# Progressive Volume Rendering of Large Unstructured Grids

Steven P. Callahan, Louis Bavoil, Valerio Pascucci, *Member, IEEE*, and Cláudio T. Silva, *Member, IEEE*

**Abstract**—
We describe a new progressive technique that allows real-time rendering of extremely large tetrahedral meshes. Our approach uses a client-server architecture to incrementally stream portions of the mesh from a server to a client which refines the quality of the approximate rendering until it converges to a full quality rendering. The results of previous steps are re-used in each subsequent refinement, thus leading to an efficient rendering. Our novel approach keeps very little geometry on the client and works by refining a set of rendered images at each step. Our interactive representation of the dataset is efficient, light-weight, and high quality. We present a framework for the exploration of large datasets stored on a remote server with a thin client that is capable of rendering and managing full quality volume visualizations.

**Index Terms**—Volume Rendering, Large Unstructured Grids, Client-Server, Progressive Rendering, Level-of-Detail

## 1 Introduction

Unstructured grids (*e.g.*, tetrahedral meshes) are common in simulation domains such as computational fluid dynamics and structural mechanics. Because of the constantly improving algorithms and computational power, the scale of these simulations has continued to grow at a rate faster than visualization techniques can explore them. Specialized graphics clusters have been developed to visualize large datasets in parallel and on large displays. However, the availability of these clusters is often limited. More recently, many techniques have been developed to visualize volumetric data on commodity PCs using graphics hardware [6, 23]. This provides a solution that allows researchers to perform their visualizations locally when other resources are unavailable. However, due to the limitations of storage and memory with most desktop machines or laptops, this solution does not scale well for extremely large datasets.

As an example, consider a scientist working remotely who would like to visualize a large dataset on his laptop computer. A reduced representation of the data (*e.g.*, simplification [12]) may not be appropriate if a high quality visualization is required for analysis. Complicating matters even further, the laptop may not have capacity on the hard disk or in memory to keep the dataset. The problem is compounded if you consider that the scientist may only want to browse through a series of datasets quickly, requiring the download of each dataset before visualization.

In this paper, we present a client-server architecture for hardware-assisted, progressive volume rendering. The main idea is to create an effect similar to progressive image transmission over the internet. A server acts as a data repository and a client (*i.e.*, a laptop with programmable graphics hardware) acts as a renderer that accumulates the incoming geometry and displays it in a progressively improving manner (see Figure 1). Our progressive strategy is unique because it only requires the storage of a few images on the client for the incremental refinement. For interactivity, a small portion of the mesh is stored on the client using a bounded amount of memory. Furthermore, the progressive representation provides a natural means for level-of-detail exploration of very large datasets without an explicit simplification step that may be difficult and costly. Because the geometry is rendered in steps, the user can stop a progression and change the view without penalty, thus facilitating exploration. Our algorithm is robust, memory efficient, and provides the ability to create and manage approximate and full quality volume renderings of unstructured grids too large to render interactively at full resolution.

Our contributions include:

- We introduce a client-server architecture and interface for rendering large datasets and managing the resulting visualizations;
- We detail a server that acts as a data repository by streaming a tetrahedral mesh in partial visibility order to one or more clients;
- We describe a client that uses hardware-assisted, progressive volume rendering to provide an interactive approximation, progressive refinement, and full-quality rendering of large datasets;
- We provide experimental results of our algorithm and include a discussion on the benefits and limitations of our approach.

The rest of the paper is outlined as follows. Section 2 provides an overview of related research. Section 3 describes an overview of our client-server architecture. More detail is provided about the server in Section 4 and about the client in Section 5. In Section 6, we outline our experimental results, in Section 7 we include a discussion of the trade-offs of our algorithm, and in Section 8, we provide brief discussion of our algorithm and our conclusions.

