Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2004)
O. Deussen, C. Hansen, D.A. Keim, D. Saupe (Editors)

Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping

V. Pascucci[†]

Center for Applied Scientific Computing, LLNL, Livermore,CA, USA.

Abstract

This paper presents a simple approach for rendering isosurfaces of a scalar field. Using the vertex programming capability of commodity graphics cards, we transfer the cost of computing an isosurface from the Central Processing Unit (CPU), running the main application, to the Graphics Processing Unit (GPU), rendering the images. We consider a tetrahedral decomposition of the domain and draw one quadrangle (quad) primitive per tetrahedron. A vertex program transforms the quad into the piece of isosurface within the tetrahedron (see Figure 2). In this way, the main application is only devoted to streaming the vertices of the tetrahedra from main memory to the graphics card. For adaptively refined rectilinear grids, the optimization of this streaming process leads to the definition of a new 3D space-filling curve, which generalizes the 2D Sierpinski curve used for efficient rendering of triangulated terrains. We maintain the simplicity of the scheme when constructing view-dependent adaptive refinements of the domain mesh. In particular, we guarantee the absence of T-junctions by satisfying local bounds in our nested error basis. The expensive stage of fixing cracks in the mesh is completely avoided. We discuss practical tradeoffs in the distribution of the workload between the application and the graphics hardware. With current GPU's it is convenient to perform certain computations on the main CPU. Beyond the performance considerations that will change with the new generations of GPU's this approach has the major advantage of avoiding completely the storage in memory of the isosurface vertices and triangles.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Isosurface computation and rendering I.3.6 [Computer Graphics]: Methodology and Techniques - View-dependent refinement

Keywords: isosurfaces, graphics hardware acceleration, view-dependent refinement, tetrahedral meshes, rectilinear grids.

1. Introduction

Isocontouring is widely used in the visualization of scalar data and an integral component of almost every visualization environment. Computation of isocontours has applications in visualization ranging from extraction of surfaces from medical volume data [Lor95] to computation of stream surfaces for flow visualization [vW93].

Inherent in the selection of an isocontour, defined by $C(w):\{x|F(x)-w=0\}$, is that only a selected subset of the data is represented in the result. In many applications, the ability to interactively modify the isovalue w while viewing the computed result is of great value in exploring the structure of the global scalar field. In fact, it has been observed in user studies that the majority of the time spent interacting with a scientific dataset is devoted to the modification of the visualization parameters, not in changing the viewing parameters [Hai92]. Hence there has been great interest in improving the computational efficiency of isocontouring algorithms [WG92, CMPS96, NH91, PSL*98, SHLJ96, UH99].

In this paper we present a simple and elegant isocontouring approach that exploits new capabilities of commodity graphics hardware. We use a vertex program to perform the interpolation and normal estimation necessary to compute an isosurface. In this way the visualization application does not need to store any auxiliary surface mesh to represent the current isosurface. The selection of an adaptive mesh is also greatly simplified by the use of view-dependent nested

[†] This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. UCRL-JC-149277

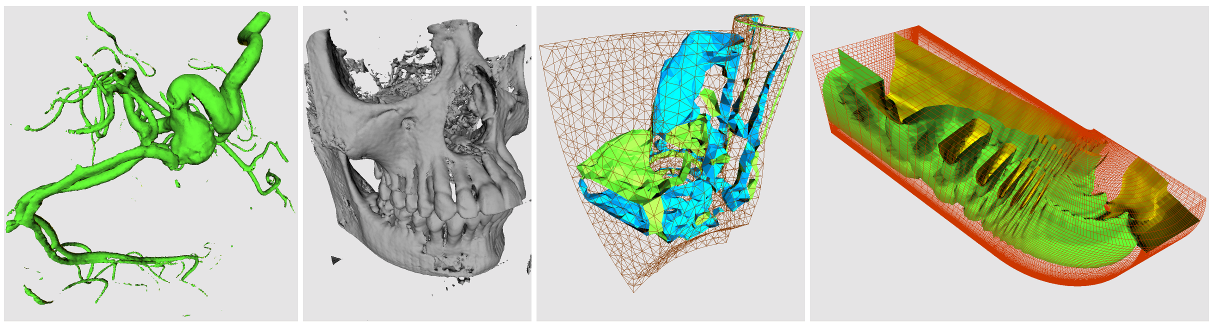


Figure 1: *Four isosurfaces computed entirely by the GPU. The main application sends the vertices of the tetrahedra to the graphics card while a vertex program executed by the GPU computes the edge interpolation and the surface normals. (a) Aneurysm (rectilinear grid). (b) Skull (rectilinear grid). (c) Engine piece (unstructured mesh). (d) Blunt Fin (curvilinear grid).*

errors. Satisfying these errors directly produces consistent meshes. We conclude with a discussion of the potential advantage of using tetrahedral strips for speeding up the data transfer from CPU to GPU. We provide a detailed description of our prototype implementation and report experimental running times obtained in our performance tests.

