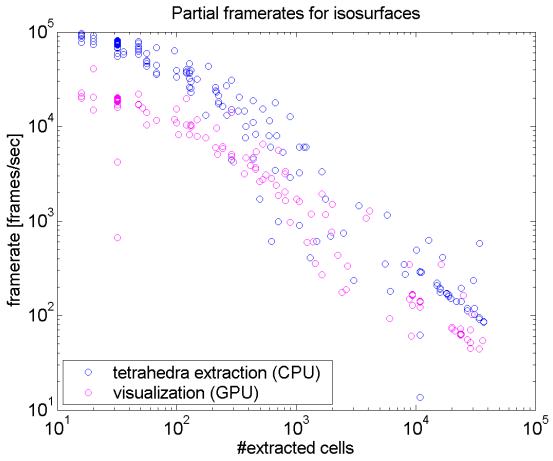


Figure 4: Timings for octree traversal (blue), done on the CPU, and interpolation and shading (magenta), done on the GPU, are shown for isosurface visualizations of data set 1.

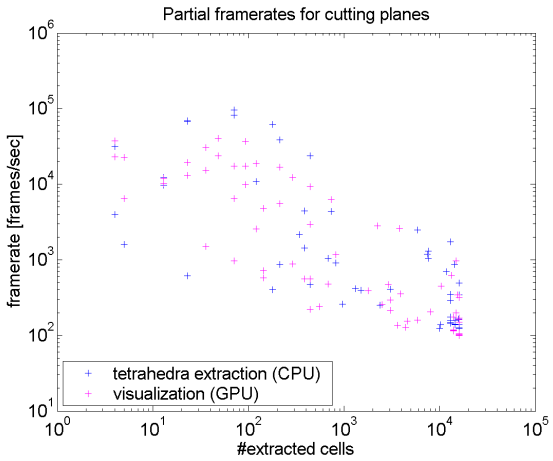


Figure 5: Timings for octree traversal (blue), done on the CPU, and interpolation and shading (magenta), done on the GPU, are shown for in section representations of data set 1.

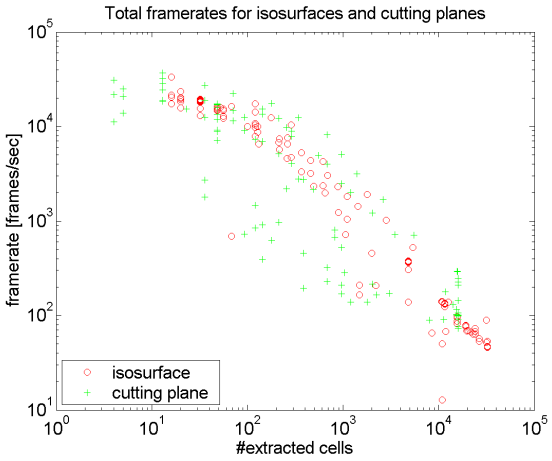


Figure 6: The total framerates for isosurface visualizations (red), and in section representations (green), of data set 1 are shown here.

A crucial performance impact results from the spatial and thematic distribution of the data. If the data is degenerated in terms of isosurfaces, built from nearly all cells within the data set, the octree traversal will run with low performance. Therefore the depth$= \Omega(\log_8 n)$ of the octree is an important parameter for checking the distribution of the data.

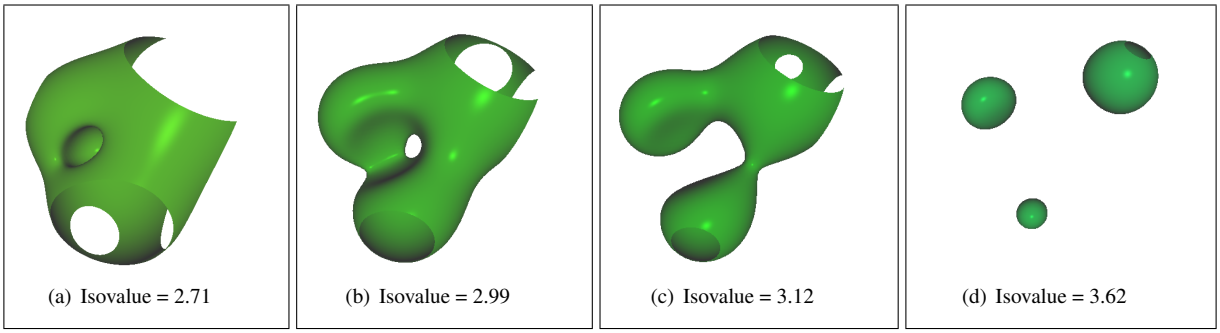
Floating-point intrinsics, provided by recent CPUs, are used to speedup parallel dot-product calculations following the SIMD principle during hit-code generation whilst octree traversal. SIMD Streaming Extensions (SSE3) provide 128 bit length registers allowing four floating-point operations to be processed in parallel. Using SSE3, the hit-code determination can be done using the CPU for little additional cost to cell extraction already done on the CPU using the octree.

