An Architecture for Interactive Tetrahedral Volume Rendering

Davis King[1], Craig M. Wittenbrink, Hans J. Wolters
Client and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2000-121 (R.3)
August 2nd, 2001*

computer graphics, scientific visualization, R-buffer, 3D compression

We present a new architecture for interactive unstructured volume rendering. Our system moves all the computations necessary for order-independent transparency and volume scan conversion from the CPU to the graphics hardware, and it makes a software sorting pass unnecessary. It therefore provides the same advantages for volume data that triangle-processing hardware provides for surfaces. To address a remaining bottleneck - the bandwidth between main memory and the graphics processor - we introduce two new primitives, tetrahedral strips and tetrahedral fans. These primitives allow performance improvements in rendering tetrahedral meshes similar to the improvements triangle strips and fans allow in rendering triangle meshes. We provide new techniques for generating tetrahedral strips that achieve, on the average, strip lengths of 17 on representative datasets. The combined effect of our architecture and new primitives is a 72 to 85 times increase in performance over triangle graphics hardware approaches. These improvements make it possible to use volumetric tetrahedral meshes in interactive applications.

* Internal Accession Date Only Approved for External Publication
[1] College of Computing, Georgia Institute of Technology
Presented at and published in the International Workshop on Volume Graphics, June 21-22, 2001, Stony Brook, New York

An Architecture For Interactive Tetrahedral Volume Rendering

Davis King*, Craig M. Wittenbrink, and Hans J. Wolters

Hewlett-Packard Laboratories
1501 Page Mill Rd., Palo Alto, CA 94304
* College of Computing, Georgia Institute of Technology

Abstract. We present a new architecture for interactive unstructured volume rendering. Our system moves all the computations necessary for order-independent transparency and volume scan conversion from the CPU to the graphics hardware, and it makes a software sorting pass unnecessary. It therefore provides the same advantages for volume data that triangle-processing hardware provides for surfaces. To address a remaining bottleneck – the bandwidth between main memory and the graphics processor – we introduce two new primitives, tetrahedral strips and tetrahedral fans. These primitives allow performance improvements in rendering tetrahedral meshes similar to the improvements triangle strips and fans allow in rendering triangle meshes. We provide new techniques for generating tetrahedral strips that achieve, on the average, strip lengths of 17 on representative datasets. The combined effect of our architecture and new primitives is a 72 to 85 times increase in performance over triangle graphics hardware approaches. These improvements make it possible to use volumetric tetrahedral meshes in interactive applications.

1 Introduction

Many interactive 3D graphics applications rely on triangle-based representations and on hardware support for rendering triangles to achieve interactive speeds. While modeling applications often use higher-order primitives such as NURBS or subdivision surfaces, these representations are usually converted to triangles at rendering time to take advantage of hardware acceleration.

Other graphics applications, such as volume rendering, have been slower to achieve sufficient performance for interactive use. Direct hardware support for volume rendering has only recently become available, and only for regular grids at fixed resolutions. Tetrahedral meshes can serve as a more flexible representation for volume rendering, providing locally adaptive resolution, integration with polygons, and fitting to complex boundaries. However, since tetrahedral meshes are more complex than either triangle meshes or regular grids, they have been used primarily in high-end visualizations and finite-element simulations.

Recently, however, renderers for tetrahedral meshes have begun to achieve interactive frame rates by using carefully optimized algorithms for visibility sorting and by taking full advantage of triangle-rendering hardware [20, 23, 17, 26, 29, 30, 19]. While the most immediate application of these algorithms is visualization, the ability to view

tetrahedral meshes interactively will open up new uses for tetrahedral meshes. Such uses will include rendering transparent objects and atmospheric effects; allowing richer forms of volumetric sculpting; and providing visual feedback to accompany simulations of models with accurate density, deformations, and haptic properties. Color Plate Figure 13 shows some examples of rendered tetrahedral grids.

It is possible to address many of the difficulties of rendering tetrahedral meshes in the same way similar difficulties have been solved for triangle meshes: using hardware acceleration for floating-point computations, visibility determination, and rasterization. In this paper we propose such a hardware-based solution. We base our approach on Shirley and Tuchman's projected tetrahedra algorithm [20]. The original algorithm requires first visibility sorting, next classifying the tetrahedra according to their projections into image space, and then splitting each tetrahedron into triangles. Additionally, color and opacity must be computed for the split vertex.

Figure 1 describes the steps in our CellFast approach [30], a modified version of the projected tetrahedra algorithm empirically tuned for fast performance on OpenGL accelerated desktops. Our previous work used current graphics hardware to accelerate only Step VI, the scan conversion of triangles. Steps II-V, meanwhile, saturate the host CPU and leave the graphics hardware running below capacity. The new architecture we propose here eliminates these bottlenecks by moving steps II-V to hardware in addition to step VI. This frees the host CPU from a lot of work, and allows tetrahedral meshes to be rendered as a retained mode display list as follows:

glCallList(globalTetList);

Therefore the host would be lightly loaded, and the primary bottleneck would be the bus to the graphics accelerator, typically an AGP (Accelerated Graphics Port) bus. We propose changing the required work of Figure 1 to the display list call above. The outline of the paper is as follows: In Section 2 we present the new hardware architecture for rendering tetrahedra directly. In Section 3 we discuss tetrahedral strips and tetrahedral fans and describe algorithms for generating these. Section 4 discusses the necessary API to communicate with the graphics subsystem. Section 5 presents results and Section 6 draws some conclusions and hints at future work.

```
I. Preprocess dataset
For a new viewpoint:
II. Visibility sort cells
    For every cell in sorted back-to-front order:
    III. Test plane equations to determine class (1,2,3,4)
    IV. Project and split cell unique frontback faces
    V. Compute color and opacity for thick vertex
    VI. (Hardware) Scanconvert new triangles
```

Fig. 1. CellFast pseudo-code.

2 Hardware Architecture

Our architecture makes three major changes in how to implement Shirley and Tuchman's algorithm [20, 30]. Figures 2 and 3 show the differences in the dataflow on a

desktop graphics system. Each figure shows four major hardware components and the buses between them – the CPU (central processing unit), main memory, the chip set, and the graphics accelerator. Our architecture changes the system in Figure 2 by moving the tetrahedral-processing operations to the hardware; by moving the sorting to the hardware; and by using strips and fans to reduce the load on the bus. Our hardware solution for sorting is similar in spirit to Carpenter's software A-buffer [2]; we describe how a *recirculating fragment buffer* (R-buffer) [31] may be used to implement order-independent transparency in Section 2.1.

Figure 2 shows the dataflow for our CellFast implementation of projected tetrahedra [30] on an OpenGL graphics hardware accelerated desktop. During sorting the tetrahedra require random access for reading and writing. Popular sorts either read and write sphere data [3, 12, 30] or connectivity and/or BSP data [27, 22, 21, 23, 6]. The sorted list is then used to read tetrahedra, where they are split into triangles. The triangles are stored back in main memory, and then finally read into the graphics accelerator. The data for sorting are read and written as much as necessary to perform the sort. The tetrahedra are read once from the main memory; In our experiments an average of 3.4 triangles are written per tetrahedron; and then the triangles are read into the graphics accelerator.

