GPU Construction and Transparent Rendering of Iso-Surfaces

Peter Kipfer, Rüdiger Westermann

Computer Graphics & Visualization, Technische Universität München
Boltzmannstrasse 3, 85748 Garching, Germany
Email: {kipfer, westermann}@in.tum.de

Abstract

Iso-surface construction and rendering on programmable graphics hardware has recently been shown for tetrahedral grids. In this paper, we present a novel edge-based approach that avoids redundant computations of edge–surface intersections. We show how to achieve a significant performance gain by considering intrinsic features of recent GPUs. The iso-surface extraction process is re-formulated in a way that reduces both numerical computations and memory access operations. A span-space data structure allows us to avoid the processing of elements not intersected by the selected surface. Finally, to allow for the rendering of transparent surfaces, a GPU sorting routine is integrated into the rendering pass. Our applications show numerical simulation results, distance volumes and advanced shading effects.

1 Introduction and Previous Work

Iso-surface extraction has established itself as a powerful visualization technique for 3D scalar data fields. For reasonably sized data sets, however, this technique can usually not run at interactive rates due to the huge amount of geometric primitives it produces, and which then have to be transfered to the GPU for rendering.

As a matter of fact, since the invention of the Marching Cubes [10] algorithm for iso-surface extraction in 3D hexahedral grids, one avenue of research has led towards interactive surface fitting approaches. Besides the use of hierarchical data structures to minimize the number of elements to be visited during surface construction [21, 19, 2, 18], other approaches try to reduce numerical computations by extracting less accurate surface approximations [11] or by view dependent surface construction [9, 6].

Alternative techniques avoid the construction of a polygonal surface representation by displaying the iso-surface on a per-pixel basis using hardware assisted texture mapping [20, 5] or cell projection [17]. However, it is difficult to simulate transparent surfaces or shadows using such techniques. In addition, these techniques require the entire data sets to be rendered in every frame, while the surface can usually be rendered much more efficiently once it is constructed as a polygonal model.

To overcome these limitations, a number of approaches that perform the Marching Tetrahedra [4] algorithm on the GPU have been presented recently. In [13, 16], the calculation of the iso-surface inside the tetrahedral elements was carried out in the vertex units of programmable graphics hardware. For each element four vertices are processed, resulting in a (possibly degenerate) quad. Because vertices are processed independently by the graphics hardware, the classification of elements as well as the computation of all possible intersection points has to be repeated for every vertex. In addition, the computed geometry cannot be stored in graphics memory, but it has to be rendered directly. As no surface mesh is constructed, the construction process has to be repeated in every frame. Furthermore, none of these methods allow for the rendering of transparent surfaces as they would need to read back the computed geometry for sorting. Smooth interpolation of vertex attributes across the tetrahedral elements was not considered by these approaches.

A significant improvement of GPU-based surface construction in tetrahedral meshes was presented by Klein et al. [8]. First, fragment units have been employed, which provide a much more efficient means for performing the construction step. Second, OpenGL SuperBuffer objects have been used to store the result of the surface extraction step. The SuperBuffer can subsequently be bound as vertex array without any copy operation and

the iso-surface can be redrawn at maximum GPU speed. While this allows the persistent storage of the iso-surface geometry for further processing, it does not improve on the processing required for each generated vertex. Because the algorithm is element-centric, multiple edge-surface intersections and classifications of the element have to be performed for each generated vertex. Additionally, due to intensive use of shader computations, the required shader length poses a major problem in the implementation. Acceleration techniques to reduce the number of elements processed on the GPU as well as transparent surface rendering was not considered.

In this paper, we introduce a new method to construct iso-surfaces from tetrahedral grids. Although this method also has the potential to accelerate CPU surface construction, it is in particular well suited for implementation on the GPU due to its compute and memory access pattern. Similar to the approach by [8], our approach exploits OpenGL SuperBuffers [14, 12] for storing and rendering the iso-surface on the GPU, but it minimizes the number of operations to be performed as well as the amount of data to be accessed on the GPU. This is achieved in two ways: First, element vertices are ordered in a unique way thus enabling an edge-based classification step that is far simpler than the one usually performed, both in terms of the number of numerical computations and the number of memory access operations. Second, an edge-based data structure is employed, which allows minimizing the number of edge-surface intersections. We present an efficient mapping from local intersection points to global vertex indices, which is amenable to standard acceleration structures. Therefore the method can selectively process those tetrahedra that have an intersection with the iso-surface. Smooth shading and GPU sorting for transparent rendering can be achieved in this way.

