CELL PROJECTION OF MESHES WITH NON-PLANAR FACES

Nelson Max, Peter Williams
Lawrence Livermore National Laboratory, Mail Stop L-560
7000 East Avenue, Livermore, CA 94550, USA
max2@llnl.gov, plw@llnl.gov

Claudio Silva
AT&T Labs - Research
180 Park Av., Room D265
Florham Park, NJ 07932-0971
silva@research.att.com

Abstract: We review the cell projection method of volume rendering, discussing back-to-front cell sorting, and approximations involved in hardware color computation and interpolation. We describe how the method accommodates cells with non-planar faces using view dependent sub-division into tetrahedra.

1. Introduction

Volume rendering converts a scalar function on a 3D volume into varying colors and opacities, and creates an image by integrating the color and opacity effects along viewing rays through each pixel [1]. For data specified on a regular grid, the ray tracing is straightforward [2, 3, 4, 5], and similar effects can be obtained with 3D textures [6]. For curvilinear or irregular grids, these methods are only applicable after the data has been resampled.

An alternative, which works directly on these more general grids, is cell projection [7, 8, 9]. The cells are composited onto the image in back to front sorted order. The projections of the edges of a single cell divide the image plane into polygons, which can be scan converted and composited by standard graphics hardware.

In references [9, 10, 11], we assumed that the cells were polyhedra with planar faces, but this is not always the case. A curvilinear grid is a mapping of a rectangular grid onto a curved volume, for example, to fit next to an airplane wing or ship hull, and quadrilateral faces may map to non-planar surfaces. Irregular grids are fitted to complex geometries, for example mechanical parts, and even faces that are initially flat may become non-planar as the grid elements deform, for example, in a car crash simulation.

Non-planar faces cause problems in the sorting and compositing when a viewing ray intersects the same face twice. We call such a face a "problem face". For example, the ray may leave cell *A* through face *F*, enter cell *B*, and then enter cell *A* again through the same face *F*. If a viewing ray intersects a cell like *A* in two disjoint segments, we call the cell a "problem cell". This makes it impossible to sort cells *A* and *B* in back-to-front compositing order.

Our solution is to divide problem cells into tetrahedra, which have planar faces. A single hexahedron can be projected and composited more quickly than the five or six tetrahedra into which it is subdivided, so we subdivide only the problem cells. In the example above, cell *B* might not turn out to be a problem cell, so it might not be subdivided. However, the face *F* must still be subdivided into two triangles when rendering cell *B*, in order not to create a gap or overlap between cells *A* and *B*. The decision whether a face or a cell is a problem depends on the viewing rays, so the subdivision is view dependent. Therefore our data structure is designed to efficiently replace faces or cells by their subdivisions, and restore them when subdivision is no longer necessary.

A preliminary description of our system appeared in [12], which is a good introduction to back-to-front sorting algorithms. Here, after sketching the subdivision and sorting algorithms in sections 2 and 3 respectively, we give more detail in section 4 on the cell projection and how it is affected by non-planar faces. In sections 5 we describe the data structures designed to handle the view-dependent subdivision, and section 6 gives results.

2. Subdivision

Our HIAC system [11] currently renders "zoo" elements, with the topology, but not the geometry, of tetrahedra, square pyramids, triangular prisms, and cubes, as read from files in the SILO format [13]. The SILO files have an array of vertex positions, and for each of the four cell types, an array of cells, each defined by a list of indices into the vertex array. Thus elements are specified only by their vertex positions, and the cube topology may correspond to a hexahedron with non-planar quadrilateral faces. There is no information about the interpolation function used to define the element shape, so we do not know the shape of the non-planar faces. A natural assumption is that they are hyperbolic paraboloids, resulting from bilinear interpolation between the four vertex positions. But volume rendering using such faces would require tracing rays to intersect the parametrized face surfaces, which is not compatible with hardware-based cell projection of a single face. It could be accomplished by subdivision, for example, using the NVidia GeForce3 tesselation engine [14], but this would require many more polygons to be set up for raster scan conversion.