As Figure 3 shows, our new architecture makes the host responsible only for initiating the transfer of tetrahedra to the graphics accelerator. With tetrahedral rendering in hardware, a display list is used with or without vertex arrays. This is a key difference, because a projected tetrahedra approach with only triangle rendering hardware requires immediate mode triangle rendering. The display list would contain all of the tetrahedra of interest. Figure 3 is a considerable advantage in just the bandwidth over the currently possible approach shown in Figure 2. While the likelihood of modifying graphics hardware for the specific application of unstructured volume rendering is low, many new graphics cards like the NVidia GForce2/GForce3 are programmable [14]. Future programmability might be used to implement the steps necessary for tetrahedral mesh rendering. In order to understand the features required, we briefly review the projected tetrahedra algorithm, and where each portion would occur, either the CPU or the graphics hardware.

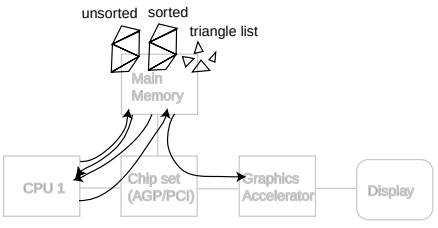

Fig. 2. Dataflow for CellFast.

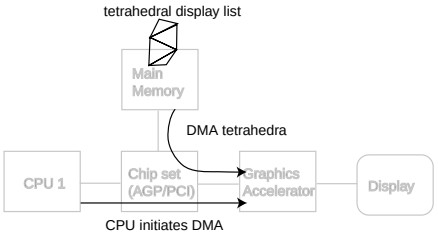

Fig. 3. Dataflow for proposed architecture.

In our experiments with CellFast, we have found that in practice, most of the rendering time in the projected tetrahedra algorithm [20] is due to the floating point operations

needed to process each tetrahedron. These operations depend on the orientation of the tetrahedron and its projection into screen space. The algorithm must process four cases (Figure 4). For each case, the operations involved are simple and are already performed in hardware for other graphics primitives. More precisely, we need to compute the vertex coordinate of the thick or split vertex together with its opacity and color values. A thick vertex is a vertex of the triangles used to represent a tetrahedron, and falls where the tetrahedron has its maximum thickness along the view ray [20]. The thick vertex is labelled for each of the four cases in Figure 4. For example, if the tetrahedron is of Class 1, we need to perform a ray plane intersection to find the point in the interior of the back (or front) face. For a tetrahedron of Class 2 we need to find two points, one point on the front facing edge and one point on the back facing edge. This corresponds to two line-line intersections. Class 3 requires us to find a point on the front facing edge (or back facing respectively). Hence again we have to perform a line-line intersection. Class 4 is a degenerate case with two faces perpendicular to the viewpoint, and no new vertices are computed.

In order to process the tetrahedra and find the split vertices, we may simply reuse the existing geometry capability for performing hardware accelerated line plane and line-line intersections. The same is true for setting up the plane equations for each of the four faces. These are needed to perform the classification in a prior step. What is left is to compute the resulting depth integrated opacity and color at the thick vertex. As outlined in Shirley and Tuchman [20], this is based on Euclidean distances and hence requires the computation of square roots. This is currently implemented as a look up table in most hardware designs. Additionally, we need to perform bilinear and linear interpolations. Figure 5 shows the revised CellFast algorithm possible on an enhanced graphics hardware architecture. The geometry processing to be added to the graphics hardware would implement the steps numbered III, IV, and V. Step II is implemented using the R-buffer discussed in the next section. Steps III, IV, and V use standard floating point operations, and therefore on architectures like HP's fx hardware [13] one only needs to modify the firmware to implement them. Obviously hardwired geometry engines would require hardwired control changes.

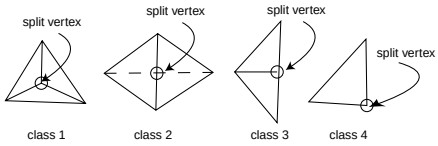

Fig. 4. Classification of projected tetrahedra

III. (**Hardware**) calculate 4 plane equation values
IV.1 (**Hardware**) switch on class type
for each class determine thick vertex and draw triangles
IV.2/V.1 (**Hardware**) class 1A, class1Color()
IV.3/V.2 (**Hardware**) 1B, class1Color()
IV.4/V.3 (**Hardware**) class 2, class2Color()
IV.5/V.4 (**Hardware**) class 3A, class3Color()
IV.6/V.5 (**Hardware**) 3B, class3Color()
IV.7/V.6 (**Hardware**) class 4, class4Color()

VI. (**Hardware**) scanconvert resulting triangles

II. (**Hardware**) R-buffer sorts fragments

Fig. 5. Hardware CellFast rendering.

2.1 Recirculating fragment buffer (R-buffer) for depth sorting

We have developed an invention that we call the *recirculating fragment buffer*, (R-buffer) that computes order-independent transparency with any number of levels economically in hardware [31]. A fragment is a sample representing a pixel on the screen consisting of typically the $RGBAZ$, and is the terminology used in OpenGL. An *R-buffer* is a buffer, separate from the Z buffer and frame buffer, that stores fragments. We are proposing that all of these buffers reside in the same off-chip memory of the graphics accelerator. By the addition of the *R-buffer*, additional comparison and control logic, and a few bits and an extra Z per pixel, we can compute order-independent transparency. Once fragment sorting is supported in hardware, this relieves the host of the requirements for any topological cell sorting, and also eliminates problems often found with cell cycles, sliding interfaces, or concavities.

The Z buffer is simple and fast in hardware. It has dominated graphics architectures for nearly two decades. But Z buffering is a *read modify write*, and so an actual sort is not being done. Therefore, order-independent transparency cannot be computed efficiently on a Z buffering architecture.

Improved methods for correct transparency have been investigated by Mammen [16], Carpenter [2], Winner et al. [28], Torborg and Kajiya [25], Jouppi and Chang [11], Farias et al. [6, 7], and Lee and Kim [15]. The proposed techniques are either software only [2, 6, 7, 25], require multiple passes of rendering the geometry [28, 16], require use of pointer based linked lists in hardware [15, 25], and/or only render a fixed number of transparent levels correctly [28, 11]. Transparency is a challenging problem to solve in hardware.

Color Plate-Figure 15 on the bottom row shows what happens in OpenGL, when rendering three transparent squares of red, green, and blue. A different image results from each different drawing order, even though the three squares have a fixed Z depth location. On the top row of Color Plate-Figure 15, different drawing order does not impact the visual appearance. This shows the results of order-independent transparency, using the R-buffer architecture.