The remainder of this paper is organized as follows: The next section presents the modification of the Marching Tetrahedra algorithm that is at the core of the improved GPU implementation. Next, the acceleration techniques we have implemented are discussed. While Section 6 presents performance results for the iso-surface extraction, Section 5 demonstrates a number of applications of the proposed technique.

2 Marching Tetrahedra Revisited

The Marching Tetrahedra algorithm is a variant of the Marching Cubes algorithm that can use smaller tables because there are less possibilities a surface can pass through such an element. Because of the linear interpolant inside a tetrahedron, the contained surface is guaranteed to be flat. The Marching Tetrahedra avoids ambiguous cases and requires less numerical computations per element compared to the Marching Cubes algorithm. When converting a hexahedral grid into a tetrahedral one, however, a larger number of surface elements is constructed and only a first order approximation to the real surface is generated. In the Marching Tetrahedra algorithm, a pre-computed case table consisting of 16 cases is used to determine for every element the edges that are intersected by the iso-surface.

The basic idea of our approach is to build an iso-surface extraction algorithm that strictly minimizes the number of operations to be performed. Because an iso-surface is uniquely defined by the intersections of the edges of the 3D mesh, edge-based processing guarantees to find all intersections and to avoid redundant computations.

2.1 Edge-based Classification

In contrast to the standard Marching Tetrahedra algorithm, our approach is based on a particular ordering of element vertices. In this way, we can base our decision on the comparison of the iso-value with the scalar values at two of the four vertices. This results in a minimal number of data values to be fetched in the classification step as well as a minimal length shader to perform the classification. Therefore, we enforce the local enumeration of the vertices of a tetrahedron to be in ascending order with respect to the scalar value they hold. Figure 1 shows the resulting layout.

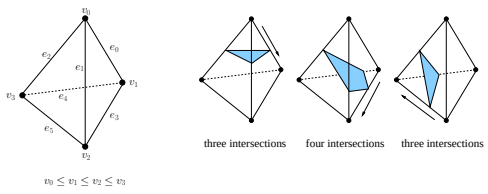

Figure 1: The canonical layout of the tetrahedron and the three basic intersection cases.

According to this setup, only three different cases of how the surface intersects the edges of an element for a specific iso-value i are possible (see Figure 1 on the right):

- **Case 1:** $v_0 \leq i < v_1 \leq v_2 \leq v_3$: edges e_0, e_1, e_2 are intersected. This gives one triangle inside the tetrahedron.
- **Case 2:** $v_0 \leq v_1 < i < v_2 \leq v_3$: edges e_1, e_2, e_3, e_4 are intersected. This gives one quad inside the tetrahedron. Because of the linear interpolant, the quad is guaranteed to be flat.
- **Case 3:** $v_0 \leq v_1 \leq v_2 < i \leq v_3$: edges e_2, e_4, e_5 are intersected. This gives one triangle inside the tetrahedron.

Because of the canonical ordering of the vertices, the decision which case to select can be done by considering edge e_3 alone. If there is an intersection with this edge, the element is classified as a case 2, otherwise it depends on whether the iso-value is less than v_1 (case 1) or larger than v_2 (case 3). Trivial cases, where no intersection between any edge (and therefore with the element) and the iso-surface is found, do not need to be addressed explicitly – they will be implicitly handled as described below. This is much simpler to implement than the classical Marching Tetrahedra approach, as we do only have to consider the scalar values at the vertices v_1 and v_2. From an edge-based perspective, the element classification is implicitly given by the intersection status of element edge e_3. Let us note here, that we never need to perform an element-centric classification and therefore need to evaluate each edge only once. A linear interpolation using a clamped interpolation parameter for all edges of the tetrahedron will trivially give us the correct partial surface.