Instead, we approximate the unknown shapes by piecewise linear interpolation. We divide the problem quadrilateral faces into two planar triangles, and extend this to the problem cells by slicing them into tetrahedra. If a quadrilateral face *F* separates cells *A* and *B*, we must make sure that it is divided by the same diagonal when considered as a face of each. We do this by starting the diagonal from the quadrilateral vertex which has the lowest index in the vertex list. In [12] (and earlier in another context in [15]) is a proof that if the face diagonals are chosen this way, each of the zoo elements can be subdivided into tetrahedra whose edges include the chosen diagonals. The proof starts with the pyramids, which are trivial, observes that a prism can be sliced into a tetrahedron plus a pyramid, and then shows that a cube can be sliced either into five tetrahedra or else into two prisms. We have recently generalized this process to grids of arbitrary convex polyhedra [16].

3. Sorting

For a convex volume filled with convex polyhedral cells, the simple O(n) MPVO algorithm [17] can sort in back-to-front order, using a directed graph. The graph has an edge between every pair of cells sharing a common face *F*, directed towards the cell on the side of the plane of *F* containing the viewpoint. A first pass through the grid counts the incoming directed graph edges for each cell. Cells with zero incoming edges are put on a queue to be output. While this queue is non-empty, the next cell from the queue is added to the end of sorted output list, and its outgoing directed edges are followed to decrement the incoming counts of the cells to which they point. If such a count decrements to zero, the corresponding cell is put on the queue. When the queue becomes empty, all of the cells should have been output. If not, there is a visibility cycle, with each cell partially occluding of the next one, and no sort is possible unless one of the cells in the cycle is subdivided.

If the data volume is convex and a viewing ray passes through cell *A* and later through cell *B*, there is a sequence of cells $A = C_0, C_1, C_2, \ldots, C_n = B$ between them along the ray, each sharing a face with the next, so the directed graph enforces the correct sorting order. This will not be the case if the data volume has concavities or holes which a viewing ray can cross without passing through cells. To deal with such gaps in the viewing rays, extra edges in the directed graph must be added between the cells on the opposite sides of the gaps. In [12], several methods of doing this or its equivalent are discussed. Their input consists of a list of the *b* exterior faces which bound only one cell instead of two, each with a pointer to the single cell it bounds. If each pair of exterior faces must be considered, to see if they are visible to each other across a gap, this could take time O(b^2).

Our system has been under development for over ten years, and we have used a variety of sorting techniques, including the topological sort of the directed graph [9, 17], special sorts for particular application geometries [18], the Newell, Newell, and Sancha sort (see [19] for the polyhedral cell version, and also [10, 11]), the XMPVO sort [20], and the BSP-XMPVO sort [21].

A faster MPVONC sort [17] starts with an O(b log b) sort on the cells with exterior faces, keyed on the distance to the viewpoint of the cells' centers of gravity. For fast rendering, we use this approximate MPVONC sort, instead of one of the slower correct sorts. Recently we have developed the SXMPVO algorithm [22] which scan-converts the exterior faces into an A-buffer, accumulating a sorted list of the exterior faces pierced by the viewing ray through each pixel. The necessary extra edges in the directed graph can then be found because pairs of consecutive entries in these lists bound the empty gaps along the rays. This proved to be the fastest guaranteed correct sort for our purposes.

4. Cell Projection

In the discussion below, the opacity τ refers to the extinction coefficient, or differential opacity, as specified by the scalar value at a 3D point, while the opacity α refers to the total integrated opacity along a ray segment, as used in the compositing. For τ constant on a ray segment of length l, $\alpha = 1 - e^{-\tau l}$ (see [1]).

The RGB color and opacity τ are determined by "transfer functions" which specify how they depend on the scalar variable s being visualized. In this section, we will assume that the transfer functions are linear in s. In fact, our system supports piecewise linear transfer functions, by slicing cells into subcells inside which the transfer functions are linear (see [11, 12]).