This approach can be economically implemented in hardware [31]. We have investigated the R-buffer for supporting transparency and antialiasing, and seek here to evaluate it for unstructured volume rendering. Essentially, a frame buffer is used for storing the closest opaque fragment, or the furthest transparent fragment if there are not any opaque fragments. For pixels with additional fragments, those fragments are sent to the buffer along with their X and Y location. Because current graphics accelerators (2001) use custom ASICs with large off-chip memory, the required memory for the R-buffer and extra Z-locations is already available. For spring 2001 consumer graphics cards have 64 MB of on card memory. This memory can be used for texture data, frame buffer, or intermediate off-screen buffers.

We propose to use the memory for the fragments that are not either the closest opaque or furthest transparent.

In successive passes, the fragments are considered, and composited [18] (Porter and Duff) into the frame buffer. Only 1 pass is needed for processing the geometry, so no extra storage is needed for geometry, and a single R-buffer is shared for the entire screen. This amortization of the extra storage over the entire screen allows unique savings over

techniques with large per pixel dedicated storage [2, 11, 15, 28]. This means that, there is one buffer for the entire screen, and it is FIFO (first-in-first-out) accessed. Using storage for the whole screen instead of large per-pixel storage is particularly important for tetrahedral meshes, since irregular grids may have a wide variation in the number of fragments per pixel. With our system, the storage and processing are independent of the screen coverage of any particular transparent fragment. The processing is the same if all of the transparent fragments land on one pixel, across a thousand pixels, or are distributed over the whole screen.

Our R-buffer is flexible and efficient for the calculation of proper transparency without multiple passes of the geometry. Many fragments are culled and eliminated with Z. Experiments show the required memory to support order-independent transparency with highly detailed CAD models to be from 2.1 to 3.6 times higher than a traditional Z buffer ('57 chevy, helicopter, spheres) [31]. Experiments with unstructured datasets show that more memory is required, but the amount is reasonable (Section 5.1). During unstructured volume rendering, texturing does not need to be used, leaving 54 MB available off-chip memory of 64 MB when rendering to 1280x1024 screen resolution. This factor of 5 X times the frame buffer storage in current hardware supports an average depth complexity of 10 fragments per pixel. Because screen resolutions are fixed, and memory densities are growing, using an off-chip memory is economical and will continue to grow at an exponential rate. Large unstructured data sets could be accommodated by increasing the off-chip memory, and paging the R-buffer if it overflows to main memory.

Figure 6 shows how a Z-buffer is augmented with an R-buffer to support order-independent transparency for an arbitrary number of layers. The architecture is a standard graphics pipeline, with geometry processing (G), and rasterization (R). We add an R-buffer, and a 2nd Z storage to the frame buffer.

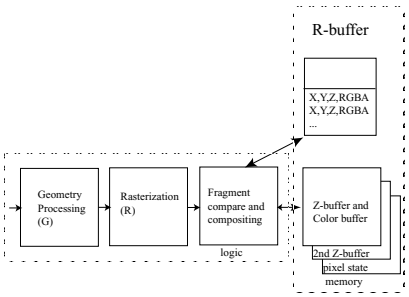

Fig. 6. Graphics architecture

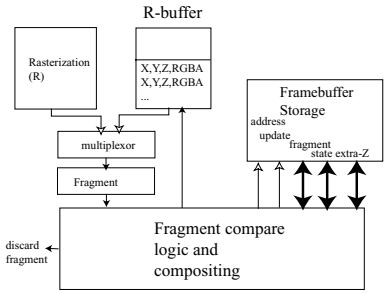

Fig. 7. Interface to R-buffer and frame buffer.

Figure 7 shows more details where a fragment coming from rasterization or from the R-buffer is multiplexed into the R-buffer comparison and control. The R-buffer is considered to be a circular first-in-first out (FIFO) queue. Depending on the fragment's opacity, depth, and the previous state of the frame buffer at that location, a fragment may be stored on the R-buffer, composited into the frame buffer, or discarded. The R-buffer

comparison and control is where the z-buffering comparison typically takes place. The z-buffering is now augmented and revised to provide true transparency. Further details may be found in our paper [31].

3 Traversing Tetrahedral Meshes

The hardware changes presented above already provide a significant data reduction. However, the bandwidth needed can be reduced even further by compressing the data. A typical uncompressed vertex sent to hardware may require as much as 52 bytes of data including geometry coordinates, $RGBA$ values for color, texture coordinates, and a normal vector. For tetrahedral meshes, the cost of transmitting vertices to hardware incurs an overhead as each tetrahedron must be approximated by three or four triangles. This means that rendering tetrahedra with current graphics hardware requires more than three times the bandwidth than for rendering triangles. If the hardware supports a tetrahedral primitive, as we are proposing, then the cost is less, requiring only four vertices per tetrahedron to be sent. Further reduction in bandwidth is achievable by using stripification which we discuss next.

3.1 Triangle Strips and Triangle Fans

The bandwidth problem has been addressed for triangle meshes in various ways. Current graphics APIs address this problem by using vertex caches and multiple-triangle primitives such as triangle strips, triangle fans, and indexed vertex arrays. Here, the idea is to reuse vertices from adjacent triangles. A triangle strip is a sequence of triangles which are linked pairwise by shared edges. A sequential or alternating triangle strip is a triangle strip whose triangles alternate left-right, so that their shared edges form a single line without branches (Figure 8 left). A generalized triangle strip is a triangle strip whose triangles may follow any sequence of left and right turns without constraints (Figure 8 middle).

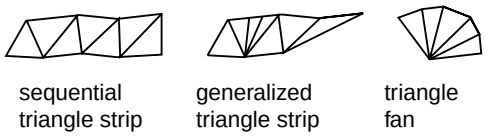

Fig. 8. Triangle strips.

A triangle fan is a special case of a triangle strip in which all the triangles in the sequence share one vertex. The triangles are arranged in clockwise or counterclockwise order around the common vertex (Figure 8 right). Current graphics APIs such as OpenGL provide primitives for both triangle fans and sequential strips that use three vertices to specify the first triangle of a strip and each subsequent vertex to specify a new triangle. Although they do not directly support generalized strips, one may simulate a generalized strip by duplicating a vertex in the vertex stream. Therefore, the cost

of transmitting a mesh to hardware in strip form depends on both the total number of strips and the number of swaps or duplicates necessary to indicate the strip directions.

In practice, systems using strips and fans can achieve transmission rates of 1.2-1.3 vertices per triangle. This can result in bandwidth savings of over 50%. Deering [4] presents a compression scheme that uses a 16-entry vertex cache to provide 0.65 vertices per triangle. Hoppe [9] shows how similar savings may be achieved with indexed vertex arrays and transparent caching, if the model-triangle representation is rendered with knowledge of the cache size. Furthermore it is possible to include strips, fans, and vertex arrays in precompiled display lists for extremely fast processing.