2.2 Edge-based Data Structure

Previous approaches to GPU iso-surface construction solely employed data structures on a tetrahedral element basis. In such data structures, each element stores indices to its four vertices and associated scalar values. By comparing each scalar value with the iso-value, a bit pattern used to classify the element is built. Finally, linear interpolation along classified edges is carried out to determine the surface–edge intersection points.

The crucial observation here is, that the data structure as described is well suited for processing on the CPU but it cannot be realized efficiently on the GPU. Once a particular element has computed the resultant edge–surface intersection points, these vertices have to be written into a vertex array for rendering. This would imply, however, that a single element—one tetrahedron—has to spawn multiple elements—the intersection points—on the GPU. Unfortunately this can not be realized, and as a matter of fact, the described computation has to be performed repeatedly for the set of all possible intersection points.

This procedure has several limitations. First, for every potentially generated intersection point four scalar values and four floating point coordinates have to be retrieved. Second, all scalar values have to be considered to perform the classification. Third, if multiple elements share an edge, the intersection point along this edge is computed multiple times.

To avoid these drawbacks, we favor an edge-based data structure, which decreases both the amount of data to be stored on the GPU and the amount of data that has to be accessed during iso-surface construction. We build the edge-based data structure as follows. In a floating point texture, we store vertex coordinates and the associated scalar value. In an edge texture, for every edge each texel carries index pairs referencing the two vertices connected by that edge. Hence, every edge can access its vertices as well as the scalar values at these vertices, and it can thus compute the intersection point along that edge. This interpolated intersection point is written into a vertex array equally sized than the edge texture.

By using this data structure, multiple computations of the same intersection point are avoided. Furthermore, if an edge does not find an interior intersection, the fourth component of the intersection point is set to -1 or -2. While the former value indicates the iso-value to be less than v_1 the latter one tells us that the value was larger than v_2. In this way, simply by fetching the intersection point for its edge e_3 can every tetrahedron classify itself according to the iso-value.

We exploit this observation by always constructing a quad inside each tetrahedron. For the cases with three intersections only, we simply repeat the first or the last intersection point producing a degenerate triangle-quad. Because the processing is now edge-based, we're guaranteed to process each edge

only once. This is a huge advantage compared to previous approaches.

3 Extraction algorithm

We implement a three-pass algorithm for creating the iso-surface geometry:

① *Interpolation:* Compute the linear interpolated intersection position with each edge of the tetrahedral mesh.

② *Global indices:* Compute the global vertex indices of the quad for each tetrahedron.

③ *Data arrays:* Map the global indices to linear arrays for rendering. This pass creates vertex and normal arrays and other vertex attribute arrays.

3.1 Interpolation

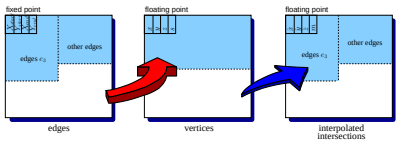

Figure 2: The first pass computes intersections for all edges.

The first pass computes the interpolated intersection position for each edge using the fragment units. We provide the shader with a four component floating point texture that holds the vertex coordinates (the fourth component stores the scalar value s) and an index texture as shown in Figure 2. It contains in the left half all edges e_3 – one for each tetrahedron. This means that an edge can be stored more than once if it is the canonical edge e_3 for more than one tetrahedron. The right half of the edge index texture holds all remaining edges uniquely. All edges are oriented such that the first vertex has the smaller scalar value. The indices are encoded as (u, v) texture coordinates into two fixed point values of sufficient Bit-width. This allows to address any number of vertices up to $64k \times 64k$. The special arrangement of the edges e_3 is crucial for the next pass. The halves may be filled unevenly, so we draw two separate quads to adaptively generate fragments for the shader.

