Realtime Isosurface Extraction with Graphics Hardware

Frank Reck, Carsten Dachsbacher, Roberto Grosso, Günther Greiner, Marc Stamminger

{Frank.Reck,Carsten.Dachsbacher,Roberto.Grosso,Guenther.Greiner,Marc.Stamminger}@cs.fau.de
Friedrich Alexander Universität Erlangen-Nürnberg, Germany
Computer Graphics Group

Abstract

In this paper we introduce a method for the display of isosurfaces extracted from unstructured tetrahedral grids. Our algorithm completely runs on the graphics hardware. The tetrahedra are streamed into a vertex program, which extracts the surface for the given isovalue and immediately renders it. The triangles are not stored explicitly but are computed during rendering time, so the user can modify the isovalue with immediate feedback. If the tetrahedra entirely fit into video memory, we achieve a throughput of more than nine million tetrahedra per second. Our performance can be further improved by using a hybrid method which pre-selects tetrahedra containing the isovalue. We compare our approach with a pure CPU based implementation which achieves about half the performance of our hardware accelerated method.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

In computational science and engineering, unstructured 3D scalar data sets are a typical result of finte element simulations. The scalar values are computed and stored at the vertices of cells that partition the computational domain. The aim of scientific visualization is to transform such three or more dimensional data sets into pictures or movies. Many different methods for this problem exist. Beside the projected tetrahedra algorithm suggested by Shirley and Tuchman [ST88] the calculation of isosurfaces is another commonly used method. The computation of isosurfaces on the graphic hardware has been an important topic of research since many years. Westermann [WE98] proposed a multipass method based on the blending operation supported by OpenGL. A draw back of this method is that Phong shading is not possible. Röttger [RKE00] has developed a method for computing isosurfaces completely based on the projection of tetrahedra and pre-integrated volume rendering. These techniques are complex and actually intended for volume rendering purposes. Thus, in this work we restrict to a much more simpler method.

We present a new method for the extraction of isosurfaces from unstructured 3D scalar fields defined on tetrahedral meshes (for an overview see [BP99]). An isosurface is the set of points with a common scalar value (isovalue). An isosurface patch within a tetrahedron can be either empty, a triangle, or a quadrilateral. In total we have to differentiate between $2^4 = 16$ cases. Taking advantage of symmetry and rotations, these 16 cases can be reduced to the 3 cases shown in Fig. 1. The isosurface for the 3 cases can be described by zero, one, or two triangles.

In Section 2 we show that that the isosurface computation can be carried out completely on the graphics hardware. Ideally, the entire tetrahedral mesh is stored in graphics memory. Then an expensive transfer of the isosurface representation from main memory to the GPU is not necessary, even if the user modifies the isovalue. For the cases that the grid does not fit into video memory, we use an interval tree data structure to reduce the necessary data transfer to a small amount.

The entire calculation is performed by a vertex program [LKM01]. Tetrahedra are fed into the vertex program as vertices, the out coming vertices form the isosurface patch. Vertex programs cannot generate or delete vertices, so in order to obtain the maximum of four possible output vertices we need to send four vertices into the vertex stage for every tetrahedron. Every vertex needs to contain the entire tetrahedron information.

2. The Algorithm

Our goal is to employ graphics hardware for the isosurfaces computation on tetrahedral grids. For each tetrahedron the edges that intersect the isosurface are needed. First, all vertices of the tetrahedron are classified to have either a higher ($+$) or a lower ($-$) scalar value as the isovalue. An edge with different classification at the end points intersects the isosurface; either none, three or four edges can have intersections. For example the classification $-+--$ has intersecting edges 12, 23, and 24. Table 1 shows all 16 possible cases and the corresponding intersecting edges. Symmetric cases where $+$ and $-$ are swapped are grouped into one line since the intersected edges are identical.

class.	class.	intersecting edges	# tri.
$++++$	$----$	-	0
$+++-$	$---+$	14 24 34	1
$++-+$	$--+-$	13 23 34	1
$+-++$	$-+--$	12 23 24	1
$+---$	$-+++$	12 13 14	1
$++--$	$--++$	13 14 23 24	2
$+--+$	$-++-$	12 13 23 24	2
$+-+-$	$-+-+$	12 14 23 34	2

Table 1: *The 16 possible cases of isosurface extraction.*

For each element, the result can be empty, one triangle, or two triangles sharing one edge, as shown in Figure 1. The empty result is obtained for the cases $++++$ and $----$, one triangle appears if three edges are intersected, and two triangles if four edges are intersected. The number of resulting triangles is shown in the last column of Table 1.