Many of those solutions, however, cannot be applied to tetrahedral meshes. Since the decomposition of tetrahedra into triangles is view-dependent, we cannot use statically computed triangle strips, generalized triangle meshes, or even display lists when changing the viewpoint from frame to frame [30]. One must therefore address the bandwidth problem in a manner specific to tetrahedral meshes. In [30, 26] it has been shown that using a triangle fan for each projected tetrahedron can lead to a speedup in rendering of tetrahedral meshes (see Table 6). Even with this approach, however, rendering N tetrahedra requires 50-60% more bandwidth than rendering N individual triangles.

3.2 Tetrahedral Fans and Tetrahedral Strips

We address this problem by introducing two new primitives for rendering unstructured volume data, the tetrahedral strip and the tetrahedral fan. These primitives are the first methods documented in the literature for reducing the bandwidth required for rendering tetrahedral meshes. We will show that we can achieve transmission rates approaching 1.2 vertex per tetrahedron.

Note that we designed the primitives for use in conjunction with either a full or partial implementation of the hardware-rendering system proposed in Section 2. While such hardware is not yet available, the face and vertex orderings introduced by these strips may also be used to compress unstructured datasets for storage and transmission with only local information needed. Such stripification may improve vertex cache management when rendering tetrahedral meshes on existing graphics cards that support vertex caching and updatable vertex arrays.

To make these primitives useful, we present the algorithms for generating tetrahedral strips from an input tetrahedral mesh. Our algorithms are as simple, efficient, and easy to implement as existing algorithms for triangle strip generation. On tetrahedral meshes, however, our results suggest that tetrahedral simplification compresses unstructured data more than triangle stripification compresses surface meshes. We provide experimental results demonstrating both the strip lengths achieved and their predicted impact on rendering efficiency.

Compared to triangle meshes, it is more difficult to describe the connectivity of tetrahedral meshes. There are several reasons for this:

1) The left-right alternation of sequential triangle strips does not apply to tetrahedra, which lack obvious notions of ‘left’,‘right’, or ‘alternating’.
2) There is no simple way to orient the tetrahedra which are incident to a given vertex.

3) The neighborhood of a single vertex in a tetrahedral mesh may be as complex as an entire triangle mesh. This can easily be seen by considering a triangulation of the sphere, adding a vertex at the center of the sphere, and computing the resulting Delaunay tetrahedralization. A tetrahedral mesh may therefore contain vertices whose neighborhoods are impossible to traverse in a single strip without leaving the neighborhood.

Some of the triangle strip algorithms using a cache [4], [9], achieve transmission rates below 1 vertex per triangle by seeking traversal orderings in which the complete neighborhood of a vertex is visited soon after the vertex enters the cache. In a typical triangle mesh, this situation happens frequently, since a strip often refers to vertices still in the cache. For a tetrahedral mesh however, an average vertex may commonly be incident to 15-20 tetrahedra which are defined by 8-10 other vertices or more. This means that tetrahedral strips need a cache holding 8-10 or more vertices just to hold the neighborhood of a single vertex in cache.

The key insight in dealing with the added complexity of tetrahedral meshes is to consider simplicial complexes in general. It is true that k-dimensional subcomplexes of an n dimensional simplicial complex are equivalent for varying k and n provided that $n-k$ is constant. Hence a vertex of a triangle ($k=0, n=2$) is equivalent to an edge of a tetrahedron ($k=1, n=3$) - the same is true for edges and faces. Just as every edge in a triangle mesh is incident to two triangles, every face in a tetrahedral mesh is incident to two tetrahedra. Just as the triangles incident to a vertex in a triangle mesh form a simple cycle around that vertex, the tetrahedra incident to an edge in a tetrahedral mesh form a simple cycle around that edge. Hence we use edges not vertices as basic element for constructing tetrahedral strips and fans and for describing an oriented traversal of a tetrahedral mesh.

Tetrahedral Fans We define a tetrahedral fan as a sequence of tetrahedra which share a common edge (Figure 9.) Unlike a triangle fan, such a fan does not include the entire neighborhood of any one vertex. Rendering a tetrahedral fan comprising N tetrahedra requires the transmission of only $N+3$ vertices - a substantial savings over the $4N$ vertices needed to render the same N tetrahedra separately.

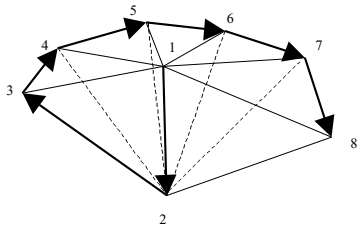

Fig. 9. A tetrahedral fan.

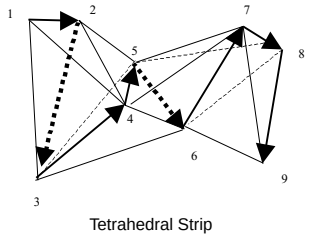

Fig. 10. A sequential tetrahedral strip.

The tetrahedral fan is a fairly simple structure, but its syntax must be defined carefully to make it clear which vertices are being reused. To be consistent with the triangle fan notation, we adopt the convention that the first two vertices being sent make up the edge being shared. The third and fourth vertices complete the first tetrahedron. The fifth vertex defines a second tetrahedron that consists of vertices 1, 2, 4, and 5; the sixth vertex defines a third tetrahedron, with vertices 1, 2, 5, and 6; and so on until the end of the fan is reached.

Tetrahedral Strips We define a tetrahedral strip as a sequence of tetrahedra that are connected by shared faces, but not all of them necessarily share one common edge (Figure 10). An application transmits a tetrahedral strip to hardware by sending the four vertices of the first tetrahedron, and sending a single vertex for each subsequent tetrahedron. We define a sequential strip as a strip in which the new vertex always replaces the least-recently-used (LRU) vertex of the previous tetrahedron. We define a generalized strip as a strip in which there are no constraints on the direction of the next tetrahedron (in other words the vertices may be replaced in any order). For a generalized strip, the application must also send a flag indicating which of the four vertices should be replaced by the following vertex (see Section 4). We assume that a tetrahedral strip is generalized unless otherwise stated.

To use strips, it may be necessary to compute a correctly-ordered list of vertex replacements from a list of the tetrahedra in each strip. One may do so by identifying the common edges shared by sequences of three or more adjacent tetrahedra in the strip. We call the union of a strip's common edges an *edge chain*. For a fan, the edge chain has a single, shared edge, and the vertex ordering may be computed by ordering the remaining vertices clockwise or counterclockwise around the shared edge. For a sequential strip, the edge chain has no branches and no edges shared by more than three consecutive tetrahedra. A sequential strip's edge chain visits all but the first and last vertex of the strip in the correct transmission order. A generalized strip's edge chain, meanwhile, has a branch or shared edge for each replacement that does not follow LRU order.