2. Related Previous Work

Techniques for the efficient computation of isosurfaces date back to the late 80's with the introduction of the Marching Cubes algorithm [LC87]. This algorithm speeds up the isosurface computation by using a reduce lookup table. The main downside of the Marching Cubes algorithm is that it traverses all the cells of the domain mesh independently of the size of the output surface. Wilhelms and Van Gelder [WG92] were the first to propose an optimized data-dependent technique. In their work, they build an octree hierarchy on top of the domain grid and store for each node the minimum and maximum values of the scalar field within the corresponding region of space. During the isosurface extraction this information allows skipping the octree nodes that do not contain the isosurface.

[LSJ96, SHLJ96] define the notion of span space, from which they developed a number of efficient isosurface extraction algorithms. [CMPS96] and [BPS96] introduced at the same time the first optimal algorithm for the extraction of full resolution isosurfaces. The second technique also introduced the use of seed sets that minimize the auxiliary storage of the algorithm. These schemes have been extended to work in external memory [CSS98, CS97], for isosurfaces of arbitrarily large data.

Recent approaches have introduced further optimization using occlusion optimization. Livnat and Hansen [LH98] proposed the first approach of this type. Their main idea is to build image-space occluders using hierarchical tiles while incrementally computing the isosurface. In a related technique Gao and Shen [GS01] propose a parallel multi-pass approach. Each processor computes a piece of the isosurface, separating visible and invisible parts, until the whole surface is completed.

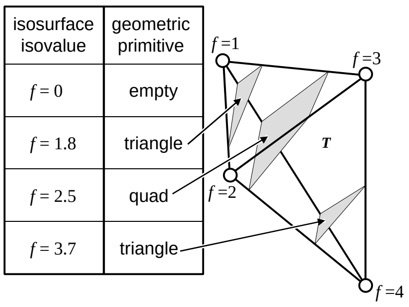
For adaptive refinements of rectilinear grids it is common the use of the longest-edge bisection subdivision rule [ZCK97, PB00, GP00, GDL*02]. [PB00] use the refinement to achieve a time critical isosurface extraction scheme. [GP00] show the use of saturated errors and their extension to detect topological changes in the isosurfaces. We use nested errors that extend the idea of saturated errors to handle the view-dependent case. This is an extension to the volumetric case of the error technique introduced in [LP02].

A recent trend in hardware accelerated techniques [RKE00, WMFC02, GRS*02, WKE02, RE02] is to exploit advanced features of commodity graphics cards to shift some of the computations from the CPU to the GPU. Even if they do not always achieve immediately high performance, they are expected to play an important role in the near future because the speed and internal parallelism of commodity GPU's is improving at a high rate. The technique that we propose differs from most of these approaches in that it computes an isosurface for a tetrahedral mesh, without performing the Shirley-Tuchman decomposition of a tetrahedron [ST90].

3. A Simple Vertex Program

In the OpenGL pipeline a vertex program is a specialized code that is executed by the graphics card each time a glVertex primitive is issued. The rasterization of a triangle or quad is performed only after the current vertex program has mapped the position, normal, color, ..., of all its vertices.

A vertex program has read-only access to 16 Vertex Attribute Registers that store the properties normally set by OpenGL functions like **glNormal**, **glColor**, or **glVertex**. Each register has four floating point components that can be set by the generic command **glVertexAttrib(i,x,y,z,w)**, where **i** is the index of the register and **x,y,z,w** are the floats stored in its components. In addition, a vertex program has read-only access to 96 Constant Registers. They can be changed only outside a **glBegin glEnd** pair. Write-only access is provided for 15 Vertex Result Registers that store the output



isosurface isovalue	geometric primitive
$f=0$	empty
$f=1.8$	triangle
$f=2.5$	quad
$f=3.7$	triangle

Figure 2: *A tetrahedron with function values associated with each vertex. The gray polygons show three possible isosurfaces of the scalar field obtained by linear interpolation of the function values at the vertices.*

of the program. Read-write access is allowed to the twelve temporary registers **R0** through **R11**.

We use a vertex program to perform (i) the linear interpolation along the edges of a tetrahedron to determine the position of the vertices of its isosurface, and (ii) compute the gradient of the function within the tetrahedron to determine the normal of the isosurface.

Consider the tetrahedron T in Figure 2. One real value is associated with each of its vertices to define, by linear interpolation, a scalar field within T. An isosurface of this scalar field can be empty, can be a triangle or can be a quad.

In all cases we draw the isosurface as a quad that may have two or all vertices coincident. OpenGL detects coincident vertices and, when necessary, reduces the quad to a triangle (two coincident vertices) or rejects the quad altogether (three or more coincident vertices). The overall structure of the OpenGL application can be as simple as the following pseudocode. Note that the indentation represents the nesting of the program blocks.

```
set_global_parameters();
set_isovalue();
glBegin(GL_QUADS);          // Start drawing quads
for i=0 to num_tets do:
    set_tet_parameters(i);  // Store vertices in registers
    glVertex2b(0,0);        // Run program four times
    glVertex2b(1,1);        //   with v[OPOS].x set
    glVertex2b(2,2);        //   successively to 0,1,2,3.
    glVertex2b(3,3);
glEnd();                    // Stop drawing quads
```