## 2 Previous Work

Hardware-Assisted Volume Rendering. Volume rendering on a commodity PC has been the subject of much research recently, due to the steady increase in processing power on graphics processing units (GPUs) and the advent of programmable shaders. Here we summarize the state-of-the-art for hardware-assisted volume rendering of unstructured grids; for more detail we refer the reader to a recent survey [22]. For unstructured grids, volume rendering algorithms generally fall into two categories: ray-casting and splatting. Recently, Weiler *et al.* [23] proposed an algorithm to perform ray-casting completely on the GPU by storing the mesh and traversal structure in GPU memory. This algorithm was made more efficient and extended to handle non-convex meshes in subsequent work by Weiler *et al.* [24] and Bernardon *et al.* [2]. These algorithms benefit from low latency because they avoid CPU to GPU data transfers. However, the limited memory of GPUs prevents these algorithms from rendering even moderately sized datasets. Pioneering work on tetrahedral splatting by Shirley and Tuchman [21] introduced the Projected Tetrahedra (PT) algorithm. For each viewpoint, PT decomposes a tetrahedron into one to four triangles that can be rendered efficiently in hardware. Unfortunately, compositing of the triangles requires an explicit visibility ordering that is implicit to ray-casting. Many algorithms have been proposed to perform the visibility ordering in object-space (*e.g.*, [26]).

- *Steven P. Callahan, Louis Bavoil, and Cláudio T. Silva are with the Scientific Computing and Imaging Institute at the University of Utah, E-mail: {stevec,bavoil,csilva}@sci.utah.edu.*
- *Valerio Pascucci is with Lawrence Livermore National Laboratory, E-mail: pascucci@acm.org.*

*Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.*

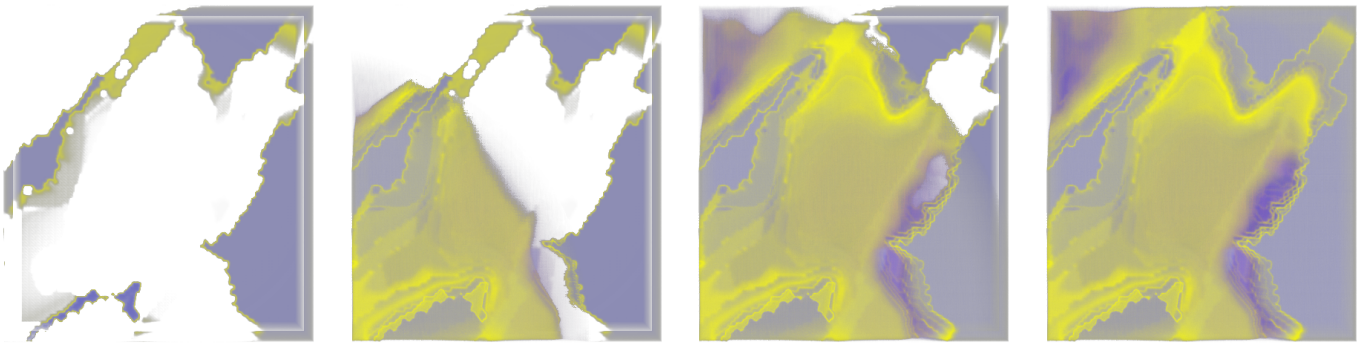

Fig. 1. A sequence of progressive volume rendering steps for the SF1 dataset with about 14 million tetrahedra. Starting from an interactive mode that uses only the boundary (left), the algorithm progressively refines the image using incoming geometry as well as the results of the previous refinement until the full-quality rendering is achieved (right).

More recent work by Callahan *et al.* [6] introduced the Hardware-Assisted Visibility Sorting (HAVS) algorithm which sorts in object-space and image-space. The HAVS algorithm is fast, efficient, and flexible enough to handle dynamically changing geometry [1]. In this paper, we exploit these benefits to provide a client renderer that handles streaming geometry in a progressive form. A more detailed overview of the algorithm is provided in Section 5.

