HAL
open science

GPU Accelerated Isosurface Extraction on Tetrahedral Grids

Luc Buatois, Guillaume Caumon, Bruno Lévy

To cite this version:

Luc Buatois, Guillaume Caumon, Bruno Lévy. GPU Accelerated Isosurface Extraction on Tetrahedral Grids. 2nd International Symposium on Visual Computing - ISVC06, Nov 2006, Lake Tahoe/USA. inria-00105584

HAL Id: inria-00105584

https://inria.hal.science/inria-00105584v1

Submitted on 11 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GPU Accelerated Isosurface Extraction on Tetrahedral Grids

Luc Buatois[1,2], Guillaume Caumon[1,3], and Bruno Lévy[2]

[1] Nancy Université, Gocad Research Group, Nancy, France,
Luc.Buatois@gocad.org,
[2] ALICE, Inria Lorraine, Vandoeuvre, France
[3] CRPG-CNRS, Vandoeuvre, France

Abstract. Visualizing large unstructured grids is extremely useful to understand natural and simulated phenomena. However, informative volume visualization is difficult to achieve efficiently due to the huge amount of information to process. In this paper, we present a method to efficiently tessellate on a GPU large unstructured tetrahedral grids made of millions of cells. This method avoids data redundancy by using textures for storing most of the needed data; textures are accessed through vertex texture lookup in the vertex shading unit of modern graphics cards. Results show that our method is about 2 times faster than the same CPU-based extraction, and complementary with previous approaches based on GPU registers: it is less efficient for small grids, but handles millions-tetrahedra grids in graphics memory, which was impossible with previous works. Future hardware evolutions are expected to make our approach much more efficient.

1 Introduction

1.1 Motivations

Visualizing isosurfaces is essential in many different fields of scientific research like Computational Fluid Dynamics (CFD), finite element modeling and medical and seismic tomography. These applications often use large unstructured grids made of millions of tetrahedra. Handling these kind of very large grids without out-of-core or parallel algorithms using a simple pc leads our approach.

Each cell, in an unstructured tetrahedral mesh, has a constant topology. Hence, very specialized algorithms for this type of cells have been carried out using hardware acceleration techniques [1–4]. Unfortunately, these methods introduce sometimes redundancy in the storage method, strongly limiting the size of the grid or, sometimes, use special non-standard functionalities implemented on few graphics cards.

We propose a way to efficiently extract an isosurface from a scalar field by means of modern GPUs for unstructured tetrahedral meshes. Our method handles very large grids made of millions of tetrahedra by means of common GPUs.

1.2 Previous Work

Isosurface extraction can be optimized by going only through the intersected cells in the grid, and by accelerating the extraction of the isovalue polygon in each intersected cell.

When marching through all cells to extract an isosurface, most of the time is spent checking for non-intersected cells. Therefore, algorithmic acceleration techniques have been investigated to discard non-intersected cells before rendering them. Contour seeds [5–7] start from seed cells to propagate over the grid to build the requested isosurface. Interval Trees [8] work on value space to classify cells within intervals of values. Octrees [9] recursively subdivide space remembering at each stage the interval of values contained in each subdivision. Others important algorithmic acceleration techniques exist [10–12] and are very efficient.

This article is mainly focused on cell tessellation of large grids. Cell tessellation has been extensively studied in the literature for the past 20 years. The Marching Cubes [13] is one of the first efficient techniques to directly tessellate a hexahedron. Cell projection [14] doing indirect extraction or a technique that turns around faces [15–17] using topological links between the vertices/faces/edges of each cell, are others basis of art. Our method is inspired by the Marching Cubes [13], so we now review the main steps of this algorithm.

The Marching Cubes algorithm efficiently tessellates a hexahedron, using two precomputed lookup tables: the table of *edges* contains the numbers of the beginning and ending vertices for each cell edge, and the table of *cases* contains all possible configurations of the isosurface to be extracted. A *plus* (resp. *minus*) is attached to each vertex if the value at this vertex is higher (resp. lower) than the isovalue. Out of the $2^8 = 256$ possible configurations for a hexahedron, the Marching Cubes method takes into account symmetries to reduce the size of the table of *cases* to only 15 entries. Knowing both description tables, extracting an isosurface from a hexahedron is easy:

- From the current hexahedron configuration, compute the index in the table of *cases* as: $index = \sum_{i=0}^{7} (F(x_i) >= w) * (i+1)^2$, where $F(x_i)$ is the value attached to the vertex x_i and w the isovalue.
- Read the table of *cases* at this index to retrieve the list of intersected edges.
- Retrieve the end vertices of each intersected edge using the table of *edges*, and compute the intersection with the isosurface by linear interpolation.