The function **set_global_parameters** is called only once in the application to load in the Vertex Constant Registers the lookup tables necessary for the isosurface computation. The function **set_isovalue**, called each time the isosurface is changed, stores the new isovalue in one constant register. The coordinates and function values of the vertices of a tetrahedron are set in Vertex Attribute Registers by **set_tet_parameters(i)**. We store these parameters in registers 8 through 11. Assume that the vertices of the mesh are stored in the vector vertices, and that the tetrahedra are stored in the vector tets. One element in tets contains the indices of the vertices of a tetrahedron. A basic implementation of **set_tet_parameters(i)** sets explicitly all four vertices of the *i*-th tetrahedron is as follows.

```
DEF set_tet_parameters(i):
    V0=vertices[tets[i][0]];
    V1=vertices[tets[i][1]];
    V2=vertices[tets[i][2]];
    V3=vertices[tets[i][3]];
    // Store vertices in registers 8,9,10, and 11
    glVertexAttrib( 8, V0.x, V0.y, V0.z, V0.w);
    glVertexAttrib( 9, V1.x, V1.y, V1.z, V1.w);
    glVertexAttrib(10, V2.x, V2.y, V2.z, V2.w);
    glVertexAttrib(11, V3.x, V3.y, V3.z, V3.w);
    return;
```

Interpolation Cases The first step in the program determines what vertices have function value greater or smaller than the current isovalue. An index in the range [0,15] is generated to identify each configuration. The isovalue is stored in the register **c[67].x**, while register **c[66]** stores the constants (1,2,4,8).

```
SGE  R0.w ,v[ 8].w,c[67].x;   # is V0.w >= isovalue ?
SGE  R0.z ,v[ 9].w,c[67].x;   # is V1.w >= isovalue ?
SGE  R0.x ,v[10].w,c[67].x;   # is V2.w >= isovalue ?
SGE  R0.y ,v[11].w,c[67].x;   # is V3.w >= isovalue ?
DP4  R0.x ,R0      ,c[66]  ;  # compute case index
```

The first four statements compare the current isovalue with the function value at the vertices of the tetrahedron. The results of the comparisons – 0 if smaller and 1 if greater or equal – are stored in the four components of **R0**. The dot product of **R0** with (1,2,4,8) yields a distinct code for each configuration in the range [0,15] (stored in **R0.x**).

Lookup Tables We use two lookup tables. The first table – stored in Constant Registers 70 to 81 – defines the edges of the tetrahedron and their endpoints. For example, edge 0 starts at vertex V0 and ends at vertex V1. The complete table is reported below. The first four elements of the same table are also used to select vertices from 0 to 3.

Edge/Vertex Table

Const. Reg.	Edge Selection	V0	V1	V2	V3	Vertex Selection
70	E0 start	1	0	0	0	V0
71	E0 end	0	1	0	0	V1
72	E1 start	0	0	1	0	V2
73	E1 end	0	0	0	1	V3
74	E2 start	1	0	0	0	
75	E2 end	0	0	0	1	
76	E3 start	0	1	0	0	
77	E3 end	0	0	1	0	
78	E4 start	0	1	0	0	
79	E4 end	0	0	0	1	
80	E5 start	1	0	0	0	
81	E5 end	0	0	1	0	

The second lookup table is a standard marching tetrahedral table that lists the edges intersected by the isosurface in each possible configuration. Repeated indices are used to complete the rows of the table corresponding to cases with less than four vertices.

Isosurface Intersection Table.					
Const.	Edge				Interp.
Reg.	0	1	2	3	case
40	70	70	70	70	0
41	74	80	78	78	1
42	80	76	80	80	10
43	74	80	76	78	11
44	70	76	78	78	100
45	70	76	80	74	101
46	70	80	80	78	110
47	70	80	74	74	111
48	80	70	74	74	1000
49	70	78	80	80	1001
50	70	74	80	76	1010
51	76	70	78	78	1011
52	74	78	76	80	1100
53	76	80	80	80	1101
54	80	74	78	78	1110
55	70	70	70	70	1111

A special one-component register **A0.x** can be used to index into Constant Registers array. First **A0.x** is used to read the entry of index **R0.x** into the Isosurface Intersection Table, and copy to **R1** the indices of the edges intersected by the isosurface.

```
ARL  A0.x ,R0.x;        # load case index
MOV  R0   ,c[A0.x+40];  # lookup isosurface case
```

The coordinate passed by **glVertex** (stored in **v[OPOS]**) is the index of the vertex in the current quad. Therefore, it is used to select the edge to be interpolated as follows.

```
ARL    A0.x ,v[OPOS].x ;  # load vertex index
MOV    R1   ,c[A0.x+70] ; # lookup vertex table
DP4    R0.x ,R0 ,R1 ;     # select edge-start vertex
```