3.3 Generating Strips and Fans

We have experimented with a variety of tetrahedral strip algorithms. Each algorithm uses an adjacency data structure identifying the four face neighbors of each tetrahedron. We mark each tetrahedron with a flag indicating that it has not been visited, and with an integer that indicates how many unvisited neighbors it has. We store the adjacency information in an array whose ith row contains integers identifying the four neighbors of tetrahedron i. We use the convention that a tetrahedron referred to by the first element in the ith row is the tetrahedron opposite the first vertex of tetrahedron i.

The baseline is a greedy algorithm that selects the first unvisited tetrahedron from the array as the start of the strip, chooses one of its neighbors as the next member of the strip, and repeats until no unvisited neighbors remain. Once all neighbors have been visited, we reverse the strip and seek to extend the strip by starting again at the very first tetrahedron. In the tetrahedral case, this algorithm produces stripifications for

our sample meshes with mean strip sizes of 9-14 tetrahedra each, corresponding to a transmission rate of 1.20-1.33 vertices per tetrahedron. Color Plate-Figure 13 shows an example of a strip generated from the Langley Fighter data set.

In search of better strip quality, we have tried several heuristics to improve the simple greedy algorithm. These heuristics center around the selection of the next unvisited tetrahedron. We experimented with the following criteria:

- Method 1) Greedy: choose the first unvisited neighbor tetrahedron.
- Method 2) choose the neighbor tetrahedron with the fewest unvisited neighbors
- Method 3) choose a sequential order first, if no more tetrahedra can be added switch to 2.)
- Method 4) attempt to create a fan first, then switch to variant 2.)

Method 2 is typically the most effective, since it helps to avoid leaving isolated singleton strips in the mesh. This boosts the average strip length up to 49 tetrahedra. This algorithm can be viewed as the tetrahedral equivalent of the greedy algorithm for triangle strip generation used by Akeley et al. [1] and Evans et al. [5].

3.4 Encoding Stripifications

Compressing 3D representations including triangle meshes and tetrahedral meshes has been a subject of extensive research, motivated by the need to transmit and store large datasets efficiently over the Internet and other networked systems. It may therefore be important to encode the stripification of tetrahedral meshes along with their connectivity. Both Deering [4] and Isenburg [10] provide compressed formats for triangle strips which involve using the triangle strips to guide the encoding. Deering's Java3D compression method encodes a *generalized triangle mesh* which may be computed from a collection of triangle strips, with additional codes to indicate vertices pushed and pulled from the cache. Isenburg encodes triangle strips by marking which edges in a triangle mesh are internal to strips; he observes that it is sufficient to know the internal edges to recover the stripification.

The two algorithms that have been introduced for encoding the connectivity of tetrahedral meshes, Grow and Fold [24], and the cut-border machine [8], allow for freedom in constructing the order of their tetrahedron spanning trees. Either algorithm may therefore be used to encode tetrahedral strips by following the order defined by the trees. Generalizing Isenburg's method, we may then use additional bits to mark the strip-internal faces that indicate where the different strips start and stop.

3.5 Hardware Support for Tetrahedral Strips

The minimum hardware requirements for supporting tetrahedral strips are a buffer that can hold the four vertices of the current tetrahedron and the additional, fifth vertex sometimes created by the Projected Tetrahedra algorithm, and the ability to swap the vertices in the buffer as necessary to follow the direction of the strip. Even greater savings may be achieved by caching information that the Projected Tetrahedra algorithm must compute for each edge and face. Depending on the classification of the current

tetrahedron (see Section 2), the algorithm may need some or all of a tetrahedron's edge lengths and face area. Since each length and area value computed requires a square root, running time and complexity may be reduced by storing these values for the edges and faces that will be reused by the next tetrahedron in a strip.

4 Application Programming Interface (API)

In this section we propose an extension to the OpenGL API to support tetrahedral fans and strips. We add a primitive GL_TET_FAN_EXT. Users would call glBegin(GL_TET_FAN_EXT), and then make glVertex and glColor calls. If we then pass vertices v_0, v_1, v_2, $\ldots, v_6$ for example then the tetrahedra $v_0v_1v_2v_3$, $v_0v_1v_3v_4$, $v_0v_1v_4v_5$, and $v_0v_1v_5v_6$ are rendered. This is a straightforward generalization of a triangle fan. In the case of tetrahedral strips we propose two extensions. The simpler primitive is the GL_SEQUENTIAL_TET_STRIP_EXT. In this case the first 4 vertices specify the first tetrahedron, each following vertex replaces the first vertex of the previous tetrahedron; see Figure 11. The primitive GL_GENERAL_TET_STRIP_EXT requires more programmer intervention but is more flexible. Here we explicitly describe which vertex should be replaced. So we have to introduce flags GL_REPLACE_VERTEX_EXT_1, _2, _3, _4 (see Figure 12). An added OpenGL entry point glReplaceVertexEXT(), would be needed to know which vertex was to be replaced. This allows the most flexibility, however there is an added expense for transmitting 2 bits for the flag. We use the convention that the new vertex is always vertex 4 similar to IrisGL triangle strips.

```
glBegin(GL_SEQUENTIAL_TET_STRIP_EXT);
glVertex(v1);
glVertex(v2);
glVertex(v3);
glVertex(v4);   // draws tet v1-v2-v3-v4
glVertex(v5);   // draws tet v2-v3-v4-v5
glVertex(v6); // draws tet v3-v4-v5-v6
glVertex(v7); // draws tet v4-v5-v6-v7
glVertex(v8); // draws tet v5-v6-v7-v8
glEnd();
```

Fig. 11. Code example for sequential strips

```
glBegin(GL_GENERAL_TET_STRIP_EXT);
glVertex(v1); // stored in slot 1
glVertex(v2);
glVertex(v3);
glVertex(v4);
glReplaceVertexEXT(REPLACE_VERTEX_EXT_1);
glVertex(v5); // goes into slot 1, draws
 // tet v2-v3-v4-v5
glReplaceVertexEXT(REPLACE_VERTEX_EXT_1);
glVertex(v6); // goes into slot 1,
 // drawing tet v3-v4-v5-v6
glReplaceVertexEXT(REPLACE_VERTEX_EXT_3);
glVertex(v7); // goes into slot 3
 // replacing v5, draws v3-v4-v6-v7
glEnd();
```

Fig. 12. Code example for generalized strips

5 Results

We have evaluated the architecture by simulating the R-buffer hardware [31], implementing the stripification algorithms in MatLab, and experimenting with optimization of CellFast [30]. A complete hardware and software system implementation is a large project. Whether such features are commercially viable is not yet clear, but the support of new functionality to support unstructured rendering in desktop graphics accelerators is practical and efficient with our proposed architecture. We present results on the

R-buffer processing of unstructured data, on the compression, and then a "back of the envelope" performance evaluation based on bottleneck analysis. The graphics accelerators cannot exceed the speed by which data can be sent to them. We believe the performance estimates to represent the possible gains, 72-86 speedup for unstructured rendering, and are currently working on implementation issues.