The hardware-based method discussed below will be mathematically equivalent to analytic integration along viewing rays when:

(*a*) The projection is orthogonal rather than perspective.
(*b*) The faces of the cells are planar.
(*c*) The scalar s varies linearly across cells, so that R, G, B, and τ are linear.
(*d*) The color is constant per cell, and only τ varies linearly.

Condition (*d*) is a specially strict version of condition (*c*), because the hardware interpolation of colors is not consistent with the analytic integration. As the discussion proceeds, we will explain the approximations we must make when one of these conditions is violated.

Figure 1 shows a hexahedral cell, projected onto the image plane. When we refer to a vertex label like F in this figure, we mean either the 3D vertex position or its 2D projection, depending on the context. The projected edges of the cell divide the image plane up into several polygons, in this case the two triangles *EIH* and *BCJ*, and five quadrilaterals like *FJCG*. These polygons are scan converted and composited into the image by the graphics hardware.

Consider the segment in which the viewing ray through F intersects the cell. The RGBτ values at the front segment endpoint are the ones determined by the data value s at F, but the values at the back endpoint must be interpolated across the face *HDCG*. The integrated opacity for the vertex F is $\alpha = 1 - e^{-\tau l}$ where l is the length of the ray segment, and τ is the average of the differential opacities τ_f and τ_b at the front and back segment endpoints, respectively. To compute l, we must interpolate the depth z across the polygon *HDCG* to get the back segment endpoint, and then l can be found. Similar computations are used at vertex D, with interpolation across polygon *ABFE* used for the front segment endpoint.

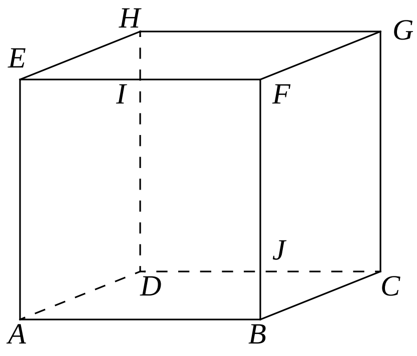

Figure 1. Projection of a hexahedron.

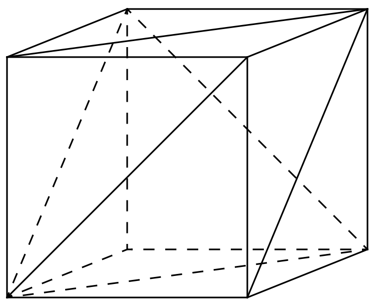

Figure 2. Hexahedron with all faces divided into triangles.

After a perspective projection, the z_s coordinate in screen space is a function of the z_e coordinate in eye space, of the form $z_s = a + b/z_e$. This transformation has the important property that planar surfaces in eye space are transformed to planar surfaces in screen space. Thus the depth z_s for the back endpoint of the ray segment through F can be determined by screen space interpolation on the planar polygon *HDCG*, and then l can be found by reversing the eye to screen coordinate transformation. In perspective, l varies nonlinearly with pixel position, so it should be computed this way at each pixel in polygon *FJCG*, using the z_s values from polygons *FBCG* and *HDCG*. This requires a square root and two divides per pixel. In our implementation, we approximate this by computing l at each vertex, and interpolating linearly across the polygon. We interpolate R, G, B, τ and l linearly across these polygons in screen space, using the standard Gouraud shading and texture coordinate interpolation hardware.

The computations for vertices I and J are simpler, because the necessary R, G, B, τ, and z values are interpolated linearly along the polygon edges. Again, we interpolate these values linearly in screen space, instead of in eye space, but in this case, the correct eye space interpolation is not difficult, because it is only done once per crossing of projected edges.