Level-of-Detail. Hardware-accelerated volume rendering algorithms allow the interactive rendering of moderately sized datasets. However, multiresolution or level-of-detail (LOD) techniques [17] are still essential to render large datasets interactively. Unlike structured grids, the area is relatively new and not well studied for unstructured grids. For this reason, initial algorithms applied techniques for structured grids on unstructured grids by sampling them uniformly [15]. A drawback with this type of approach is that the resulting resampled meshes are larger than the original by at least an order of magnitude. A more efficient algorithm proposed by Museth *et al.* [19] uses a hierarchy of splats to render culled, opaque geometry that can be explored with the use of CSG cuts. For direct volume rendering, Cignoni *et al.* [8] present a hierarchy of simplification steps that can be traversed to a desired LOD for interactive visualization. In general, these algorithms are not suitable for streaming in a client-server architecture due to a combination of increased data size, inability to display a full-quality volume rendering, or the difficulty of traversing the levels-of-detail dynamically. More recently, Callahan *et al.* [5] introduced sample-based simplification for dynamic LOD on unstructured grids. The algorithm samples the geometry by importance and determines the number of elements to render at each pass based on the frame-rate of the previous pass. Our algorithm uses a similar idea to create an approximate volume rendering from the portion of the geometry that has arrived from the server.

Client-Server Visualization. The problem of remotely visualizing large datasets has been the subject of research for many years. The most widely recognized solutions perform the visualization task on large clusters using software algorithms [20] or with hardware-assisted algorithms [18] through the use of specialized graphics hardware [13]. Typically, an image is created using the cluster, then compressed for transmission to the client, where it is decompressed and displayed to the user [10]. Systems such as Vizserver[1] are available from vendors for performing client-server visualization in this manner. The Visapult system [3] was developed to push more of the burden onto the client. This is done by rendering blocks of the data from the server in a distributed system and compositing the results on the client. A less restrictive class of algorithms performs the visualization on more limited resources by assuming a simple server (*e.g.*, a web server). To this end, Lippert *et al.* [16] introduced a system in which the server stores compressed wavelet splats that are transmitted to the client for rendering. As the splats are received by the client, the image is progressively refined. Another approach by Engel *et al.* [9] describes a progressive isosurface visualization algorithm for use on the web. This is done by allowing the server to compute a hierarchy of isosurfaces that are transmitted to the client progressively. For efficiency, only the difference between two successive levels of the hierarchy is sent across the network. More recently, the client-server architecture provided by Kaehler *et al.* [14] performs visualization of Adaptive Mesh Refinement (AMR) data that is stored remotely on a server and adaptively rendered locally on a client by interpolating the hierarchical structure of the grids. Our algorithm is similar to this latter class of algorithms, but for the more difficult case of unstructured grids. We assume a limited server that prepares the geometry and streams it to the client in a series of progressive steps that avoid redundant transmission and unnecessary storage. This allows the client to receive the non-overlapping geometry and refine the image with assistance of the GPU.

[1] http://www.sgi.com/products/software/vizserver

## 3 System Overview

Our client-server system architecture is depicted in Figure 2. The server acts as a data repository and geometry processor. The data is stored on the server hierarchically for efficient traversal and iterative object-space sorting. The client keeps the boundary triangles for an *Interactive Mode* and requests geometry from the server in a *Progressive Mode*. In the *Progressive Mode*, the client uses hardware-assisted LOD volume rendering to refine the image using the results of the previous progressive step. Upon completion of the progressive volume rendering, the client saves a copy of the image for later browsing in a *Completed Mode*. Viewing changes in any of the client steps causes the progressive renderer to stop and return to the *Interactive Mode*.

## 4 The Server

### 4.1 Geometry Processing

Overview. The server acts as a data repository that sorts and streams triangles. However, sorting large datasets in one pass may be cumbersome. For the client to remain interactive, it should begin receiving nearly-sorted faces immediately from the server. Furthermore, the client should be able to interrupt the streaming of the faces at any time to keep the exploration interactive.

To keep the processing of the datasets to a minimum, we perform a one-time preprocessing of the datasets to extract the unique triangle faces and vertices and store them in a binary format that we read when starting the server. Then, we use the server to process the faces of the mesh into an octree structure that can be traversed by depth ranges, from front to back. For every packet of faces requested from the client, the server first culls faces outside the current depth range, sorts the remaining faces, and sends them to the client. Subsequent packets use incremented depth ranges. This has the effect of distributing the sorting burden between each step of the progression. It also prevents unused geometry from being sorted.