5.1 R-buffer Results

We ran our R-buffer architectural simulator on seven datsets ranging in size from 12 thousand to 2.4 million tetrahedra. The data sets shown in Figure 13 have sizes as provided in Table 6. We used a trace of fragments generated by our CellFast implementation [30] on each data set as input to a behavioral simulator. Table 1 shows the results of processing these datasets, and Figure Color Plate 13 top row shows the rendering of CellFast in OpenGL. Figure Color Plate 13 bottom row shows the actual output of the architectural simulator. The number of recirculating fragments passes ranges from 52 to 552, and average depth complexities of covered pixels range from 9.98 to 201.55 fragments per pixel. All images were rendered to a 512x512 framebuffer.

Our simulator actually produces better results than the several OpenGL implementation tested, because of the use of unassociated color storage with associated compositing. The difference can be seen on the phoenix dataset, where there are greatly reduced triangle artifacts at the edges of the model. The ratio of Z bandwidth, meaning the amount of bandwidth used to simply read-modify-write, to the R-buffer bandwidth is provided, and ranges from 13.7 to 135.1. The statistics depend upon the viewpoint. The number of passes is related to the worst case depth complexity of any pixel. The torso and head datasets were culled by a semiautomatic classification algorithm, Table 6, and also rendered with no culling. The memory ratios for the proposed architecture range from 1.8 to 48.7 for these datasets. Because the torso full and head full datasets result in such high average depth complexity, 154.5 and 201.6, users would likely use simplification, hierarchical models, or culling to regions of interest. Our results show the memory remains reasonable $O(Nd)$, but the run time complexity does depend upon the average depth complexity squared, $O(Nd^2)$. Here N is the number of pixels, and d is the average depth complexity over those pixels. For small average depth complexities, the sorting is reasonable as shown by the tot/Z bandwidth provided in Table 1 for the first five datasets. Front-to-back compositing with adaptive opacity threshold termination would also be an effective way to limit the amount of layers to be sorted.

5.2 Stripification Results

We ran our stripification algorithms on five test data sets, ranging in size from 3,000 tetrahedra to 240,000 tetrahedra. The datasets 'bracket' and 'arm' are small data sets containing 3398 and 3157 tetrahedra. Using Method 2 above, we achieve the best results with mean strip lengths of 29 tetrahedra per strip and median lengths over 10 tetrahedra per strip. Method 2 is followed by Method 4 (mean 15.4), then by Method 1 (mean 11.4), and last is Method 3 (mean 10.9). Results are presented in Tables 2, 3, 4, and 5. The columns are defined as: *nstrips* - number of tetrahedral strips; *tmean* - mean number of tetrahedra in each strip; *tmed* - median number of tetrahedra in each strip;

data set	frags	depth comp.	passes	tot/Z	mem	tot/Z
	1000's	pixels		band	MB	mem
phoenix	1022	27.6	66	17.4	14.9	6.6
langley	624	31.5	83	22.5	10.4	4.6
f117	75	10.0	83	14.9	4.1	1.8
torso	214	48.8	255	32.5	5.7	2.5
head	105	19.2	52	13.7	4.5	2.0
torso full	5416	154.5	552	120.5	65.2	29.0
head full	9287	201.6	475	135.1	109.5	48.7

Table 1. R-buffer statistics.

data set	nstrips	tmean	tmed	tstd	tmax	vpert
phoenix	1000	12.9	4	20.5	182	1.23
langley	4745	14.8	3	29.1	370	1.20
f117	22203	10.8	6	17.7	404	1.28
bracket	367	9.3	2	27.1	440	1.32
arm	349	9.1	3	18.5	301	1.33

Table 2. Results for Method 1.

tstd - standard deviation of the number of tetrahedra in each strip; *tmax* - the maximum number of tetrahedra in a strip; *vpert* - the number of vertices per tetrahedra.

data set	nstrips	tmean	tmed	tstd	tmax	vpert
phoenix	441	29.4	10	46.4	347	1.10
langley	1432	49.0	13	86.8	865	1.06
f117	6504	36.9	10	80.5	1472	1.08
bracket	222	15.4	5	23.7	119	1.20
arm	225	14.1	5	26.9	232	1.21

Table 3. Results for Method 2.

data set	nstrips	tmean	tmed	tstd	tmax	vpert
phoenix	1231	10.5	3	37.2	962	1.29
langley	4907	14.3	4	66.7	3486	1.21
f117	20578	11.7	3	71.6	6238	1.26
bracket	398	8.6	3	18.4	188	1.35
arm	340	9.3	3	24.4	301	1.32

Table 4. Results for Method 3.

data set	nstrips	tmean	tmed	tstd	tmax	vpert
phoenix	882	14.7	4	32.4	293	1.20
langley	3763	18.6	5	44.6	676	1.16
f117	10030	23.9	6	62.9	2229	1.13
bracket	349	9.8	3	17.0	148	1.31
arm	313	10.1	4	17.8	159	1.30

Table 5. Results for Method 4.

data set	culled	tets	f/sec HW	f/sec	Speedup
phoenix	0%	12,936	2441	33.6	72.6
langley	20%	70,125	592	7.2	82.2
f117	36%	240,122	212	2.47	85.7
torso	81%	1,293,238	120	1.53	78.3
head	99.6%	2,443,013	2724	11	247.7

Table 6. Projected and measured performance.

Since the purpose of our algorithm is to improve rendering performance, our success cannot be completely measured by the length of the strips we produce.

To take full advantage of volume rendering with tetrahedral strips requires the new architecture we have proposed. But, using middleware much of the advantages may be retained by storing datasets on disk and in memory in strips. We have experimented with schemes that check and update strips on the fly to ensure the visibility ordering remains correct.

Some software algorithms for tetrahedron visibility sorting, such as the MVPO [27] and BSP-XMPVO and ZSweep [23, 6] algorithms, generate adjacency information for the sorted tetrahedra. These representations could be used directly to group tetrahedra for transmission to hardware as relatively short strips.

5.3 Performance Estimates

We present here performance estimates for our architecture. The system supports tetrahedral rasterization, R-buffer sorting, and tetrahedral stripping as just described (See Figure 3). The system bus bottlenecks determine peak performance. Bottleneck analysis is a key tool for system architects and designers, as it allows accurate speed limits to be defined. For example, H.P.'s Visualize fx graphics hardware has targeted the sustained throughput of AGP, AGP2X, and AGP4x in order to match the capabilities of the platform. Other factors come into play, such as fill rate when more sophisticated features such as multitexturing are used. Therefore an application may often be slower than the bottleneck, but it shall never exceed the predetermined system bottlenecks. An optimal system may go no faster than the peak transfer rate of the graphics bus, currently AGP (accelerated graphics port) 4X. AGP 4X has a peak rate of approximately 1000MB/sec. Each tetrahedra that would be sent to the graphics hardware would have vertex location data v_x, v_y, and v_z, as well as color data r, g, b, a. If we take all 7 values to be floats to avoid any difficulties with opacity precision for classification, then we have 7 floats/vertex. (Our simulator uses the 8 bit components for RGBA from Mesa.) The following equation determines how many bytes per tetrahedra there are:

$$(\frac{7 \text{ floats}}{\text{vertex}}) \times (\frac{4 \text{ B}}{\text{float}})(\frac{4\text{vertices}}{\text{tet}}) = 112\frac{\text{B}}{\text{tet}} \tag{1}$$

Now using the AGP 4X rate, we can compute the expected rendering rate which AGP cannot exceed:

$$(\frac{1000\text{MB}}{\text{second}}) \times (\frac{1\text{tet}}{112\text{B}}) = 8.9\frac{\text{Mtet}}{\text{second}} \tag{2}$$

The tetrahedra/second rate of an optimized projected tetrahedra implementation is 400,000 tet/second (See Table 6). A rate of 8.9 Mtet/second is a 20X improvement.