The interpolation shader fetches the two vertices of each edge and interpolates the position according to the given iso-value. The interpolation coefficient is computed straight forward. The computed position will be illegal if the iso-surface does not intersect the tetrahedron, but those positions will be ruled out later on anyway so we ignore that fact here to keep the shader simple. The shader writes the interpolated position $(\hat{x}, \hat{y}, \hat{z})$ to a floating point render target. The fourth component of the output holds a marker m that is set to -1 if the iso-value is smaller than the first vertex and it is set to -2 if it is larger than the second vertex. If there is a correct intersection, m holds the computed interpolation parameter clamped to $[0; 1]$. The following pseudo-code implements the interpolation shader.

```
edge = tex2D(Edges,TCoord[0]);
v0 = tex2D(Vertices,edge.xy);
v1 = tex2D(Vertices,edge.zw);

// we know that v0 has smaller scalar
d = max(v1.w - v0.w, epsilon);
i = clamp((Iso - v0.w) / d);

result = lerp(v0,v1,i);

if (Iso > v1.w) result.w = -2;
else if (Iso < v0.w) result.w = -1;
else result.w = i;
```

3.2 Creating Global Indices

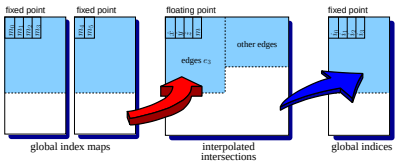

Figure 3: Computing the global indices of the iso-surface vertices.

In the second pass, we need to determine the four global indices for each tetrahedron that form the (possibly degenerate) iso-surface quad. As we have shown above, this can be decided solely based on the status of edge e_3. We therefore can determine the *local* indices of the quad simply by looking at the m component produced for all edges e_3 in the previous pass. Because we have put all edges e_3 on the left side, a fixed point render target half in width nicely fits our needs. Figure 3 illustrates the concept. Here is the table for mapping m to the local quad indices. It is encoded as a constant array in the

shader. Note that it is much smaller than the tables needed in previous approaches. We thus never risk of running out of shader registers.

m	$local_0$	$local_1$	$local_2$	$local_3$
-1	0	1	2	2
$[0;1]$	1	2	3	4
-2	2	2	4	5

The local indices are now mapped to *global* indices by selecting the indicated value from the provided global index maps of the tetrahedron according to its canonical layout. Because the first step produced an intersection position for each edge, these indices now represent the correct partial surface inside the tetrahedron. These maps are static and do not need to be recomputed like it is the case in cell-projection approaches. This operation therefore does not require any dependent texture lookups, as we carefully aligned the index maps and edge e_3 at the same texture coordinate position. Although this sounds like a quite expensive operation, it can be encoded very efficiently in an ARB fragment program using only 17 instructions by exploiting the vectorization capability of the compare statement. Here is pseudo-code for the shader computing the global indices (i_0, i_1, i_2, i_3) from linear 1D edge indices.

```
v = tex2D(InterpVtx, TCoord[0]);

if (v.w == -1)
  // iso smaller than values at edge3
  idx = [0, 1, 2, 2];
else if (v.w == -2)
  // iso larger than values at edge3
  idx = [2, 2, 4, 5];
else
  // flip last two for GL_QUAD draw
  idx = [1, 2, 4, 3];

// get global edge indices of tet
map0 =  tex2D(Map0, TCoord[0]*[2,1]);
map1 =  tex2D(Map1, TCoord[0]*[2,1]);

res = map1.yyyy;
res = (idx < 5) ? map1.xxxx : res;
res = (idx < 4) ? map0.wwww : res;
res = (idx < 3) ? map0.zzzz : res;
res = (idx < 2) ? map0.yyyy : res;
res = (idx < 1) ? map0.xxxx : res;
```

3.3 Creating Vertices

The optional third pass converts the global indices into linear data arrays that can be drawn using the `glDrawArrays` command or can be further processed. We choose to expand the global indices into two arrays in order not to get buffers with large aspect ratios. The shader only needs to do a dependent texture fetch to get the interpolated vertex corresponding to the global index and writes it to a floating point SuperBuffer target that can be bound as vertex, normal, color or attribute array (see Figure 4). The first pass already took care of setting the m component of the interpolated vertex. When creating a vertex array, negative values, i.e. non-valid edge intersections, push the geometry outside the viewing frustum. The shader only has to replace values in $[0;1]$, which signal valid intersection positions, with 1 if rendering using the fixed-function pipeline is required. If one uses custom shaders to draw the iso-surface, the value can simply be ignored or used as parameter to some visualization. Tetrahedra that don't have an intersection at all will therefore automatically be pushed completely outside the viewing frustum.