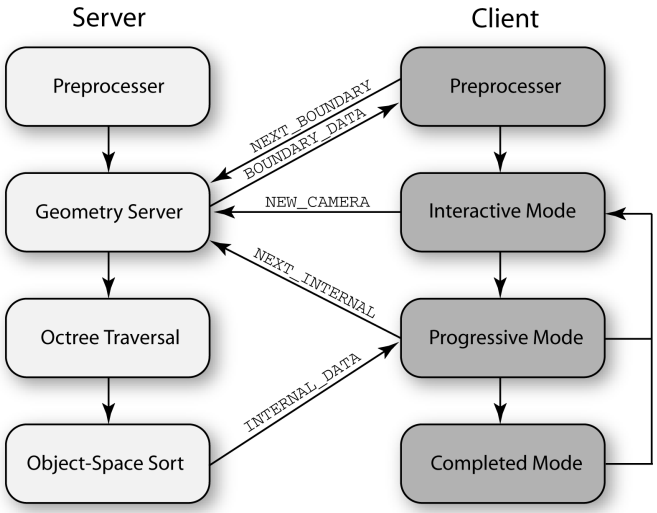


Fig. 2. The client-server architecture. Communication between the client and the server is shown with annotated arrows.

Depth-range culling. For a given depth range, we cull the associated geometry using a depth-range octree. Our octree is a geometric partition of the faces, according to their centroids. Our depth-range octree is similar to the octree from [25] for isosurface extraction. However, instead of using scalar values, our octree uses dynamically changing depth ranges to cull the geometry outside the current depth range. Each octree node contains an array of face indices. The depth range of a node is the minimum and maximum distance from the eye to the bounding box corners of the node. To find the faces matching a given depth range, the octree nodes are culled hierarchically by traversing nodes that may contain triangles in the given depth range. Next, the collection of triangles in the matching octree nodes are culled according to their distance from the viewpoint. The remaining triangles are then inserted into an array for sorting. We use a radix sort on the face centroids, as in [6]. The triangles are then sent as a vertex array for direct rendering on the client.

To increment the depth range on each pass, we uniformly divide the range of the minimum and maximum distance from the eye to the bounding box of the mesh. This has the unfortunate side effect that the number of triangles per slice can vary in size. We address this issue by collecting packets on the server and only transmitting them to the client when a user-specified target triangle count is reached. This results in good performance since the sorting and network transfers are more efficient for larger packets.

View-frustum culling. When interactively exploring regions of the mesh in detail, often many of the faces are outside of the view-frustum. These faces should not be transmitted to the client. Therefore, in addition to the depth range test, we also perform view-frustum culling for each node of the octree. The left, right, bottom and top planes of the frustum are computed from the modelview-projection matrix sent by the client. This results in significant performance improvements for zoomed-in views of the dataset (see Figure 5).

Compression. Network transfers become a bottleneck for client-server systems with high bandwidth. To reduce the bandwidth, we compress the transmitted vertex arrays using the open-source zlib library[2] based on Huffman codes, which is fast and robust. We use the maximum level of compression for our client-server data transfers.

### 4.2 Network Protocol

The following description assumes a single client per server, and can be extended for multiple clients by storing the context of the stream on each client.

[2]http://www.zlib.net

The server understands three types of commands: NEW_CAMERA, NEXT_BOUNDARY, and NEXT_INTERNAL.

New Camera. Each new client initializes the stream by sending a NEW_CAMERA command, containing a frame ID and the camera information. The frame ID is an integer initialized to 0 and incremented by the client for every NEW_CAMERA command.

Next Boundary. Upon connection, the client requests boundary faces from the server using the NEXT_BOUNDARY command. The server incrementally sends the boundary faces in a BOUNDARY_DATA packet consisting of an unsorted triangle soup (*i.e.*, a sequence of vertex coordinates) for the client to use during interactive rendering. By sending the boundary in chunks, the client can request only the portion that it can retain in memory. If the client is limited and can only use a portion of the boundary during interaction, the NEXT_BOUNDARY command can be used at the beginning of the progressive rendering to fill in the missing boundaries before the internal faces are transmitted.