Note that this performance improvement of 20X is based on just rendering tetrahedra from a display list or retained mode. The sorting on the host, as well, as the floating point processing on the host would be removed, instead one would pass the display list to the hardware. Adding stripification, the improvement would be even greater, as the amount of bandwidth consumed per tetrahedra drops dramatically. Assuming the average rate of 17 tets/strip averaging the average stripification over all datasets shown, the amount of data is 20/17 vertices/tet or 1.18 vertices/tet. If we assume a begin/end overhead of 24 bytes for each strip on AGP from handshaking, signaling, and OpenGL commands to the hardware, we get the following:

$$(\frac{28\text{B}}{\text{vertex}}\frac{20\text{verts}}{\text{strip}} + \frac{24\text{B overh}}{\text{strip}}) \times (\frac{1\text{strip}}{17\text{tets}}) = 34.35\frac{\text{B}}{\text{tet}} \tag{3}$$

or using the AGP 4X rate,

$$(\frac{1000\text{MB}}{\text{second}}) \times (\frac{1\text{tet}}{34.35\text{B}}) = 29\frac{\text{Mtet}}{\text{second}} \tag{4}$$

Computing with this rate, provides 34.35 Bytes/tet, which improves performance by an additional 3.3X. The expected tetrahedral rate is 29 million tet/second. This formidable rate makes even unculled datasets truly interactive. For the expected performance, the frame rates can go up by this 72X to 86X improvement, for example as shown by 5 available datasets in Table 6. Table 6 provides actual CellFast measurements on an fx10 HP PC for the 5 datasets, as well as projected performance with the new architecture, and the calculated speedup of the new architecture over CellFast. The frame/sec HW is the frame rate with the proposed hardware architecture and stripping, while the f/sec is with CellFast implementation using hardware triangle rasterization. Note that the number of tetrahedra that are rendered is the number of tetrahedra in the data set minus the number of tetrahedra culled. The head data set is so large, that it causes cache problems, even when running on the 99.6% culled version, which provides a slower run time than for data sets that are smaller. This exaggerates the expected speedup calculated to 247X. While our claim of a potential for 29 million tetrahedra/second may seem high, current graphics cards achieve millions of triangles/second, and polygon improvement rates have been beating Moore's Law. Since our architecture is designed to leverage consumer graphics hardware development, we believe such estimates to be a reasonable goal.

6 Conclusion and Future Work

We have shown that addition of tetrahedral primitives moves much of the work from the host CPUs to the graphics hardware. While it is no surprise that this can improve performance, it is surprising that the possibility exists to improve performance by over 70X. The advantages of our architecture are to unburden the host from performing the sorting or splitting. The sorting per fragment is exact, and also very fast in hardware, as each pixel now requires a reduced amount of sorting versus the per primitive sort required without our architecture. By adding tetrahedral primitives, data are more compact than being split into triangles. An additional 3.3X factor resulted from tetrahedral stripification. While nearly a direct analog to triangle stripification, the gains are greater than expected. Our results show that tetrahedral stripping may be inherently more efficient than triangle stripping. Future work is to perform stripification on the culled datasets to get the greatest advantage. A representation that allows dynamic classification, and generation of triangle strips would be ideally suited for our architecture. Several heuristics were demonstrated and evaluated, and results with an average strip length of up to 50 tetrahedra were achieved.

A drawback of our approach is that since it is based on Shirley and Tuchman's projected tetrahedra algorithm, it suffers some of the inaccuracies that algorithm may produce. But, the linear approximation is the easiest to place into hardware. The linear interpolation involved in the projected tetrahedra algorithm, for example, can produce mach banding. Rottger et al. [19] propose using texture mapping as a method to produce higher-order interpolation effects for projected tetrahedra. Their method would be interesting to integrate with ours.

Another disadvantage is that although our system can be used to render a variety of complex, transparent models, none of the current algorithms for rendering of tetrahedral

meshes may be easily adapted to support refraction. Refraction would require sampling the irregular grid with refraction rays at a variety of directions which could not be sampled accurately without resorting the tetrahedral mesh from a different viewpoint.

Finally, the tetrahedral API support is simple, an analog of triangle and triangle strip primitives. The amount of changes in hardware to support the tetrahedral primitives is small, especially for those graphics architectures with microcoded geometry acceleration. What we have presented is a novel and powerful approach to make unstructured volume rendering truly interactive.

ACKNOWLEDGEMENTS

We thank our reviewers. We thank for the NASA Langley Fighter, Neely and Batina; for the Super Phoenix Nuclear reactor, Bruno Nitrosso, Electricite de France; for the F117, Robert Haimes, MIT; for the bracket and arm data sets, Jeffrey Berkley, University of Washington HIT Laboratory; and for the head and torso datasets, the Center for Scientific Computing and Imaging, University of Utah.

References

1. K. Akeley, P. Haeberli, and D. Burns. tomesh.c. C Program on SGI Developer's Toolbox CD, 1990.
2. L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, pages 103–108. ACM, July 1984. Vol. 18, No. 3.
3. P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In *Proceedings of the Eurographics Workshop, Visualization in Scientific Computing'95*, pages 59–71, Chia, Italy, May 1995.
4. M. Deering. Geometry compression. In *Proceedings of SIGGRAPH 95*, pages 13–20, Aug. 1995.
5. F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization*, pages 319–326, San Francisco, CA, Oct. 1996. IEEE.
6. R. Farias, J. S. B. Mitchel, and C. T. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *ACM/IEEE Volume Visualization and Graphics Symposium 2000*, page in press, Oct. 2000.
7. R. Farias and C. Silva. Parallelizing the zsweep algorithm for distributed-shared memory architectures (ST). In *to appear International Workshop on Volume Graphics*, Long Island, NY, June 2001.
8. S. Gumhold, S. Guthe, and W. Strasser. Tetrahedral mesh compression with the cut-border machine. In *Proceedings IEEE Visualization '99*, pages 51–58. IEEE Computer Society Press, 1999. Mesh Compression Techniques.
9. H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH*, pages 269–276, Los Angeles, CA, Aug. 1999. ACM.
10. M. Isenburg. Triangle strip compression. In *Proceedings of Graphics Interface*, pages 197–204, 2000.
11. N. P. Jouppi and C.-F. Chang. Z3: An economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of Graphics Hardware*, pages 85–93, Los Angeles, CA, Aug. 1999. ACM/Eurographics.
12. M. Karasick, D. Lieber, L. Nackman, and V. Rajan. Visualization of three-dimensional delaunay meshes. *Algorithmica*, 19(1-2):114–128, Sept.-Oct. 1997.