The Marching Cubes can be adapted to tetrahedral grids (Marching Tetrahedra), using specific tables of *edges* and *cases* for a tetrahedron (Fig1).

Hardware Acceleration Techniques using GPUs. Programmable graphics hardware have opened new perspectives for isosurface extraction. Currently, due to hardware limitations, the only types of grids that have been hardware-accelerated for direct isosurface extraction are unstructured tetrahedral meshes

and regular hexahedral meshes. For regular hexahedral grids, isosurface extraction can be achieved using pre-integrated volume rendering [18, 19]. These methods volume-render the whole grid using a Dirac opacity transfer function so that only one isovalue is rendered. They have a high computational complexity as compared to polygonal isosurface extraction.

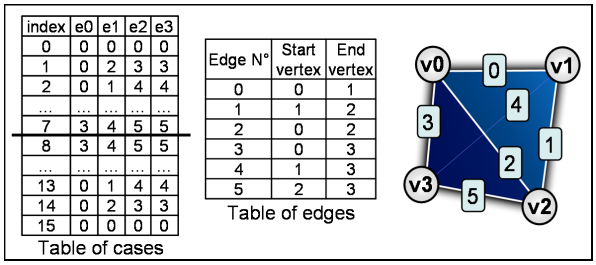
For unstructured tetrahedral meshes, the fastest known GPU hardware accelerated technique was recently introduced by Kipfer et al. [1]. This method is fast, limits data storage redundancy and processing redundancy. However, this method uses the SuperBuffers extension only available on some ATI graphics cards, which are not part of any revision of the Shader Model standard and, therefore, may disappear from the list of supported features on ATI graphics cards. Consequently, we decided to base our work on another more general fast technique shown by Pascucci [2] (also present in other papers [4]). Pascucci proposes a hardware implementation of the Marching Tetrahedra, based on a table of *edges* and a table of *cases*. Its method uses standard features but is only applicable to small and medium-sized grids (less than one million of tetrahedra with 256MB of graphics memory; see Figure 4 for details). Our technique is similar, with the noticeable difference that we completely avoid data redundancy by using indirect indexing.

On a GPU, textures are only reachable on the pixel shading unit for graphics cards supporting Shader Model 2.0, and are reachable in both vertex and pixel shading units for cards fully supporting Shader Model 3.0. Shader Model 3.0 is supported by all Nvidia graphics cards of 6th generation and above; ATI does not currently support this functionality, but claims that its next generation of products will. Previous approaches [2, 4] use the vertex shading unit to implicitly extract the isosurface. They send to the GPU four numbered vertices for each tetrahedron, since an iso-polygon in a tetrahedron contains at most four vertices. The number of the iso-polygon vertex, the geometry of the tetrahedron vertices and the corresponding scalar are also streamed to the GPU through variable registers. Constant GPU registers are used to efficiently store the tables of *edges* and *cases*. For each vertex sent to the vertex shader unit, the Marching Tetrahedra algorithm is applied: compute the entry in the table of *cases*, find the edge corresponding to the vertex number (from 0 to 3), seek the extremities of this edge in the table of *edges*, compute the intersection with the isosurface, then move the current vertex to this position. When the iso-polygon is a triangle, the extra vertex sent is stacked at the same place than the last found intersection, and hence is ignored by the graphics card.

It is possible to combine this hardware accelerated isosurface extraction with algorithmic optimization by sending to the GPU only the intersected cells by using, e.g., an Interval Tree [8], an Octree [9] or a Seed Set [5].

1.3 Contributions

Pascucci's method is similar to ours, but one of the strongest bottlenecks of the Pascucci implementation is that sending all required data to the GPU for each tetrahedron introduces a strong redundancy, since most of the vertices are



index	e0	e1	e2	e3
0	0	0	0	0
1	0	2	3	3
2	0	1	4	4
...	...	...	...	...
7	3	4	5	5
8	3	4	5	5
...	...	...	...	...
13	0	1	4	4
14	0	2	3	3
15	0	0	0	0

Table of cases

Edge N°	Start vertex	End vertex
0	0	1
1	1	2
2	0	2
3	0	3
4	1	3
5	2	3

Table of edges

Fig. 1. Tables of *cases* and *edges*.