Since it is not possible to use the register **A0.x** as an index in the Vertex Properties array, we use the following trick (already used in [WMFC02]) that loads the first vertex of the edge in **R1** and the second vertex of the edge in **R0**. Their function values are loaded in registers **R6** and **R7** respectively.

```
ARL   A0.x   , R0.x;  # load edge-start vertex index
MUL   R1     , v[ 8] , c[A0.x].x ;
MAD   R1     , v[ 9] , c[A0.x].y, R1 ;
MAD   R1     , v[10], c[A0.x].z, R1 ;
MAD   R1     , v[11], c[A0.x].w, R1 ;
MOV   R7     , R1.w ;  # Store the function value in R7
MOV   R1.w   , c[66].x ;  # Set w coordinate to 1
ADD   R0.x   , R0.x ,c[66].x ;  # Increment R0 by 1
ARL   A0.x   , R0.x ;  # load edge-end vertex index
MUL   R0     , v[ 8] , c[A0.x].x ;
MAD   R0     ,v[ 9]  , c[A0.x].y , R0 ;
MAD   R0     , v[10], c[A0.x].z , R0 ;
MAD   R0     , v[11], c[A0.x].w, R0 ;
MOV   R6     , R0.w ;  # Store the function value in R6
MOV   R0.w , c[66].x ;  # Set w coordinate to 1
```

Interpolation Next, we compute the position p of the isosurface vertex along an edge. We use the following interpolation formulas, which are based on the parameter a, on the function value at the endpoints a, b and on the isovalue h.

$$\alpha = \frac{h - f(b)}{f(a) - f(b)} \qquad\qquad p = \alpha\, a + (1-\alpha) b$$

The corresponding implementation is shown below. Note that the result is recorded in the output register **o[HPOS]**, after applying the projection matrix stored in the constant registers **c[0]** through **c[3]**.

```
ADD   R7 , R7       , -R6 ;       # R7 = f(a) − f(b)
RCP   R7 , R7.x ;                 # R7 = 1/(f(a) − f(b))
ADD   R6 , c[67].x , -R6 ;        # R6 = h − f(b)
MUL   R6 , R6       , R7 ;        # R6 = α
ADD   R7 , c[66].x , -R6 ;        # R7 = 1 − α
MUL   R0 , R0       , R7.x ;      # R0 = (1 − α)b
MAD   R1 , R1       , R6.x , R0 ; # R1 = p = α a + (1 − α)b
DP4   o[HPOS].x , c[0]  , R1 ;    # Project and store
DP4   o[HPOS].y , c[1]  , R1 ;
DP4   o[HPOS].z , c[2]  , R1 ;
DP4   o[HPOS].w , c[3]  , R1 ;
```

The remainder of the program simply computes the normal to the isosurface. In particular, we first determine the orientation of tetrahedron from the sign s of the following determinant,

$$s = sign\left(\left\| \begin{array}{ccc} x1 & y1 & z1 \\ x2 & y2 & z2 \\ x3 & y3 & z3 \end{array} \right\| \right)$$

where $xi = Vi.x - V0.x$, $yi = Vi.y - V0.y$, and $zi = Vi.z - V0.z$. Assuming also that $fi = Vi.w - V0.w$, we compute the normal N with three determinants as follows:

$$N = s \left[\left\| \begin{array}{ccc} y1 & z1 & f1 \\ y2 & z2 & f2 \\ y3 & z3 & f3 \end{array} \right\|, \left\| \begin{array}{ccc} z1 & f1 & x1 \\ z2 & f2 & x2 \\ z3 & f3 & x3 \end{array} \right\|, \left\| \begin{array}{ccc} f1 & x1 & y1 \\ f2 & x2 & y2 \\ f3 & x3 & y3 \end{array} \right\| \right]^T .$$

4. View-dependent Refinement

In this section we discuss a simple way to adaptively refine a tetrahedral mesh with the longest-edge-bisection subdivision rule. We implemented the approach to the restricted case of regular grids (not necessarily rectilinear) but the technique is applicable to a more general class of unstructured subdivision meshes [Pas02].

Longest-edge Bisection Consider the decomposition of a cube into six tetrahedra, shown in Figure 3(top left). The repeated subdivision of each tetrahedron by bisecting its longest edge, leads to the same refinement produced by an octree. The only distinction is that three tetrahedral subdivisions are required to generate the same refinement of one step of octree subdivision. We maintain the parallel between the two data structures and call levels of refinement those

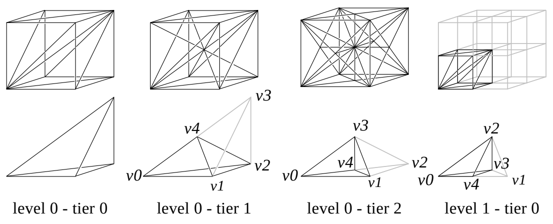


Figure 3: *Four steps of longest-edge-bisection for a rectilinear grid. (top row) Concurrent refinement of tetrahedra and octree cells. Three tiers of tetrahedral refinement are equivalent to one level of octree refinement. (bottom row) A single tetrahedron refined with repeated longest-edge bisections.*

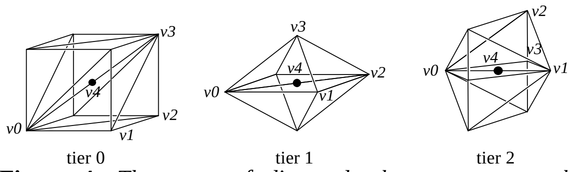


Figure 4: *The types of diamonds that occur at each subdivision-tier. The vertices of one tetrahedron in the diamond are marked consistently to Figure 3(bottom row). Each diamond can be uniquely identified with the vertex v4 where its tetrahedra are bisected.*

generated by an octree, and call tiers the three intermediate refinements of the tetrahedral mesh.