13. A. Krech. Blitzen: Lightning speed 3D geometry accelerator. Presentation slides in Hot Chips 10, Aug 1998.
14. A. Lastra, S. Molnar, M. Olano, and Y. Wang. Real-time programmable shading. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 59–66, Monterey, CA, Apr. 1995. ACM.
15. J.-A. Lee and L.-S. Kim. Single-pass full-screen hardware accelerated antialiasing. In *Proceedings of Graphics Hardware*, pages 67–75, Interlaken, Switzerland, Aug. 2000. ACM/Eurographics.
16. A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, July 1989.
17. N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *ACM Computer Graphics (Proceedings of the 1990 Workshop on Volume Visualization)*, 24(5):27–33, 1990.
18. T. Porter and T. Duff. Compositing digital images. In *Computer Graphics*, pages 253–259, Aug. 1984.
19. S. Rottger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the Visualization*, pages 109–116, Oct. 2000.
20. P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *1990 Workshop on Volume Visualization*, pages 63–70, San Diego, CA, Dec. 1990.
21. C. T. Silva and J. S. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):142–157, 1997.
22. C. T. Silva, J. S. Mitchell, and A. E. Kaufman. Fast rendering of irregular grids. In *ACM/IEEE Symposium on Volume Visualization*, pages 15–22, San Francisco, CA, October 1996.
23. C. T. Silva, J. S. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *ACM/IEEE Symposium on Volume Visualization*, pages 87–94, Research Triangle Park, NC, October 1998.
24. A. Szymczak and J. Rossignac. Compressing the connectivity of tetrahedral meshes. *Computer-Aided Design*, 32(8/9):527–538, Jul./Aug. 2000.
25. J. Torborg and J. T. Kajiya. Talisman: Commodity realtime 3D graphics for the PC. In *Proceedings of SIGGRAPH*, pages 353–363, New Orleans, LA, Aug. 1996. ACM.
26. P. L. Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
27. P. L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.
28. S. Winner, M. Kelley, B. Pease, and A. Yen. Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. In *Proceedings of SIGGRAPH*, pages 307–316, Los Angeles, CA, Aug. 1997. ACM.
29. C. M. Wittenbrink. Irregular grid volume rendering with composition networks. In *Proceedings of IS&T/SPIE Visual Data Exploration and Analysis V*, volume 3298, pages 250–260, San Jose, CA, Jan. 1998. SPIE. Available as Hewlett-Packard Laboratories Technical Report, HPL-97-51-R1.
30. C. M. Wittenbrink. CellFast: Interactive unstructured volume rendering. Technical Report HPL-1999-81(R1), Hewlett-Packard Laboratories, July 1999. Appeared in 1999 IEEE Visualization-Late Breaking Hot Topics.
31. C. M. Wittenbrink. R-buffer: A pointerless A-buffer hardware architecture. In *In press Proceedings of Graphics Hardware*, Los Angeles, CA, Aug. 2001. ACM/Eurographics. Also available as Technical Report, HPL-2001-12, HP Confidential, Jan. 2001.

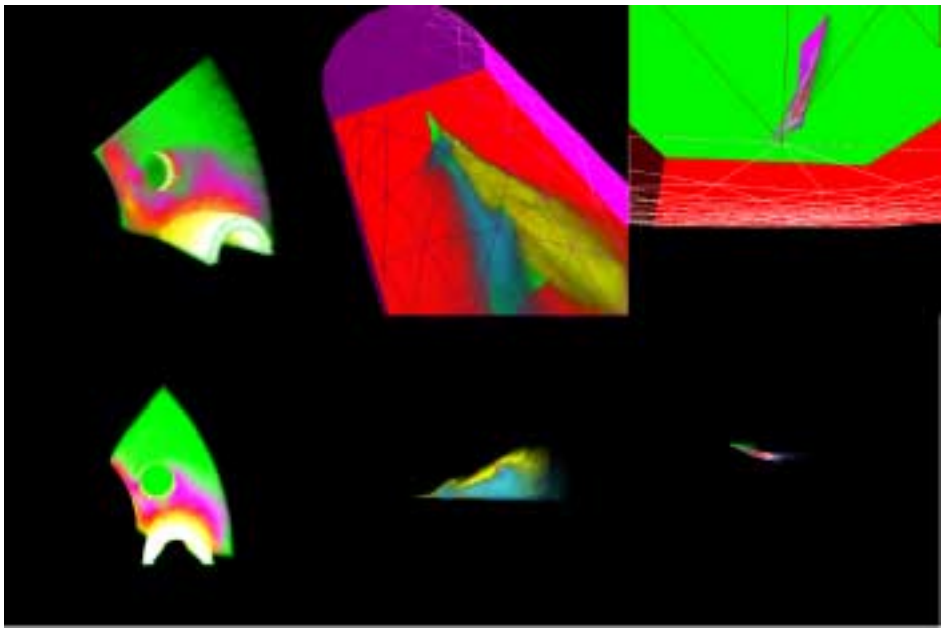

Fig. 13. Color Plate: Unstructured volume rendering of the example data sets Phoenix (left), NASA Langley Fighter (middle), and F117 (right). The top row shows CellFast output to HP OpenGL, and the bottom row shows the proposed architectural simulator's output.

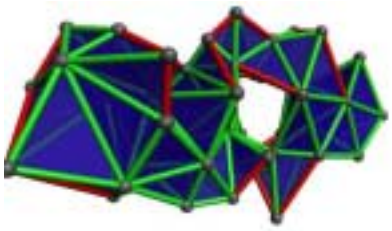

Fig. 14. Color Plate: Tetrahedral strip from the Langley data set

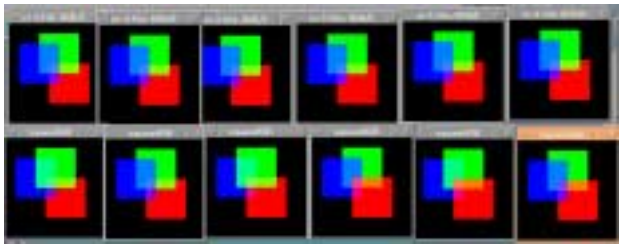

Fig. 15. Color Plate: Top row, R-buffer, same appearance. From near to far, the squares are ordered Blue, Green, Red. Bottom row, conventional Z-buffer, different every time.