In the original Shirley and Tuchman method [7], the color assigned to a "thick" vertex, like *F, D, I,* or *J* in figure 1, is the average of the colors at the front and back endpoints of the viewing ray through *F*, and the integrated opacity at *F* is $\alpha = 1 - e^{-\tau l}$ where $\tau = (\tau_f + \tau_b)/2$. The colors at the profile vertices like *A* come directly from the scalar values *s* at these vertices, and the integrated opacities are set to zero. The hardware then interpolates the colors and opacities across the image plane polygons like *FJCG*, and composites them onto the image, using

$$\mathrm{RGB}_{\mathrm{new}} = (1-\alpha) \times \mathrm{RGB}_{\mathrm{old}} + \alpha \times \mathrm{RGB}_{\mathrm{polygon}} . \qquad (1)$$

As pointed out in [10, 11], linear interpolation of integrated opacity α across the polygon is not mathematically equivalent to doing the correct calculation, which requires an exponential per pixel. This can cause unwanted Mach bands in the volume rendered image. Correct values of α can be generated using texture mapping hardware. In an orthogonal view, τ_f and τ_b vary linearly across the polygon in screen space, and therefore so does $\tau = (\tau_f + \tau_b)/2$. If faces *FBCG* and *HDCG* are planar, *l* will also vary linearly across polygon *FJCG*. Thus τ and *l* can be specified as texture coordinates, and linearly interpolated by the hardware. The correct α is then extracted from a 2D texture map, which is preloaded with the values $1 - e^{-\tau l}$.

For a perspective view, τ_f and τ_b should actually be interpolated in eye space. OpenGL can give a correct perspective of a textured surface, with texture coordinates interpolated in eye space, by using appropriately specified *s*, *t*, and *q* texture coordinates [23]. However, τ is the average of τ_f on the front face and τ_b on the back face, and the perspective distortion is different on these two faces, so this feature will not help us. In addition, the correct computation of *l* in perspective would require a square root per pixel. Therefore our texture mapping technique is only an approximation in perspective. We similarly make approximations in the perspective case by interpolating the color components R, G, and B in screen space.

Regarding requirement (*d*) above, the color integrated along the ray is not the average of the front and back colors, weighted by α as in equation (1), because the opacity near the front of the ray segment hides more of the back color. For precise color, the analytic integration described in [11, 24] should be performed once per pixel. However, it requires exponentials, square roots, and evaluations of special functions (the Error function or the Dawson integral), and is thus beyond the per-pixel capabilities of current hardware pipelines. Therefore, we instead compute the correct integrated color only at the thick vertices, and divide by the integrated α value to get the polygon color. Even this can be slow, so we also provide the option to use simply the average of the front and back colors, as in [7].

Another solution is to use 3D textures. Röttger *et al*. [25] interpolate across each subdivision polygon (1) the scalar function s_f on the front face, (2) the scalar function s_b on the back face, and (3) the ray segment length *l*. They use the three interpolated values as addresses into a 3D texture containing the integrated color. The discussion here on the interpolation of color and opacity across cell projections then applies to the interpolation of the scalar function.

So far we have discussed the approximations resulting when conditions (*a*) and (*d*) are violated, and we now turn to conditions (*b*) and (*c*). They are related, since (*b*) concerns the interpolation of z across the faces, and (*c*) concerns the interpolation of R, G, B, and τ across the faces as well as inside the volume. There are standard “linear” finite element interpolation functions for the zoo elements, which are linear on edges and triangular faces, bilinear on quadrilateral faces, and linear (for the tetrahedra only) or trilinear inside the volume. For example, inside a hexahedron, the interpolation is trilinear.

The corresponding interpolation produced by our hardware scheme is hard to determine, because interpolation across polygons is not completely determined in the OpenGL specifications [26]. Instead two possibilities are suggested there: (1) divide the polygon into triangles, and interpolate linearly across the triangles, or (2) divide the polygon into trapezoids by horizontal lines through the vertices, and interpolate bilinearly in the trapezoids, or linearly in the case the trapezoid degenerates into a triangle. Our hardware approximation to the integration along the ray segments assumes a further linear interpolation along the ray segments. Thus it is equivalent to piecewise bilinear interpolation in case (1), and piecewise trilinear interpolation in case (2). This piecewise interpolation is view dependent, since the subdivision into pieces depends on the subdivision of the view plane into polygons, in addition to any further subdivision produced by the hardware. Only in very special viewing situations will it correspond to the interpolation used by the finite element interpolation functions.