shared between several tetrahedra. The GPU-accelerated method presented in this paper overcomes these limitations by:

- avoiding data redundancy through shared vertices
- limiting AGP/PCI-Express bus transfers to a minimum
- efficiently storing the whole data inside a texture to improve isosurface extraction performance for large grids
- handling grids made of millions of cells

Moreover, our method also supports both brute force extraction and combination between GPU tessellation and CPU algorithmic filtering of non-intersected cells like an Interval Tree, an Octree or a Seed Set. In the next section, we describe our method to tessellate tetrahedra in very large grids. For this, we use vertex texture lookup, introduced in the Shader Model 3.0 standard, corresponding to the Nvidia 6th generation (and above) graphics cards.

2 Our Approach

2.1 Preprocessing

Adapted Data Storage for a GPU. The tables of *cases* and *edges* (Figure 1) are defined during the preprocessing stage. The table of *cases* is symmetric, so only the first half of the table must be stored. To handle every possible configuration of the isosurface within a tetrahedron, some edge indexes are duplicated in the table of *cases*. This means that the corresponding vertices will be stacked at the same location and discarded during the real-time rendering stage.

The central question is how to store these two description tables and all required data to efficiently feed a GPU. Previous approaches [2, 4] send geometry, values and both tables of *edges* and *cases* through GPU registers. These approaches send for each cell every needed data, and, since most of vertices are shared between several cells, there is a strong redundancy which consumes a lot of graphics memory and calls for more RAM/Graphics-RAM transfers than necessary. Moreover, the whole data is packed in OpenGL vertex arrays or display lists for efficiency, which are also memory hungry. For a tetrahedral mesh,

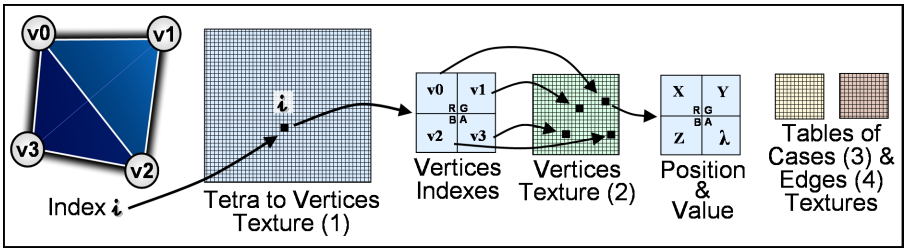


Fig. 2. Texture storage of the tetrahedra, the vertices and the description tables (each texel obviously contains four indexes).

with Reck et al's approach [4], size limitation is about 200k cells with a 128MB graphics card. We propose to push this limit by storing the relevant information in textures in order to deal with grids made of several million cells.

Our goal is to store every needed data in a texture. Four indexes can be stored per texel of a texture, one in each RGBA component. As shown in Figure 2, tables of *edges* and *cases* are considered as 1D tables and are stored sequentially.

The following step of texture generation consists in iterating on each vertex of the grid to pack the corresponding data in a texture. The 3D geometry of the current vertex is packed in one texel, and so uses three of the four components of this texel. The remaining component is used to store the scalar value attached to the vertex. For rendering purposes, one more texel can be used to store a precomputed gradient.

Similarly, the last step iterates on each tetrahedron to pack the index of its four vertices in one texel. On the CPU side, each tetrahedron stores its unique entry in the corresponding texture.

Remark. Using our data storage method (Figure 2), the texture memory usage is evaluated to one texel per vertex without shading and one more texel per tetrahedron, plus few texels for the storage of the description tables. For a 128MB graphics card, a theoretical calculus shows that about 7 million tetrahedra could be stored on the graphics memory. This theoretical limit is higher than the practical limit because the graphics memory is not exclusively used for storing our textures. Nevertheless, current graphics cards commonly board 512MB (up to 1GB sometimes) of graphics memory. See results section 3 for more details.

2.2 Real-Time Rendering

Overview (Brute-force algorithm) After preprocessing, the isosurface extraction proceeds as follows:

- Done Once (Step 1)
  - Load the texture in the graphics memory
  - Load the vertex shading program

  - Set the isovalue
- For Each Cell (Step 2, see details below)
  - Send current cell index to the GPU
  - Send four vertices to the GPU, and implicitly execute the vertex shading program
- Updating (Step 3)
  - Update isovalue and go to Step 2

Sending Vertices to the GPU. The requested vertex number is sent to the vertex shading program through the position argument of the OpenGL vertex creation call:

```
glBegin(GL\_QUAD) ;
  glVertex2i(0, 0) ; // sends vertex 0
  glVertex2i(1, 1) ; // sends vertex 1
  glVertex2i(2, 2) ; // sends vertex 2
  glVertex2i(3, 3) ; // sends vertex 3
glEnd() ;
```

Vertex Shading Program. To code our vertex shading programs, we decided to use the high level programming language from Nvidia called CG[4], which is compatible with both ATI and Nvidia graphics cards. In this subsection, the pseudo CG code for extracting an isosurface from a tetrahedron is provided.