8 Conclusions and Future Work

Using the tetrahedron as basic primitive for interpolation of cross-sections simplifies the information transfer from CPU to GPU and supports a stable and homogeneous solution.

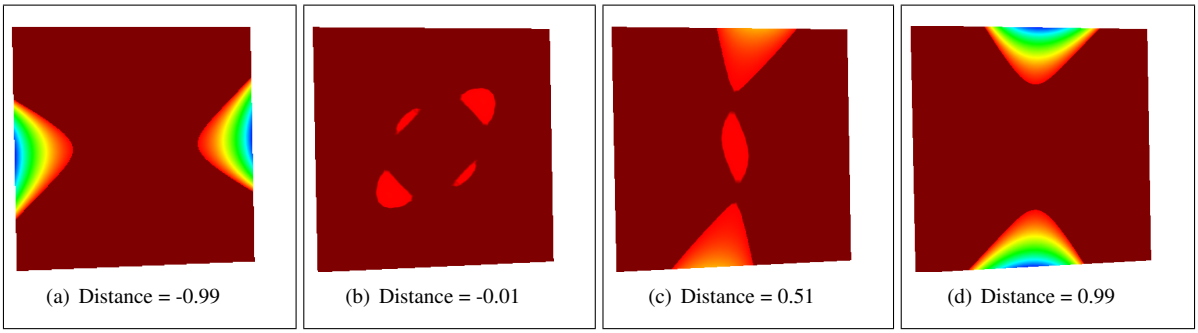
The performance of the presented visualization system depends on the spatial data distribution. Equally distributed vertices over the volume, with equally distributed values over the hole value range, are advantageous preconditions for isosurface visualization methods to be efficient. In our case the depth of the octree is minimal and the scalar value ranges are consistent allowing fast searches for data set distributions mentioned above.

Caused by using VBO technology for the data transfer from main memory to video memory, we have to transfer edge indices from the CPU to the GPU using OpenGL commands. The set of edges of the tetrahedral mesh is generated in an off-line stage.



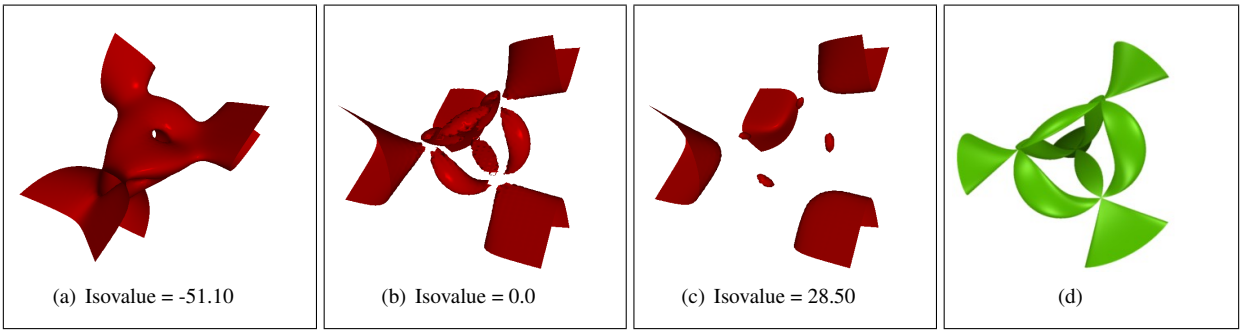
(a) Isovalue = 2.71 (b) Isovalue = 2.99 (c) Isovalue = 3.12 (d) Isovalue = 3.62

Figure 7: Isosurface examples for an artificial data set generated from addition of five terms of the form: $\frac{c_i}{\|x \cdot p_i\| + d_i}$.