Now consider the effect of non-planar faces. If we subdivide each quadrilateral face by the diagonal from its lowest index vertex, the resulting cell will have more edges and faces, so polygonal subdivision of the image plane by the projections of its edges will be more complex, and take longer to compute. For example, figure 2 has 25 polygons, instead of the 7 polygons in figure 1. This will require OpenGL to output more data, and require the hardware to transform more vertices and set up more polygons. However the number of fragments (pixels rendered, see [23]) is still the same, because each pixel in the projection still belongs to exactly one polygon. If the cell were instead subdivided into tetrahedra, the fragment count would also increase, because most pixels would lie inside the projections of several tetrahedra.

If only the problem faces are subdivided, the hardware rendering is equivalent to rendering a cell whose faces are piecewise linear or bilinear. This interpolation is determined partly by the hardware interpolation of cases (1) and (2) above, and partly by the depth values interpolated in software at the other endpoint of a the ray segment through a thick vertex like *F*. In figure

1, the back endpoint F' of the ray through F lies in the interior of the face *HDCG*. We currently choose the z, color, and τ values at F' by subdividing the face *HDCG* with the diagonal from its vertex of lowest index, and then interpolating linearly in screen space across one of the two resulting triangles. Another possibility would be bilinear interpolation, but doing this correctly would require intersecting the viewing ray with the parametrized bilinear surface for the face. This would still not produce the completely correct results for bilinear faces, because the hardware interpolation of the texture parameter l is not the same as doing a ray/surface intersection per pixel.

Similarly, our current method is not consistent with subdividing all the quadrilateral faces into triangles and rendering the polygons in figure 2, because a single polygon from figure 1 will usually overlap several polygons from figure 2. The projection of an interior face will be subdivided differently by the projected edges of each of the two cells it bounds, resulting in two different interpolations of z. This can produce a gap or overlap between the ray segments for these two adjacent cells, causing errors in the volume rendering. In most of our applications, the quadrilaterals are almost planar, and the errors introduced are not large. We also have a slower face subdivision alternative which divides all quadrilaterals into two triangles, and renders all the polygons of figure 2, eliminating these particular z interpolation inconsistencies.

5. Data Structures

As mentioned above, the vertices are stored in a large array. The SILO data has no information on the connectivity of cells across common faces, so this must be reconstructed after the data is input. The data structure for each cell is as follows.

```
struct NewCell {
   char subdivided;        /*currently subdivided */
   char oldsubdivided;     /*subdivided in last frame */
   char numbInbound;       /*for directed graph sort */
   char type;              /*zoo element or new tetrahedron */
   char actual;            /*actual, virtual, or degenerate */
   char cycleTestBit;      /*for detecting visibility cycles */
   char notVisited;        /*for MPVONC sort */
   char nverts;            /*number of vertices */
   int UsedTIndex;         /*into list of cells actually used */
   struct Newcell parent;  /*parent of tetrahedron in block or
       pointer to a block of tetrahedra for subdivided cell */
   struct NewFace **face;  /*head of face list */
   int vertex[4]; }        /*vertex pointers */
```

Extra space for more vertices is allocated if `nverts` is more than four. The list of face pointers is actually an array stored after the array of vertex pointers, but because `nverts` is variable, it must be accessed via a pointer.