First, the algorithm reads the indexes of the vertices of the currently processed tetrahedron in the `tetra_texture` ((1) in Fig.2). From these indexes, the locations and values of the tetrahedron vertices are read in the `vertices_texture` ((2) in Fig.2):

```
float4 verticesIndex = tex1D ( tetra_texture, index_tetrahedron ) ;
float4 vertex_0 = tex1D ( vertices_texture, verticesIndex.r ) ;
float4 vertex_1 = tex1D ( vertices_texture, verticesIndex.g ) ;
float4 vertex_2 = tex1D ( vertices_texture, verticesIndex.b ) ;
float4 vertex_3 = tex1D ( vertices_texture, verticesIndex.a ) ;
float4 values = float4(vertex_0.a, vertex_1.a, vertex_2.a, vertex_3.a );
```

where tex1D is a function which performs a 1D texture lookup.

Then, the four vertices are assigned a 1 or a 0 flag depending on the isovalue, and the index in the table of *cases* is computed to determine the current configuration:

```
bool4 tested_vertices = ( values >= isovalue ) ;
int index = dot ( tested_vertices, float4(1,2,4,8) ) ;
// The symmetry of the table of cases is exploited
if ( index >= 8 ) { index = 15 - index } ;
```

According to the number of the vertex being processed (from 0 to 3, denoted `vertex_number`), the index of the intersected edge and of its end-points are retrieved from the `table_of_cases_texture` ((3) in Fig.2) and the `table_of_edges_texture` ((4) in Fig.2):

```
int intersected_edge = tex1D(table_of_cases_texture,index*4+vertex_number);
int vertex_index_0   = tex1D(table_of_edges_texture,intersected_edge*2   );
int vertex_index_1   = tex1D(table_of_edges_texture,intersected_edge*2+1 );
```

The vertex shading program then linearly interpolates the intersection of the edge with the isosurface, and finally moves the isosurface vertex to this location.

[4] http://developer.nvidia.com/object/cg_toolkit.html

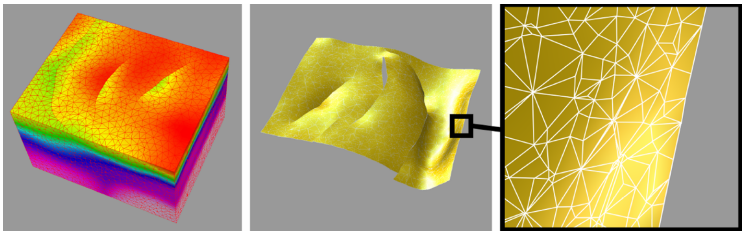


Fig. 3. Isosurface rendering of the fluid pressure field in a tetrahedralized geological model.

Comments. Texture lookup in vertex shader unit is, at this time, relatively slow. Nvidia's documentation[5] indicates that a Geforce 6800 is theoretically capable of processing more than 600 million vertices per second. Add to this bench a vertex texture lookup for each vertex, and the performance falls to 33 million processed vertices per second. This drop of performance is due to long latencies introduced when a texture lookup is requested. 'Latency' means that vertex shading program could execute some assembly codes which are not related to the texture lookup (even other texture lookups) while waiting for this lookup. Therefore, grouping the texture lookups in the vertex shading program parallelizes texture lookup latencies and optimizes performance.

3 Results & Comparisons

Our algorithm has been tested on several tetrahedral grids used in geological modeling, made of up to 5 million cells (see for instance Fig.3). All benchmarks (Fig.4) are done on a laptop with an Intel Centrino 2Ghz processor with 1GB of RAM and an Nvidia QuadroFX Go 1400 256MB PCI-Express graphics card (6th generation from this manufacturer). This card has only 3 vertex shading units, against 8 units in a 7th generation card: using the last 7th generation from Nvidia would have further increased the differences between the CPU and the GPU methods. Notice that the desktop equivalent to our testing graphics card is now a low cost hardware.