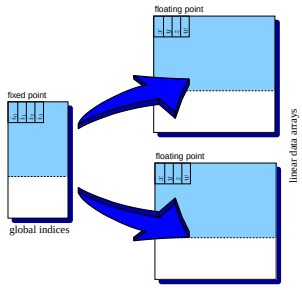

Figure 4: Creating the data arrays of the iso-surface.

Note that if the same approach is taken to produce a normal array for lighting and color or texture coordinate arrays for mapping additional values, sophisticated mappings that choose to sample the remaining edges of the tetrahedron are easy to do, as this information is available from the global index maps that can be accessed directly thanks to the alignment with the current position of the fragment.

The third pass can be skipped, if only the geometry and topology of the iso-surface are desired for further processing or if the application decides to use indexed drawing, e.g. using `glDrawElements`. The global index array from the second pass is already in the correct tightly packed format for rendering quad primitives. The

SuperBuffer specification allows it to be bound as on-GPU index array without copying.

4 Acceleration structures

For a particular iso-value, many of the tetrahedra won't be intersected. Our algorithm will produce quads for them that lie outside the viewing frustum. Although we can safely assume that these quads will be efficiently culled by the rendering pipeline, we like to avoid computing them beforehand. For doing so, many acceleration structures have been proposed in the past to avoid redundant computation. For iso-surface extraction on the GPU, we need a structure that adapts nicely to the instrument driving our computation: the screen-space quad covering the fragments to produce.

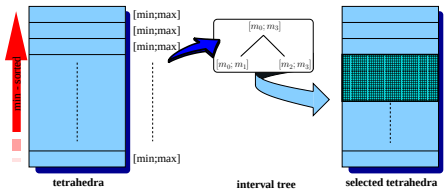

Figure 5: A interval tree built from the per-row min/max scalar values of a sorted tetrahedra field is used to accelerate the iso-surface extraction up to a factor of 2.8.

We observe that using the iso-surface extraction algorithm proposed above, the processing for one tetrahedron is independent of all others, and we can freely rearrange the tetrahedra sequence if we move the edges e_3 and the global tetrahedra index maps accordingly. If we sort the tetrahedra according to the smallest scalar they contain, we get a two dimensional field of which we can determine the minimum and maximum scalar value that occurs per row. Figure 5 illustrates this process. From the min/max intervals we build an interval tree [3]. It can be queried rapidly in which row any concrete iso-value occurs. For accelerating the iso-surface extraction, we want to process only tetrahedra that have an intersection. So we simply determine the smallest and largest row number from the interval tree for the specific iso-value and draw a full-width quad that exactly fits only these rows in height. In order to also rule out the non-contributing tetrahedra in the quad, the first pass can write out appropriate depth values to exploit the early-z culling feature of modern GPUs for the successive passes. However, for rendering the resulting arrays, one has to clear the array buffers with vertex positions that have a negative w component so they won't be drawn. Unfortunately, this optimization offers only limited speedup, as the clear operation is rather expensive as it always fills the entire target buffer whereas the quad identified by the interval tree is often small.

Note also that as we have aligned the intersection interpolation of the edges e_3 and the computation of the global tetrahedra indices exactly with the tetrahedra, the adaptive quad height computed from the interval tree is valid for all passes of our algorithm and therefore accelerates all passes accordingly. This is especially beneficial for the last one that generates the data arrays and attributes as it has the largest output bandwidth. For drawing the iso-surface, the top row of the quad must simply be specified as a starting offset for a call to the `glDrawArrays` command with appropriate array length.