Now ideally, a vertex program would do the classification, make a lookup into Table 1 and output zero vertices, three vertices that form a triangle, or four vertices forming two triangles with a common edge as shown in Fig. 1. But since a vertex program cannot generate multiple output vertices for a single input vertex, we have to adapt to the worst case and send four input vertices into the pipeline for every tetrahedron. As output, four vertices v_i are generated, which are meshed as two triangles sharing one edge. The case that only one or zero triangles are generated is handled by outputting degenerate triangles with zero area, which are discarded by the rasterizer. This approach is illustrated in Fig. 1. For convenience the shader code is also given in an Addendum. Table 2 shows the output generated by our vertex program. In

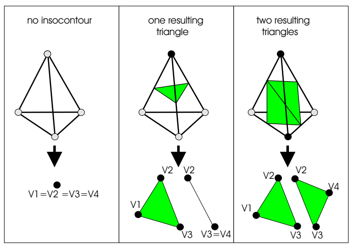

Figure 1: *The three different situations who a tetrahedron can be intersected by the isosurface and the resulting triangulations.*

this table, edge i is the edge that is intersected with the isosurface by input vertex i to obtain output vertex v_i. A “0” means to output an arbitrary constant point without intersection computation.

class.	class.	edge 1	edge 2	edge 3	edge 4
$++++$	$----$	0	0	0	0
$+++-$	$---+$	24	14	24	34
$++-+$	$--+-$	13	13	23	34
$+-++$	$-+--$	12	24	23	24
$+---$	$-+++$	12	14	13	13
$++--$	$--++$	13	14	23	24
$+--+$	$-++-$	12	13	23	24
$+-+-$	$-+-+$	12	14	23	34

Table 2: *The edges that are intersected with the isosurface by our vertex program for all 16 possible cases.*

If we consider the case $-+--$ again, we see that the four output vertices are obtained by intersecting edges 12, 24, 23, and 24 with the isosurface. The table is generated such that the four output vertices can be meshed as a short triangle strip of length four to obtain the desired result. For the previous example, this means that we obtain a triangle from edges 12, 24, and 23. The second triangle with edges 24, 23, and 24 has the first vertex repeated. Thus the resulting triangle has size zero.

Our vertex program has to perform three steps. First, the tetrahedron must be classified. Then the program performs a lookup into Table 2, indexed by the classification pattern and the identifier i. This delivers the edge which is to be intersected by the isosurface. The resulting intersection point is the output of the vertex program. The input vertices are stored in a vertex array, and rendered as indexed face set, where the indices are chosen so that the output stream is meshed as triangle strips of length four. The vertex cache of current graphics boards then avoids the double computation of the shared vertices.

Since we cannot share information between the four processed vertices of a tetrahedron, every input vertex must carry the entire tetrahedron information plus the unique identifier $i \in \{1,2,3,4\}$. Therefore memory requirements are quadrupled. However, this redundance is for purely technical reasons. For future graphics hardware, solutions can be foreseen that allow to get rid of this redundancy, e.g. when vertex programs get access to textures or if textures can be bound as vertex arrays. For the latter experimental implementations already exist.

Using the simple table lookup version as described above, we need a table of size 128 for 16 cases and 4 edges with 2 endpoints each. However, first generation vertex program (vp 1.0) hardware only supports at most 96 constants. We are quite flexible in the ordering of the generated points. We exploit this property to describe an alternative implementation that does not require such a large table, but is only slightly slower.

Depending on i, the vertex program checks three edges of the tetrahedron for an intersection. These edges are shown in Table 3. If $i = 0$, we first look for an intersection on edge 12. In this case, the intersection is computed and output. Otherwise, edge 13 is tested the same way. If yet no intersection is found, edge 24 is tested. If none of these edges is intersected the entire tetrahedron does not contain an isosurface patch. Then we output vertex $(0,0,0,0)$ at each stage to generate an empty patch.

As a result Table 3 produces the same output Table 2, however with a significantly smaller table size.

i	edge test 1	edge test 2	edge test 3
1	12	13	24
2	14	13	24
3	23	24	13
4	34	24	13

Table 3: *Edge traverse order for input vertices i.*

For the computation of the intersections we need to send the entire tetrahedral information with every input vertex i. This is the vertex positions of the tetrahedron and their scalar values. For shading the normal vector is also needed at each vertex. Altogether, this sums up to 144 bytes per vertex (including identifier i, and four additionally swizzeled scalar values to allow further optimization of the vertex shader code).

3. Hybrid CPU/GPU Rendering

The performance of the method can be considerably improved, if those tetrahedra that are cut by isosurface are pre-selected by the CPU. For this purpose the interval tree method [CMM*97] was implemented.

Due to hardware restrictions, the memory requirements are high, so that the entire data set may not fit into video memory. If the entire data has to be transfered to the GPU for each frame, we measured an approximate performance drop of 10-20. However, we can easily combine our method with an interval tree and select those vertices that have to be transferred to the GPU.

4. Results and Conclusion

We tested our implementation on an Intel Pentium 4 with 2.4 GHz using DirectX 9. For the measurements four different data sets have been used: elec1 (19549 tetrahedra), elec2 (142131), valve (35756) and Bluntfin (224874).

The memory consumption is 4×144 bytes per tetrahedron for 4 corner positions, 4 corner normals, and 4 scalar values. The data set is preferably stored in the video memory of the graphics board. Table 4 shows the achieved throughput in millions of tetrahedra in this optimal case. If the data set is stored in main memory instead, and is accessed by the GPU via the AGP bus, processing speed is dramatically reduced by a factor of 10 to 20. Fig. 3 and Fig. 2 show different data sets rendered with our method.