The following data structure for the faces allows either one subface, for triangles, or two subfaces, for quadrilaterals.

```
struct NewFace {
   short concave;          /*1 if a problem face; 0 if not */
   short nsubfaces;        /*1 for quadrilateral; 0 if not */
   struct Subface[1];}     /*more space allocated if needed */
```

Each subface is a triangle, and therefore has a linear plane equation, as well as two `shared` pointers to the two cells that share it. The first entry points to the cell with the lowest index; a tetrahedron from a subdivision gets its parent's index. For exterior subfaces, the second pointer is -1.

```
struct Subface {
   struct Newcell *shared[2];
   float A;                /* The plane equation is: */
   float B;                /* Ax + By + Cz + D = 0. */
   float C;
   float D;
   char arrow;}            /* graph edge direction */
```

The `arrow`'s direction is with respect to the first shared cell, `shared[0]`. It indicates whether the viewpoint is on the same side of the plane of the triangle as cell `shared[0]`, on the opposite side, or exactly on the plane. The plane equation is computed once when the geometry is determined, but the arrows must be recomputed each time the viewpoint moves.

The plane equations and arrows for the two subfaces of a quadrilateral face determine whether it is a problem face (setting `concave`) and which of the cells it bounds is a problem cell (setting `subdivided`). See [12] for details. If `subdivided` is true and `oldsubdivided` is false, the cell is subdivided by taking a block from one of four pre-allocated arrays of such blocks, for pyramids, prisms, hexahedra requiring five tetrahedra, and hexahedra requiring six tetrahedra. (Actually, we currently use seven arrays instead of four, because of an earlier attempt to save time by reusing preset face pointers that point to internal faces inside the block, which required a separate list for each block topology.) When a cell is subdivided, the `shared` pointers originally pointing to the cell must be revised to point to the new tetrahedra. This is the reason for including the `shared` pointers in the subfaces; a quadrilateral face pointing to a single cell may later need to point to two subtetrahedra.

If `subdivided` is false, and `oldsubdivided` is true, the parent pointer in the `NewCell` structure is used to restore the pointers to their state prior to subdivision, and the block is placed on a free list for its topological type. Below is the data structure for a block of cells.

```
struct BlockOfCells {
   char type;              /*inherited from parent type */
   char used;              /*1 if currently in use; else 0 */
   short kind;             /*which block topology */
   struct BlockOfCells *next; /*lists of used & free blocks*/
   struct NewCell tets[2];}/*more space allocated if needed */
```

The internal faces are stored directly after the end of the `tets` array, in the memory allocated for the block, to make memory reference more local.

The XMPVO [20] and BSP-XMPVO [21] sorting algorithms require a list of planar exterior triangles, each pointing to the cell they bound. Thus we divide all exterior quadrilateral faces into triangles, whether or not they are problem faces. In the BSP-MPVO algorithm, the required BSP tree is constructed once per new geometry, independent of the viewpoint position. We modified the algorithm so that each exterior triangle points to one of our subface structures, instead of to a cell. Thus we can follow a `shared` pointer to the correct cell, even after view dependent subdivision replaces a subdivided cell by tetrahedra.

6. Results

Figure 3 shows a volume rendering of a curvilinear grid of 19,000 cells on a half-cylindrical shell, twisted so that its faces are non-planar. The color transfer function is piecewise linear, although the resulting colored bands are not visible in this black and white figure. The cells containing contours for the breakpoints of the transfer function are divided into tetrahedra, some of which are further subdivided into slabs on which the transfer functions are linear. In addition, 2589 problem cells were subdivided, giving a total of 31945 cells to be projected and rendered. The subdivision took 0.12 seconds, and the quick approximate MPVONC sort took 0.2 seconds. The total time to project and render the 31945 cells was 14 seconds, using OpenGL and X, and coloring the thick vertices by the simple average of the front and back interpolated colors for their ray segment endpoints. The resolution is 641 by 465. The server was an SGI ONYX with 48 250 MHZ R10000 processors, of which we only used one. We have now parallelized the time consuming image plane subdivision and color integration steps on these multiple processors, as reported in [27].