If the dataset has more elements, edges or vertices than can be indexed with the given Bit-width, we split the model into regions that are stored in separate textures. Because processing of the tetrahedra is independent, the regions do not need to have any spatial relation. The procedure takes care, that the global sortedness of the tetrahedra is not changed. Consequently, the interval tree also tells us whether a region can be skipped completely because it doesn't contain any contributing tetrahedra. Sorting the tetrahedra, creating the regions and building the interval tree is done in the preprocessing step. For the bluntfin dataset, the preprocessing needs about 5 seconds. The structure can also easily be saved to disk. At runtime, the interval tree is queried every time the iso-value changes. Our implementation of the interval tree accounts for that by using a B-Tree as basis for the interval tree. The B-Tree can be configured for a fixed tree depth or limited number of elements at its nodes. This allows for fine tuning of the query performance by balancing CPU versus GPU performance depending on how many intervals are accumulated in the tree nodes. For fast GPUs, processing one row more or less doesn't make the difference while saving CPU time by traversing a shallow tree does.

5 Applications

Because passes two and three of our system create results in native OpenGL format used for indexed or array drawing, it can be integrated straightforward into existing rendering systems. This section demonstrates versatile applications of our system to efficiently drive further processing steps. Our system can interpolate any per-vertex attribute for the iso-surface. Figure 4 on the color page shows the ability to create per-face or per-vertex normals for flat or smooth shading and to interpolate texture coordinates for mapping a precomputed 3D LIC texture. We can also make use of OpenGL extensions like `ATI_pn_triangles` to smooth the geometry. In Figure 4 on the color page we render subdivided triangles using cubic interpolation for the vertex positions.

Although our system allows for highly interactive iso-surface extraction rates, one might be interested in displaying multiple level sets at once. For this, the surfaces have to be rendered transparently. Previous approaches needed to read back the geometry for sorting on the CPU or employed depth-peeling. We take advantage of the indices created in pass two and feed them into a GPU sorter [7]. Now the iso-surfaces can be blended correctly in back-to-front order. Figure 2 on the color page shows an example. Note that it's advantageous to use sorting compared to depth peeling, as the complexity of the sorter does not depend on the depth complexity of the iso-surface, which varies and is not trivial to determine. The GPU sorter can sort 7 million items per second. In Figure 5 on the color page we compute smooth per-vertex normals and texture coordinates to apply a third-party glass shader [1] without modifications for convincing refraction.

6 Results

In the following, we present performance results. We provide timings for a system equipped with a P4 3.0 GHz processor and ATI Radeon 9800Pro graphics card. This represents the same hardware that has been used by Klein et al. in [8]. Table 1 shows the performance we achieve using 6 regions, an 8-Bit (u, v) edge map and 1D indexed 16-Bit global index maps. The two pass method means, we only extract the interpolated vertices of the iso-surface and the global index array. The three pass method produces a complete vertex array. The last method extracts a vertex array of the iso-surface, computes per-vertex normals and draws the lit surface onto the screen. On the left column, we have disabled the interval tree and process and draw all tetrahedra. This is the lower performance bound. In the right column, the interval tree selects only contributing tetrahedra. Consequently, the performance varies with the iso-value. The numbers shown here is the maximum performance occurring if the iso-value performs a full sweep from min to max. The bluntfin model was used for these timings where each hexahedral cell is split into 5 tetrahedra (see Figure 3 on the color page for multiple iso-values). Table 2 lists the performance of our method for state-of-the-art graphics hardware.

Method	million tets / sec	
	Interval tree disabled	*Interval tree enabled*
Extract (2 pass)	65.1	83.2
Extract (3 pass)	21.2	52.5
Extract & Render	15.2	42.2

Table 1: Performance on an ATI 9800Pro GPU.

Method	million tets / sec	
	Interval tree disabled	*Interval tree enabled*
Extract (2 pass)	112	143
Extract (3 pass)	43.5	69.4
Extract & Render	28.6	57.1

Table 2: Performance on an ATI X800 XT GPU.

These numbers are valid for datasets that fit into GPU memory. Most of the performance gain of our system comes from the fact that we process each edge intersection interpolation only once. This can be expressed equivalently by comparing the texture read bandwidth of the two systems. Klein's system requires 216 Byte per tetrahedron. If we assume an edge valence of 6 on average, our system reads only 130 Bytes per tetrahedron. Both systems store identical 128 Bit of floating point data per vertex. For storing per-tetrahedron information our system is more efficient and needs only 128 Bit compared to 224 Bit of the previous solution.