GPU	data set	diffuse lighting	per pixel lighting
nVidia GeForce Quadro FX 3000 256MB	elec1	7.22	6.70
	elec2	6.39	6.33
	valve	7.50	5.68
	Bluntfin	6.41	6.37
nVidia GeForce FX 5800 Ultra 128MB	elec1	7.66	6.70
	elec2	9.92	9.38
	valve	8.48	7.54
	Bluntfin	0.61	-

Table 4: *Million tetrahedra processed per second by the GPU. Performance drop in the last column is due to the fact that the Bluntfin data set does not fit into 128 MB of video memory.*

In Table 5 we compare our algorithm with a CPU based implementation that uses an interval tree for the selection of the intersected tetrahedra, computes the isosurface triangles on the CPU and sends them to the graphic hardware for rendering. If the tetrahedral grid fits into video memory, our algorithm computes isosurfaces almost twice as fast as the CPU based implementation. Although the GPU version iterates over all cells, where most of them do not intersect the isosurface we are already faster. If we use the hybrid CPU/GPU algorithm that only iterates over the intersecting tetrahedra an additional speedup of approximately 7 is achieved. If the isovalue is changed, and new nodes of the interval tree have to be transfered to video memory, frame rate

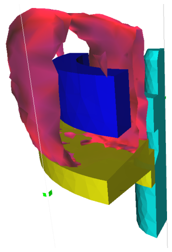

Figure 2: *The strength of the magnetic field around the coil (blue part) is represented by the scalar values of this data set (red).*

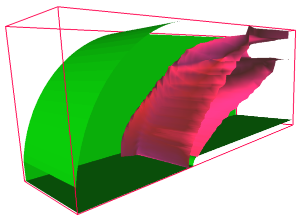

Figure 3: *The Bluntfin data sets hold the result of a flow simulation. The isosurface shows points with the same pressure.*

decreases slightly. A video card with 128 MB can hold over 200.000 tetrahedra. With interval trees data sets can have up to 600.000 tetrahedra. Data sets with more tetrahedra have to be transfered over the AGP bus which results in decreased rendering performance. In the near future we expect that this issue is solved by the increased bandwidth of PCI express. The issue of quadrupling each vertex will be solved by the superbuffer extension. Therefore we predict the same performance gain of 700% also for large data sets in comaprison to a pure CPU based isosurface extraction algorithm.

version	fps
CPU + interval tree	150
GPU brute force	246
GPU + interval tree	1166

Table 5: *Comparison of the pure CPU, GPU variation and an hybrid version of the algorithm in frames per second for the elec1 data set. The interval tree approximately discarded 70 % of all tetrahedra.*

References

[BP99] BAJAJ C., PASCUCCI V.: *Data Visualization Technique*, vol. 6 of *Trends in Software*. John Wiley and Son, 1999, ch. Accelerated isocontouring of scalar fields.

[CMM*97] 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*, 2 (1997), 158–170.

[LKM01] LINDHOLM E., KLIGARD M. J., MORETON H.: A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM Press, pp. 149–158.

[RKE00] RÖTTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the conference on Visualization '00* (2000), IEEE Computer Society Press, pp. 109–116.

[ST88] SHIRLEY P., TUCHMAN A.: A Polygonal Approximation to Direct Scalar Volume Rendering. *San Diego Workshop on Volume Visualization, Computer Graphics 24*, 5 (December 1988), 63–70.

[WE98] WESTERMANN R., ERTL T.: Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM Press, pp. 169–177.

Addendum: Vertex Shader

```
fragment myvs( vertex IN ) {
  fragment OUT;
  float4 pos, F, E, idx, edge1, edge2;
  float  iso1, iso2, frac;

  // finding the intersected edges
  F = IN.isoVswizzle - isoValue.xxxx;
  E = ( F.yxxw * F.wzyw ) < 0.0f;

  // determine edgeTable offset
  idx = dot( E, float3( 1, 2, 4 ) );
  idx    = jumpTable[ idx.x ];
  idx.x += IN.coord1.w;

  // receiving the mask
  edge1 = edgeTable[ idx.x - 2 ];
  edge2 = edgeTable[ idx.x - 1 ];

  // ratio
  iso1  = dot( edge1, IN.isoV );
  iso2  = dot( edge2, IN.isoV );
  frac  = (isoValue-iso1)/(iso2-iso1);

  // interpolate the weights
  E = lerp( edge1, edge2, frac );

  // interpolate position
  pos = IN.coord1 * E.x + IN.coord2 * E.y +
        IN.coord3 * E.z + IN.coord4 * E.w;
  // interpolate normal vector
  OUT.COL0 = IN.nrml1.xyz * E.x + IN.nrml2.xyz * E.y +
             IN.nrml3.xyz * E.z + IN.nrml4.xyz * E.w;

  // correct offset factor
  pos *= idx.wwww;

  // transform position
  OUT.HPOS = mul( worldViewProj, pos );
  return OUT;
}
```