Next Interior. To avoid unnecessary sorting when the camera is moving, the server does not sort the faces until it receives a NEXT_INTERNAL command from the client. The server culls the geometry by depth and frustum, sorts by centroid, and sends back a chunk of the interior faces in an INTERNAL_DATA packet.

Synchronization issues. We use a TCP/IP socket to transmit our data. Unlike UDP, TCP guarantees that the data packets sent from the server arrive in the same order they were sent and without error. Since the client can change the camera at any time, the state of the client and the server need to be synchronized. This is necessary to avoid issues when the camera moves and the client is receiving data asynchronously, or the server is processing data. All the packets from the client or from the server contain a frame ID. Before processing a packet, the client and the server check that the frame ID from the packet is the same as the current frame ID. Packets with obsolete frame IDs are ignored by the server and the client. In addition, when the client receives an obsolete packet, it resets the stream.

## 5 The Client

The main function of the client is as a progressive volume renderer. Because the client may be limited in disk space as well as memory (*e.g.*, a laptop), the goal is to minimize the data stored on the machine at each progressive step. Therefore, the client acts as a *stream renderer*— it receives geometry transmitted from the server and renders it directly with the GPU. This requires a volume renderer capable of handling dynamic data in an efficient manner. In addition, our algorithm requires an approximation technique for partial geometry as well as a means of keeping previously computed information for subsequent refinement steps.

To leverage GPU efficiency, we extend the Hardware-Assisted Visibility Sorting (HAVS)[3] algorithm of Callahan *et al.* [6] to perform progressive volume rendering. The HAVS algorithm operates in both object-space and image-space to sort and composite the triangles that compose a tetrahedral mesh. In object-space, the triangles are sorted by their centroids using a linear-time radix sort for floating-point numbers. This provides a partial order for the triangles. Upon rasterization, the fragments are sorted again in image space using a fixed size A-buffer [7] implemented with programmable shaders called the *k*-buffer. For each pixel of the resulting image, $k$ entries (scalar value $v$ and depth $d$) are stored in textures on the GPU. An incoming fragment is compared to entries in the $k$-buffer to find the two closest to the current viewpoint (for front-to-back compositing). These entries are then used to look up the color and opacity for the volume gap in a pre-integrated table by using the front scalar, back scalar, and distance between the entries. This color and opacity are then composited to the framebuffer, the front entry is discarded, and the remaining entries are written back into the $k$-buffer. For more detail on the HAVS algorithm, see [6]. Our progressive renderer uses the HAVS algorithm as a basis

[3]http://havs.sourceforge.net

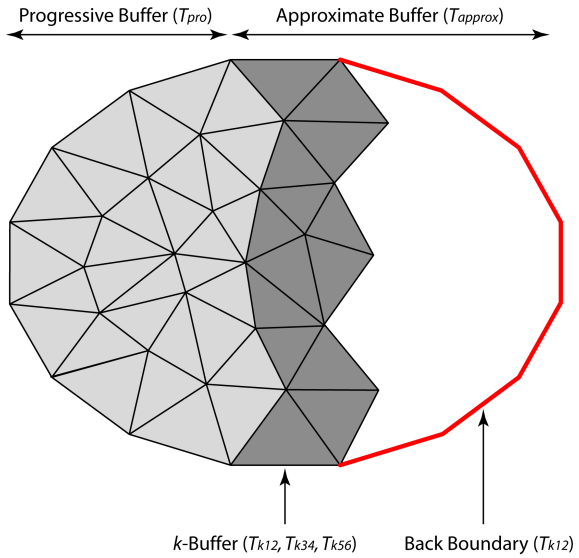


Fig. 3. The progressive volume renderer on the client. A progressive buffer is maintained between steps and an approximate buffer is created to fill the unknown region of the mesh.