The performance of our system could be improved by future hardware that supports three-component fragment shader output to packed target formats or has more output bandwidth. Then we would be able to split the output of the first pass without penalty and render the interpolated intersection position and the marker to two separate tar-

get buffers. The marker can be encoded easily in 8 Bits. The second pass then would only need to fetch from this low bandwidth texture.

7 Conclusion and future work

We have presented a system for iso-surface extraction on the GPU that produces both geometry and topology and stores it in GPU memory for rendering or further processing. Using an innovative edge-based approach, it minimizes both the necessary operations and the shader bandwidth required. Additionally, it provides a more compact storage format than previous approaches. The capability to interpolate arbitrary per-vertex attributes offers a very versatile tool for highly interactive rendering of level sets.

We extended the system with post-processing techniques for high quality rendering of the surface geometry, including advanced shading for mapping additional data on the surface as well as techniques that add additional depth cues to the visualization like high-quality sorted transparency or reflections. Because our system processes iso-surface geometry only in actually contributing tetrahedra, the post-processing is efficient, too.

An exciting future extension would be the integration of more sophisticated automatic acceleration techniques similar to [15] that can be evaluated directly on the GPU. This would improve the timings in Table 2 as the CPU is the limiting factor here. Our approach provides all the geometric and topologic information that is necessary. An implicit occluder can be used to drive the early-z culling feature of the GPU to allow for view-dependent processing. The SuperBuffer technique allows us to render to a compatible render target that can be used as hierarchical z-buffer in successive passes without involving any copy operations.

References

[1] 3Dlabs. Opengl shading language demo. http://developer.3dlabs.com/openGL2/downloads, 2005.

[2] Stephan Bischoff and Leif P. Kobbelt. Isosurface reconstruction with topology control. In *PG '02: Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

[3] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. In *IEEE Transactions on Visualization and Computer Graphics*, volume 3, pages 158–170, 1997.

[4] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. In *IEICE Transactions Commun. Elec. Inf. Syst.*, volume E-74, pages 214–224, 1991.

[5] Klaus Engel, Martin Kraus, and Thomas Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc., 2001.

[6] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K.I. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings of IEEE Conference on Visualization*, pages 475–482, 2002.

[7] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: A GPU-based particle engine. In T. Akenine-Möller and M. McCool, editors, *Proceedings Eurographics Graphics Hardware Conference*, pages 115–122. IEEE, 2004.

[8] 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.

[9] Yarden Livnat and Charles Hansen. View dependent isosurface extraction. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 175–180, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[10] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Computer Graphics (SIGGRAPH 87 Proceedings)*, volume 21, pages 163–169, 1987.

[11] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 281–287. IEEE Computer Society Press, 1994.

[12] nVidia. Data Storage and Transfer in OpenGL. http://developer.nvidia.com/docs/IO/8229/Data-Xfer-Store.pdf.

[13] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Proceedings of IEEE TCVG Symposium on Visualization*, pages 293–300, 2004.

[14] J. Percy. Opengl extensions. http://www.ati.com/developer/SIGGRAPH03/Percy_OpenGL_Extensions_SIG03.pdf, 2003.

[15] S. Pesco, P. Lindstrom, V. Pascucci, and C. Silva. Implicit occluders. In *IEEE/SIGGRAPH Symposium on Volume Visualization*, pages 47–54, 2004.

[16] Frank Reck, Carsten Dachsbacher, Roberto Grosso, Günther Greiner, and Marc Stamminger. Realtime isosurface extraction with graphics hardware. In *Eurographics 2004 Short Presentations*, 2004.

[17] S. Röttger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of IEEE Visualization '00*, pages 109–116, 2000.

[18] Dietmar Saupe and Jürgen Toelke. Optimal memory constrained isosurface extraction. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001*, pages 351–358. Aka GmbH, 2001.

[19] Han-Wei Shen, Charles D. Hansen, Yarden Livnat, and Christopher R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 287–294, 1996.

[20] R. Westermann and T. Ertl. Efficiently using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 169–177, 1998.

[21] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227, July 1992.