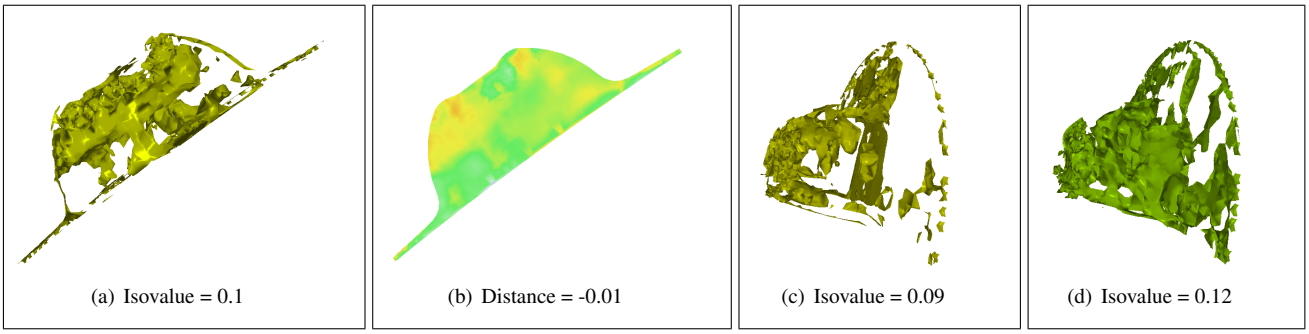
(a) Distance = -0.99 (b) Distance = -0.01 (c) Distance = 0.51 (d) Distance = 0.99

Figure 8: Cutting plane example for another artificial data set with constant cutting plane normal.



(a) Isovalue = -51.10 (b) Isovalue = 0.0 (c) Isovalue = 28.50 (d)

Figure 9: Isosurface variations for artificial data set 1 (see Table 2). Compare Sub-figures (b) and (d). Sub-figure (d) shows the SuSE Linux 8.2 Professional Edition Title Surface.



(a) Isovalue = 0.1 (b) Distance = -0.01 (c) Isovalue = 0.09 (d) Isovalue = 0.12

Figure 10: Isosurfaces and in section representation of the engine data set 2 (see Table 2).

View-frustum culling can be implemented through detection of visible areas given the actual camera position using the octree. This culling technique may lead to higher framerates especially during observation of details (zooming) via clipping invisible areas as early as possible during octree traversal. The octree can be further optimized for memory- and cache efficiency using contiguous address spaces.

Last but not least data reduction would be an efficient method for increasing the performance of our method through mesh simplification and vertex clustering including edge removals during preprocessing [13].

9 Acknowledgments

Many thanks to my adviser Markus Grabner [2] for his great support during carrying out this work. I also thank Oliver Labs for providing functions based on E. Stagnaro quintic functions to generate nice artificial data sets used for testing [3].

References

[1] Carneiro B.P., Silva C., and Kaufman A.E. Tetracubes: an algorithm to generate 3d isosurfaces based upon tetrahedra. volume 9, pages 205–210. Proceedings of SIGGRAPH 96, Aug 1996.

[2] Christopher S. Co, Bernd Hamann, and Kenneth I. Joy. Iso-splatting: A point-based alternative to isosurface visualization. In *Proceedings of the Eleventh Pacific Conference on Computer Graphics and Applications - Pacific Graphics 2003*, pages 325–334, October 2003.

[3] David S. Ebert, Roni Yagel, Jim Scott, and Yair Kurzion. Volume rendering methods for computational fluid dynamics visualization. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 232–239. IEEE Computer Society Press, 1994.

[4] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004.

[5] M. Levoy. Display of surfaces from volume data. In *Computer Graphics and Applications*, volume 8 of *3*, pages 29–37, May 1988. Volume Rendering.

[6] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Trans. Vis. Comput. Graph.*, 2(1):73–84, 1996.

[7] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[8] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. pages 293–300, 2004.

[9] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *IEEE Visualization*, pages 109–116, 2000.

[10] Henning Scharsach. Advanced GPU Raycasting. In *Proceedings of CESCG 2005*, pages 69–76, 2005.

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

[12] Hang Si and Klaus Gärtner. Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations. In *Proceedings of the 14th International Meshing Roundtable*, September 2005.

[13] Graham M. Treece, Richard W. Prager, and Andrew H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers & Graphics*, 23(4):583–598, 1999.

[14] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, New York, 1992.

[15] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, 2003.

[16] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

[17] B. Wylie, K. Moreland, L.A. Fisk, and Crossno P. Tetrahedral projection using vertex shaders. pages 7–12, Oct 2002.

[2] grabner@icg.tu-graz.ac.at

[3] `http://www.oliverlabs.net/suse`