For a cube mesh, the longest-edge bisection can be written in pure combinatorial terms, without measuring the edge lengths. One parent tetrahedron is subdivided into two children tetrahedra using the subdivision rules shown in the table below and illustrated in Figure 3(bottom row).

Longest edge bisection of a tetrahedron T=(v0,v1,v2,v3).				
T tier	children's vertices		children's tier	v4
0	v0,v1,v2,v4	v3,v1,v2,v4	1	(v0+v3)/2
1	v0,v1,v4,v3	v2,v1,v4,v3	2	(v0+v2)/2
2	v0,v4,v2,v3	v1,v4,v2,v3	0 (next level)	(v0+v1)/2

We store auxiliary information on a per diamond basis, where a diamond is formed by the set of tetrahedra that share the same longest edge. For example a tier 0 diamond is a cube. Figure 4 shows the type of diamonds that occur at each tier of the subdivision process. We call *v*4 the center of the diamond and the diamond itself. We say that the diamond *vi* is a parent of *vj* if any tetrahedron in *vi* has a child in *vj*. Note that the hierarchy of the tetrahedra is a binary tree, while the hierarchy of the diamonds is a Directed Acyclic Graph (DAG).

We implement a simple recursive function that splits a tetrahedron until a view dependent error tolerance is met or the finest resolution is reached (l=0). The refinement is also stopped if a tetrahedron is part of a diamond that is outside of the view frustum or that does not contain any piece of isosurface.

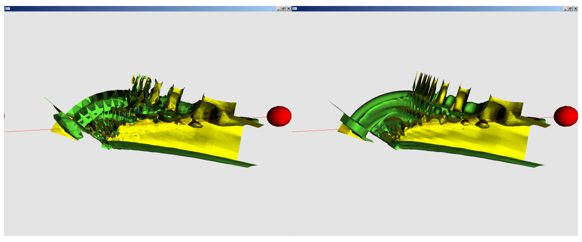


Figure 5: *View-dependent refinement of an isosurface of the Blunt Fin dataset. The view-point is at the center of the red sphere and the view direction is along the red line. (a) Side view of the isosurface refined adaptively. (b) Same view at full resolution.*

```
DEF Refine_Mesh (iso,V0,V1,V2,V3,l,tier):
      V4 = (V1+V3)/2
      if ((l=0) or satisfy_tolerance(V4)) then
          DrawIso (V0,V1,V2,V3)
      else
          if (( min_f [V4]>iso or max_f [V4]<iso) or
             ( not_in_frustum(V4))) then
                return
          if (tier=0) then
                Refine_Mesh (iso,V0,V1,V2,V4,l,1)
                Refine_Mesh (iso,V3,V1,V2,V4,l,1)
          if (tier=1) then
                Refine_Mesh (iso,V0,V1,V4,V2,l,2)
                Refine_Mesh (iso,V2,V1,V4,V2,l,2)
          if (tier=2) then
                Refine_Mesh (iso,V0,V4,V1,V2,l-1,0)
                Refine_Mesh (iso,V1,V4,V1,V2,l-1,0)
      return
```

The arrays **min_f** and **max_f** store respectively the minimum and maximum function values contained in a diamond. For error tolerance and frustum checks, we use the following view-dependent nested evaluations. The diamond *vi* is associated with a bounding sphere *Si* centered in *vi*. If *vi* is an ancestor of *vj*, then the sphere *Si* contains the sphere *Sj*. The function **not_in_frustum(vi)** simply checks if *Si* is outside the view frustum. Similarly, any error associated with *vi* is inflated to be larger then the errors of all its descendants. The function **satisfy_tolerance(vi)** projects the error of *vi* onto the current view plane from the closest point of *Si*. **satisfy_tolerance(vi)** returns true if the projected error is smaller than a given tolerance. In this way we can guarantee that if any diamond is included in the current adaptive mesh, all its parents are also included. Therefore the adaptive mesh has no cracks. Figure 5 shows an example of such refinement for a curvilinear grid.

For a rectilinear grid the radius of the diamonds depends only on the level of resolution and therefore does not need to be stored in the diamonds. If the tolerance is computed only on the basis of the projected size of the tetrahedra,

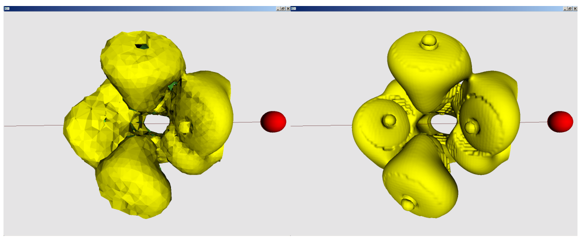


Figure 6: *View-dependent refinement of an isosurface of the neghip dataset. The view-point is at the center of the red sphere and the view direction is along the red line. (a) Side view of the isosurface refined adaptively. (b) Same view at uniform resolution.*

one can achieve a view-dependent refinement without storing any auxiliary information. This is illustrated in Figure 6.