The client was an SGI Octane with one 250 MHZ R10000 processor, and an ESI graphics board with texture option. When run on the Octane alone, the sorting and projecting time added up to 0.42 seconds, and the total time was 15.4 seconds. The latter increased to 17 seconds when the more accurate color integration was done on the thick vertices, to 31.6 seconds when all quadrilateral faces were divided up into two triangles. When all cells, whether problem cells or not, were divided up into a total of 114,000 tetrahedra, subdivision and sorting time increased to 1.27 seconds and the total rendering time to 40.7 seconds. Software rendering [9, 11], without dividing non-problem cells into tetrahedra, but with analytic integration of the correct color on each pixel's ray segments, took a total of 31.2 seconds. Our source code is available through http://www.llnl.gov/graphics/software.html.

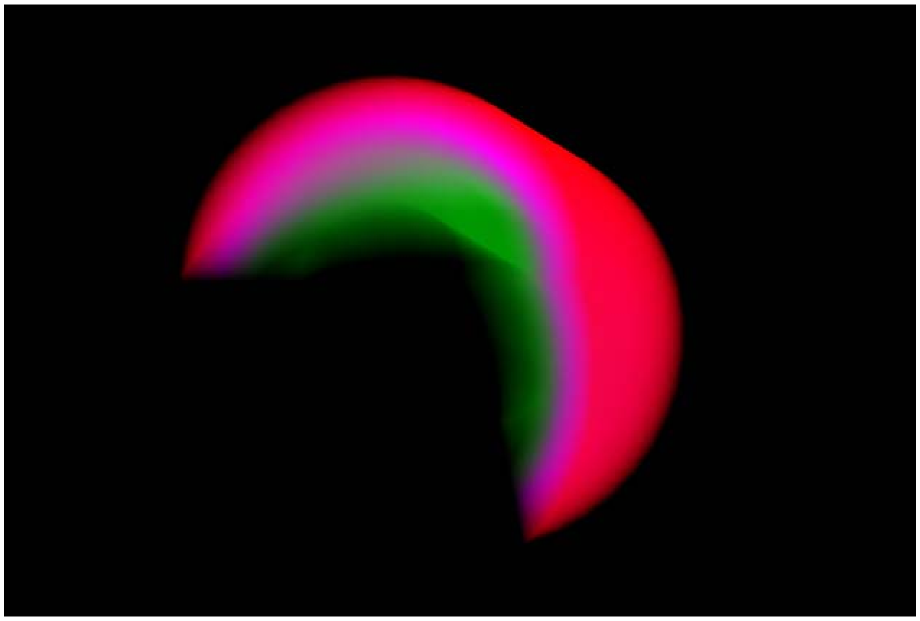

Figure 3. Rendering of a twisted curvilinear grid, initially with 19,000 hexahedral cells.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48, and specifically supported by the Accelerated Strategic Computing Initiative. Dan Schikore wrote the routines to reconstruct the face connectivity for the SILO data.

References

[1] Nelson Max, Optical Models for Direct Volume Rendering, IEEE Transactions on Visualization and Computer Graphics, Vol. 1, No. 2, 1995, pp. 99 - 108.

[2] Marc Levoy, Display of Surfaces from Volume Data, IEEE Computer Graphics and Applications, Vol. 8, No. 3, 1988, pp. 29 - 37.

[3] Robert Dreben, Loren Carpenter, and Pat Hanrahan, Volume Rendering, Computer Graphics Vol. 22, No. 4, 1988, pp. 65 - 74.

[4] Craig Upson and Michael Keeler, VBUFFER: Visible Volume Rendering, Computer Graphics Vol. 22, No. 4, 1988, pp. 59 - 64.

[5] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro Real-Time Ray-casting System, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, 1999, pp. 251-260.

[6] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware, 1994 Symposium on Volume Visualization, pp. 91-98.

[7] Peter Shirley and Alan Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering, Computer Graphics Vol. 24, No. 5, 1990, pp. 63-70.