Figure 4 presents the number of tessellated tetrahedra per second in a brute-force algorithm with shading for several grid sizes using different methods: a pure CPU based, a GPU register based [2, 4] and our GPU texture based extraction algorithm. The GPU register curve can be split into three parts: below 600K tetrahedra, the whole grid fits in graphics memory, figuring a constant processing speed of about 6.7 million tetrahedra per second; between 600k tetrahedra and about 1 million, the grid does not fit in graphics memory and the PCI-Express bus is used to swap some data with the RAM, resulting in an important performance drop; for more than one million tetrahedra, both the 256MB of

[5] http://developer.nvidia.com/object/using_vertex_textures.html

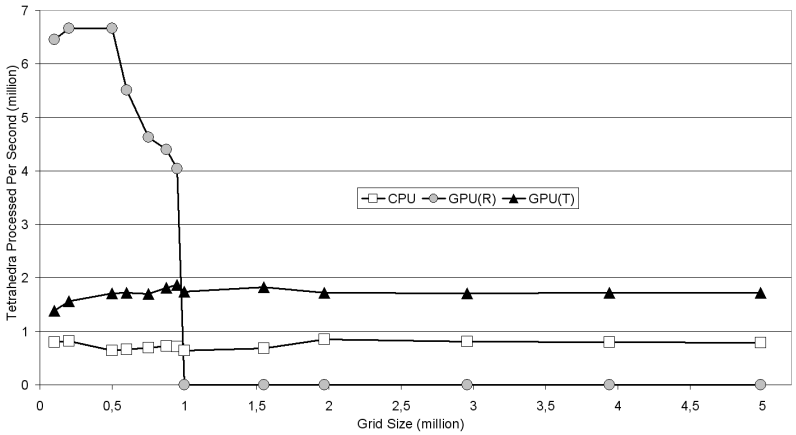


Fig. 4. Number of tetrahedra tessellated per second versus the size of the grid in brute force mode with shading. GPU(R) denotes GPU method using Registers [2, 4] and GPU(T) our method using Textures.

graphics memory and the 1GB RAM are fully loaded, and then performance drops tremendously, since swapping between hard-drive and PC memory is necessary. Pascucci and Reck et al. [2, 4] note that performance drops by a factor of 10 to 20 with an AGP graphics card when the whole data do not fit in video memory. They also suggest that using a PCI-Express card may greatly help, which is confirmed by this work. In our tests, at the limit of a grid counting 1 million tetrahedra, the speed is still near 4 million tetrahedra processed per second (to compare with about 0.5 million processed per second using an AGP card). Both CPU and GPU texture-based methods are linear, and our texture-based method applied to tetrahedra appears to be, on average, twice faster. It is, hence, slower than the GPU register-based method but only for grids made of less than 1 million tetrahedra. Above (up to at least 5 million tetrahedra), our method remains linear. Thanks to these benchmarks, choosing the fastest method for isosurface extraction can be done automatically depending on the grid size and the available memory (RAM and graphics memory).

Comments. The results above were obtained using a brute-force grid traversal to prove the efficiency of the compared algorithms. These methods, including ours, can be combined with algorithms that narrow marching time by previously discarding non-intersected cells (section 1.2). In this case, the whole texture is uploaded once to the graphics memory and only the list of indexes corresponding to intersected cells is sent to the graphics card at the rendering stage. For example, with our approach combined with an Interval Tree [8], we get an acceleration factor of about 4 times on average.

Latencies in accessing textures in the vertex shading unit will soon be im-

proved, according to manufacturers' plans for their graphics cards supporting Shader Models 3.0 and 4.0. These accesses would be as fast as accessing a texture in the pixel shading unit, so about as fast as accessing a register. Then, our method would combine the speed of the register based approaches [2, 4] and the advantages of the texture based storage. Also, most manufacturers[6] are considering the possibility of unifying the vertex and pixel shading units under only one unique hardware unit. Since the pixel shading unit is much more powerful than the vertex shading unit, our method would be strongly accelerated by this evolution. Moreover, these manufacturers will introduce the Geometry Shading Unit in their next generation of graphics cards, enabling the creation of vertices within the GPU. This functionality will limit the redundancy of the computation of the configuration index, and will speed up the extraction process.