5. Streaming Vertices

Depending on the speed and number of the vertex processing units of the GPU, a possible bottleneck of the rendering operation is the cost of sending the vertices from the CPU to the graphics card. For the vertex program in Section 3 this cost can be greatly reduced using tetrahedral strips. In a tetrahedral strip, any two consecutive tetrahedra differ only by one vertex. In the OpenGL pipeline this can be exploited because the vertex properties are persistent in the vertex registers. The function **set_tet_parameters(i)** used in Section 3 can be replaced by a function **set_new_vertex(i)**, setting only the vertex that transform tetrahedron i into tetrahedron $i+1$ in the strip. The pseudocode becomes.

```
glBegin(GL_QUADS);            // Start drawing quads
for i=0 to num_tets do:
      set_new_vertex (i);     // Store tet vertices in registers
      glVertex2s(0,0);        // Execute with v[OPOS].x=0
      glVertex2s(1,1);        // Execute with v[OPOS].x=1
      glVertex2s(2,2);        // Execute with v[OPOS].x=2
      glVertex2s(3,3);        // Execute with v[OPOS].x=3
glEnd();                      // Stop drawing quads
```

In terms of data flow, the function **set_new_vertex(i)** sends four floats to the graphics card, instead of the sixteen sent by **set_tet_parameters(i)**. The four **glVertex2s** instruction send two short each. Overall, there is a 60% reduction in data transferred (from 80 to 32 bytes per tetrahedron).

In the case of a single-resolution tetrahedral mesh it is easy to build the strips by traversal of the adjacency graph of the mesh. For tetrahedral meshes generated by longest-edge bisection, it is know that is not possible to maintain a tetrahedral strip while performing the simple recursive subdivision discussed in the previous section.

To maintain a tetrahedral strip while performing the simple recursive refinement we adopt the oldest-edge refinement strategy illustrated in Figure 7.

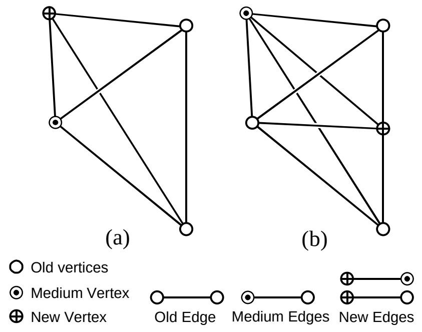


Figure 7: *Oldest-edge refinement strategy. (a) The vertices of a tetrahedron are marked by age and the edges are classified accordingly. (b) The oldest edge is bisected and the age of each vertex is increased.*

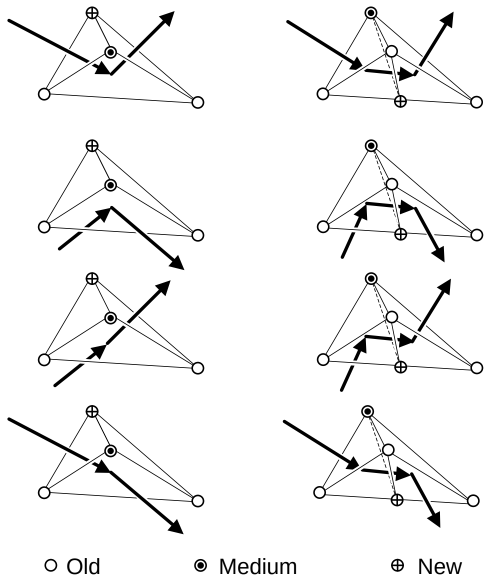


Figure 8: *Oldest-edge refinement strategy. (left column) The only traversal configurations (modulo symmetries) of a tetrahedron within a strip. The arrows indicate the incoming/outgoing face of the tetrahedron. (right column) In all cases it is possible to maintain the continuity of the strip after the bisection of the oldest edge.*

This subdivision is based on a single rule, where the tetrahedron $(v0,v1,v2,v3)$ is bisected at $v4=(v0+v1)/2$ to generated the children $(v0,v2,v3,v4)$ and $(v1,v2,v3,v4)$. Figure 8 shows how to maintain locally the continuity of a strip while bisecting the tetrahedra.

Figure 9 (top two rows) shows the adjacency graph of a complete tetrahedral strip built in this way. This is the 3D generalization of the 2D Sierpinski curve used to render an adaptive refinement of a terrain as a single strip [Paj98]. To the best of our knowledge this 3D space-filling curve was not known before. Figure 9(bottom row) shows two simple (spherical) isosurfaces highlighting the artifacts that may derive from the use of a different mesh.

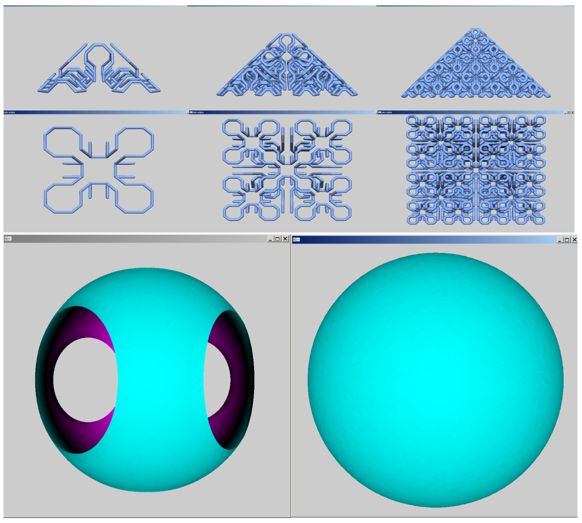


Figure 9: *Three levels of refinement of our 3D space-filling curve. The curve traverses once all the tetrahedra in the mesh, forming a tetrahedral strip. The base mesh is a pyramid with square basis (one sixth of a cube). (top row) Side view. (middle row) Bottom view. (bottom row) Test isosurfaces of a simple scalar field (distance from a point).*

6. Results

We tested the performance of the proposed scheme on two types of NVidia graphics cards: GeForce3 and GeForce4. We have considered three variations of the scheme and reported the results in the table below.