[8] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering, Computer Graphics (Proceedings of SIGGRAPH 91), 25 (4), 1991, pp. 275-284.

[9] Nelson Max and Pat Hanrahan and Roger Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions, Computer Graphics (San Diego Workshop on Volume Visualization), 1990, 24 (5), pp. 27-33.

[10] Clifford Stein, Barry Becker, and Nelson Max. Sorting and Hardware Assisted Rendering for Volume Visualization, 1994 Symposium on Volume Visualization, ACM SIGGRAPH, pp. 83-90.

[11] Peter Williams, Nelson Max, and Clifford Stein. A High Accuracy Volume Renderer for Unstructured Data, IEEE TVCG, 4(1), 1998, pp. 37-54.

[12] Nelson Max, Peter Williams, and Claudio Silva, Approximate Volume Rendering for Curvilinear and Unstructured Grids by Hardware-Assisted Polyhedron Projection, International Journal of Imaging Systems and Technology, Vol. 11, 2000, pp. 53 - 61.

[13] Silo User's Guide, Revision 1, August 2000, Lawrence Livermore National Laboratory, UCRL-MA-118751, ftp://ftp.llnl.gov/pub/meshtv/meshtv4.1.1/silo.ps .

[14] Henry Moreton, Watertight Tesselation using Forward Differencing, Proceedings of ACM SIGGRAPH / Eurographics Workshop on Graphics Hardware, 2001, pp. 25 - 32.

[15] Gregory Nielson and Junwon Sung, Interval Volume Tetrahedralization, Proceedings of IEEE Visualization 1997, pp. 221 - 228.

[16] Nelson Max, Consistent Subdivision of Convex Polyhedra into Tetrahedra, journal of graphic tools, Vol. 6 (3) 2001, pp. 29 - 36.

[17] Peter L. Williams. Visibility-Ordering Meshed Polyhedra, ACM Transactions on Graphics, 11(2), (April 1992) pp. 103-126.

[18] Nelson Max, Sorting for Polyhedron Compositing, in “Focus on Scientific Visualization,” H. Hagen, H Müller, and G. Nielson, editors, Springer-Verlag, Berlin, 1993, pp. 259 - 268.

[19] Martin Newell, The Utilization of Procedure Models in Digital Image Synthesis, PhD thesis, University of Utah, 1974 (UTEC-CSc-76-218 and NTIS AD/A 039 088/LL).

[20] Claudio Silva, Joseph Mitchell, and Peter Williams, An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes, Proceedings of the 1998 Symposium on Volume Visualization, ACM, 1998, pp. 87 - 94.

[21] João Comba, James Klosowski, Nelson Max, Joseph Mitchell, Claudio Silva, and Peter Williams. Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids, Computer Graphics Forum, 18(3) 1999, pp. 369-376.

[22] Richard Cook, Nelson Max, Claudio Silva, and Peter Williams, “Efficient Exact Visibility Ordering of Unstructured Meshes. Submitted to IEEE TVCG.

[23] Mason Woo, Jackie Neider, and Tom Davis, OpenGL Programming Guide, Second Edition, Addison Wesley, 1997, p. 372.

[24] Peter Williams and Nelson Max. A Volume Density Optical Model, 1992 Workshop on Volume Visualization, (1992), ACM, pp. 61-68.

[25] Stefan Röttger, Martin Kraus, and Thomas Ertl, Hardware-Accelerated Volume and Isosurface Rendering Based on Cell Projection, Proc. IEEE Visualization 2000, pp. 109 - 116.

[26] Mark Segal and Kurt Akeley, The OpenGL Graphics System: A Specification, http://www.opengl.org/Documentation/Specs.html (1998).

[27] Janine Bennett, Richard Cook, Nelson Max, Deborah May, and Peter Williams, Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes of Arbitrary Polyhedra, Proceedings of 2001 Symposium on Parallel and Large-Data Visualization and Graphics, ACM SIGGRAPH, pp. 101 - 106.