4 Conclusion

Our method introduces a hardware-accelerated algorithm to efficiently tessellate very large unstructured tetrahedral meshes, as used for finite element modeling. This method overcomes several limitations by using textures to store efficiently the whole data without introducing redundancies through shared vertices, and limits the AGP/PCI-Express transfers. Our technique handles grids up to five million tetrahedra at least, while improving tessellation performance as compared to CPU extraction. Moreover, it supports both brute-force extraction and extraction after discarding non-intersected cells using, e.g., an Interval tree, an Octree, a Seed Set, etc...

Future works include studying the extraction of isosurface on strongly heterogeneous grids using GPU acceleration; the first step in this direction will be the support for hexahedral meshes. Moreover, non-linear interpolation in isopolygons could be interesting to improve rendering quality. This study could also be extended to time-varying data, volume-rendering and parallel processing.

Acknowledgements

The authors thank the members of the GOCAD research consortium for their support(www.gocad.org), especially EarthDecision for providing the Gocad software. The authors also thank T. Frank and L. Macé for their help.

References

1. Kipfer, P., Westermann, R.: GPU construction and transparent rendering of isosurfaces. In Greiner, G., Hornegger, J., Niemann, H., Stamminger, M., eds.: Proc Vision, Modeling and Visualization 2005, IOS Press, infix (2005) 241–248

[6] http://downloads.gamedev.net/features/programming/atid3d10/ATI-TheFutureofPCGaming.pdf

2. Pascucci, V.: Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In: IEEE TVCG Symposium on Visualization '04. (2004) 293–300
3. Klein, T., Stegmaier, S., Ertl, T.: Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In: Proc. 12th Pacific Conference on Computer Graphics and Applications (PG'04), Washington, DC, USA, IEEE Computer Society (2004) 186–195
4. Reck, F., Dachsbacher, C., Grosso, R., Greiner, G., Stamminger, M.: Realtime isosurface extraction with graphics hardware. In: Proc. Eurographics. (2004)
5. Bajaj, C.L., Pascucci, V., Schikore, D.R.: Fast isocontouring for improved interactivity. Proc. Symposium on Volume Visualization (1996)
6. van Kreveld, M.: Efficient methods for isoline extraction from a tin. GIS, 10:523-540 (1996)
7. van Kreveld, M., van Oostrum, R., Bajaj, C.L., Schikore, D.R., Pascucci, V.: Contour trees and small seed sets for isosurface traversal. Proc. Thirteenth ACM Symposium on Computational Geometry (1997)
8. McCreight, E.M.: Priority search trees. SIAM J. Comput. 14, 257-276 (1985)
9. Wilhelms, J., Gelder, A.V.: Octrees for faster isosurface generation. ACM Trans. Graph. **11** (1992) 201–227
10. Silva, C.T., Mitchell, J.S.B.: The lazy sweep ray casting algorithm for rendering irregular grids. IEEE Transactions on Visualization and Computer Graphics **3** (1997) 142–157
11. Livnat, Y., Shen, H.W., Johnson, C.R.: A near optimal isosurface extraction algorithm using the span space. IEEE Transactions on Visualization and Computer Graphics **2** (1996) 73–84
12. Cignoni, P., Marino, P., Montani, C., Puppo, E., Scopigno, R.: Speeding up isosurface extraction using interval trees. IEEE Transactions on Visualization and Computer Graphics **3** (1997) 158–170
13. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. ACM SIGGRAPH'87 (1987)
14. Shirley, P., Tuchman, A.A.: A polygonal approximation to direct scalar volume rendering. Proc. San Diego Workshop on Volume Visualization, Computer Graphics **24** (1990) 63–70
15. Bloomenthal, J.: Polygonization of implicit surfaces. Computer Aided Geometric Design **5** (1988) 53–60
16. Lévy, B., Caumon, G., Conreaux, S., Cavin, X.: Circular incident edge lists: a data structure for rendering complex unstructured grids. In Ertl, T., Joy, K., Varshney, A., eds.: Proc. IEEE Visualization. (2001)
17. Caumon, G., Lévy, B., Castanié, L., Paul, J.C.: Visualization of grids conforming to geological structures: a topological approach. Computers and Geosciences **31** (2005) 671–680
18. Engel, K., Kraus, M., Ertl, T.: High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In: Eurographics / SIGGRAPH Workshop on Graphics Hardware '01. Annual Conference Series, Addison-Wesley Publishing Company, Inc. (2001) 9–16
19. Castanié, L., Lévy, B., Bosquet, F.: VolumeExplorer: Roaming large volumes to couple visualization and data processing for oil and gas exploration. In: IEEE Visualization. (2005) 32