Comparison of average rendering performance in millions of tetrahedra per second.

	isocontour complete	w-out normal	w-out normal + streaming
GeForce4	1.674 Mt/s	2.169 Mt/s	2.192 Mt/s
GeForce3	0.645 Mt/s	0.864 Mt/s	0.864 Mt/s

The complete vertex program computing interpolation and normal is always slower (first column). This confirms the consideration that the slow speed of the GPU becomes quickly the rendering bottleneck when the vertex program is too long. For both graphics cards sending the normals instead of computing them in the vertex program provides a 30increase in performance. Note that this increase in performance comes at the cost of additional storage since the normals are pre-computed.

Adding the streaming component (right column) provides no measurable benefit on the slow card. On the faster card we measure a consistent but marginal benefit. This means that the GeForce4 is at the performance threshold for benefiting sensibly of the streaming component. If, in future cards, the internal parallelism of the vertex processing units will be increased more than the bus bandwidth, we can expect the streaming component to play a more important role.

As expected, the view dependent adaptive refinement has a substantial impact on the interactivity of the application since it allows concentrating the rendering power to the regions closer to the viewpoint. More importantly, it can run with a very small memory footprint since min-max information is sufficient for grids. On a 800Mhz Pentiun PC with 800M of memory (running linux operating system) we can navigate interactively through a 512x512x512 dataset changing the isovalue in real-time.

7. Conclusions and Future Directions

In this paper we have introduced a simple technique for computing isosurfaces of a scalar field exploiting the features provided by commodity graphics cards. In particular we have introduced a vertex program that maps a standard quad into the isosurface contained in a tetrahedron. We also provided an elegant way to generate consistent adaptive refinements of a tetrahedral mesh subdivided by longest-edge bisection.

To optimize the data transfer from the CPU to the GPU we discussed the idea of using tetrahedral strips and introduced a subdivision scheme that allows maintaining strips during the refinement. This scheme yields a new 3D space-filling curve generalizing the Sierpinski curve widely used to optimize the rendering of triangulated terrain datasets.

Since the execution of vertex programs appears to be the current bottleneck of the approach we plan to better exploit the concept of tetrahedral strips. At the moment we are able to provide only one new vertex information per tetrahedron but four vertex programs need to be always executed. Since only three edges are really new, it may be possible to draw one tetrahedron in the strip executing only three vertex programs.

For high-resolution data, we plan to combine the longest-edge and oldest-edge refinements, because the latter asymptotically degrades the aspect ratio of the tetrahedra. To improve the scalability of the scheme for larger datasets, we plan to extend our approach by developing appropriate cache coherent data layouts that may be appropriate for out-of-core execution.

8. Acknowledgments

Special thanks go to Claudio Silva for explaining all the subtilties involved in writing a vertex program. Section 3 is mostly an account of the unpublished details that Claudio presented in his lectures given at LLNL. I insisted in including so many implementation details in the hope that others will benefit from this information when writing their first vertex program.

References

[BPS96] BAJAJ C. L., PASCUCCI V., SCHIKORE D. R.: Fast isocontouring for improved interactivity. In *1996 Volume Visualization Symposium* (Oct. 1996), IEEE, pp. 39–46. ISBN 0-89791-741-3. 2

[CMPS96] CIGNONI P., MONTANI C., PUPPO E., SCOPIGNO R.:

Optimal isosurface extraction from irregular volume data. In *Proceedings of the Symposium on Volume Visualization* (New York, Oct. 28–29 1996), ACM Press, pp. 31–38. 1, 2

[CS97] CHIANG Y.-J., SILVA C. T.: I/O optimal isosurface extraction. In *IEEE Visualization '97 (VIS '97)* (Washington - Brussels - Tokyo, Oct. 1997), IEEE, pp. 293–300. 2

[CSS98] CHIANG Y.-J., SILVA C. T., SCHROEDER W. J.: Interactive out-of-core isosurface extraction. In *IEEE Visualization '98 (VIS '98)* (Washington - Brussels - Tokyo, Oct. 1998), IEEE, pp. 167–174. 2

[GDL*02] GREGORSKI B., DUCHAINEAU M., LINDSTROM P., PASCUCCI V., JOY K. I.: Interactive view-dependent rendering of large IsoSurfaces. In *Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02)* (Piscataway, NJ, Oct. 27– Nov. 1 2002), Moorhead R., Gross M.„ Joy K. I., (Eds.), IEEE Computer Society, pp. 475–484. 2

[GP00] GERSTNER T., PAJAROLA R.: Topology preserving and controlled topology simplifying multiresolution isosurface extraction. In *Proceedings Visualization 2000* (2000), Ertl T., Hamann B.„ Varshney A., (Eds.), IEEE Computer Society Technical Committee on Computer Graphics, pp. 259–266. 2

[GRS*02] GUTHE S., RÖTTGER S., SCHIEBER A., STRASSER W., ERTL T.: High-quality unstructured volume rendering on the PC platform. In *Proceedings of the 17th Eurographics/SIGGRAPH workshop on graphics hardware (EGGH-02)* (New York, Sept. 1–2 2002), Spencer S. N., (Ed.), ACM Press, pp. 119–126. 2

[GS01] GAO J., SHEN H.-W.: Parallel view-dependent isosurface extraction using multi-pass occlusion culling. *2001 Symp. on Parallel and Large-Data Visualization and Graphics* (Oct. 2001), 67–74, 152. 2

[Hai92] HAIMES R.: Techniques for interactive and interrogative scientific volumetric visualization. Available from http://raphael.mit.edu/visual3/unpub.ps, 1992. 1

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algoritm. *Computer graphics 21*, 4 (July 1987), 163–168. 2

[LH98] LIVNAT Y., HANSEN C.: View dependent isosurface extraction. In *IEEE Visualization '98 (VIS '98)* (Washington - Brussels - Tokyo, Oct. 1998), IEEE, pp. 175–180. 2

[Lor95] LORENSEN W.: Marching through the visible man. In *IEEE Visualization '95 (VIS '95)* (Atlanta, Georgia, Oct. 1995), IEEE, pp. 368–373. 1

[LP02] LINDSTROM P., PASCUCCI V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (July/Sept. 2002), 239–254. 2

[LSJ96] LIVNAT Y., SHEN H. W., JOHNSON C. R.: A near optimal isosurface extraction algorithm for structured and unstructured grids. *IEEE Transactions on Visual Computer Graphics 2*, 1 (1996), 73–84. 2

[NH91] NIELSON G. M., HAMANN B.: The asymptotic decider: Removing the ambiguity in marching cubes. In *Visualization '91* (1991), pp. 83–91. 1

[Paj98] PAJAROLA R.: Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings IEEE Visualization'98* (1998), IEEE, pp. 19–26. 5

[Pas02] PASCUCCI V.: Slow growing subdivision (SGS) in any dimension: Towards removing the curse of dimensionality. *Computer Graphics Forum 21*, 3 (Sept. 2002), 451–460. 4

[PB00] PASCUCCI V., BAJAJ C. L.: Time critical adaptive refinement and smoothing. In *Proceedings of the ACM/IEEE Volume Visualization and Graphics Symposium 2000* (Salt lake City, Utah, October 2000), pp. 33–42. 2

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98 (VIS '98)* (Washington - Brussels - Tokyo, Oct. 1998), IEEE, pp. 233–238. 1

[RE02] ROETTGER S., ERTL T.: A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics (VOLVIS-02)* (Piscataway, NJ, Oct. 28–29 2002), Spencer S. N., (Ed.), IEEE, pp. 23–28. 2

[RKE00] RÖTTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings Visualization 2000* (2000), Ertl T., Hamann B.„ Varshney A., (Eds.), IEEE Computer Society Technical Committee on Computer Graphics, pp. 109–116. 2

[SHLJ96] SHEN H. W., HANSEN C. D., LIVNAT Y., JOHNSON C. R.: Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization '96* (1996), pp. 287–294. 1, 2

[ST90] SHIRLEY P., TUCHMAN A.: A polygonal approximation to direct scalar volume rendering. *Computer Graphics 24*, 5 (Nov. 1990), 63–70. 2

[UH99] UDESHI T., HANSEN C. D.: Parallel multipipe rendering for very large isosurface visualization. In *Data Visualization '99*, Gröller E., Löffelmann H.„ Ribarsky W., (Eds.), Eurographics. Springer-Verlag Wien, May 1999, pp. 99–108. 1

[vW93] VAN WIJK J. J.: Implicit stream surfaces. In *Proceedings of the Visualization '93 Conference* (San Jose, CA, Oct. 1993), Nielson G. M., Bergeron D., (Eds.), IEEE Computer Society Press, pp. 245–252. 1

[WG92] WILHELMS J., GELDER A. V.: Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (July 1992), 201–227. 1, 2

[WKE02] WEILER M., KRAUS M., ERTL T.: Hardware-based view-independent cell projection. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics (VOLVIS-02)* (Piscataway, NJ, Oct. 28–29 2002), Spencer S. N., (Ed.), IEEE, pp. 13–22. 2

[WMFC02] WYLIE B., MORELAND K., FISK L., CROSSNO P.: Tetrahedral projection using vertex shaders. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics (VOLVIS-02)* (Piscataway, NJ, Oct. 28–29 2002), Spencer S. N., (Ed.), IEEE, pp. 7–12. 2, 3

[ZCK97] ZHOU Y., CHEN B., KAUFMAN A.: Multi-resolution tetrahedral framework for visualizing regular volume data. In *IEEE Visualization '97 (VIS '97)* (Washington - Brussels - Tokyo, Oct. 1997), IEEE, pp. 135–142. 2