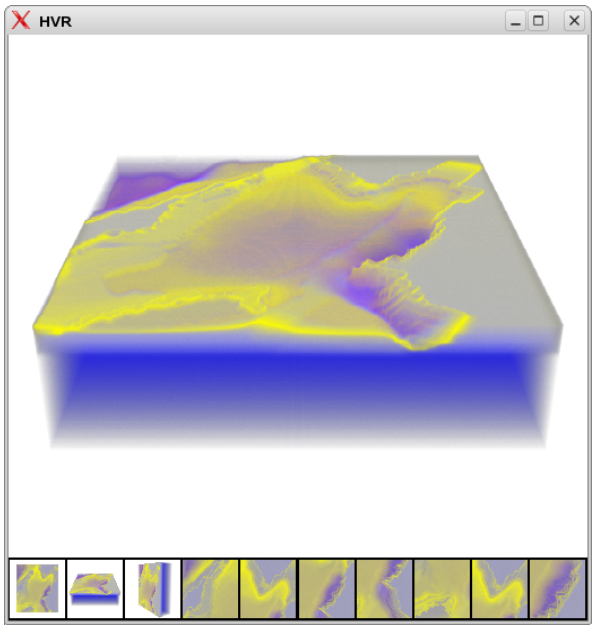


Fig. 4. A snapshot of the interaction with the *Completed Mode* for the SF1 dataset. Upon completion of a full quality rendering, the image is automatically stored for future browsing by selecting the icons at the bottom of the window. The user may also save intermediate steps with a keystroke.

for sorting and compositing. The server handles the object-space sorting, while the client handles the fixed-size image-space sorting and compositing.

The $k$-buffer is implemented in hardware using multiple render targets (MRTs), which allows the reading and writing of multiple off-screen textures in each pass. Currently, hardware limits the number of MRTs to four, which limits the size of $k$ to six (one texture for an off-screen framebuffer and three textures for $k$-buffer entries). In OpenGL, the simplest access to multiple render targets is in the form of Framebuffer Objects (FBOs). An FBO is a collection of logical buffers such as color, depth, or stencil. Multiple color buffers (up to four) can be *attached* to an FBO for off-screen rendering. FBOs make it possible (and efficient) to swap attached buffers between rendering passes. Currently, this is faster than switching between multiple FBOs. The ability to render into a subset of buffers in multiple passes is at the heart of our progressive algorithm.

The progressive volume rendering works by using five different render targets for each pass, represented as four channel, 32-bit floating-point textures. The textures are used as follows:

$T_{pro}$: An off-screen framebuffer for the progressive image $(R_p, G_p, B_p, A_p)$.

$T_{k12}$: $k$-buffer entries 1 and 2 $(v_1, d_1, v_2, d_2)$.

$T_{k34}$: $k$-buffer entries 3 and 4 $(v_3, d_3, v_4, d_4)$.

$T_{k56}$: $k$-buffer entries 5 and 6 $(v_5, d_5, v_6, d_6)$.

$T_{approx}$: A temporary framebuffer for the approximation of the portion of the mesh not yet received $(R_a, G_a, B_a, A_a)$.

A combination of these textures is used for each step of the progressive volume rendering. The contents of all the textures except $T_{approx}$ are reused in subsequent progressive passes.

Our progressive volume rendering is separated into three modes of operation. *Interactive Mode* is used during camera events such as rotation, pan, or zoom. *Progressive Mode* is used when interaction stops to stream triangles from the server to the client in chunks. The *Progressive Mode* can be interrupted at any time if the user begins interaction again or the stream finishes. When a complete image is generated with the *Progressive Mode*, *Completed Mode* automatically stores the image for future browsing.

### 5.1 Interactive Mode

A reduced representation of the original mesh is often necessary when considering large datasets. Our interactive mode has two requirements. First, it is fast enough to render at interactive rates (*e.g.*, 10 fps). Second, the results of the interactive mode can be used as a first step in the progressive volume rendering. Because of the second requirement, level-of-detail techniques that require simplification hierarchies [8] are not feasible. Instead, we use a similar approach to Callahan *et al.* [5], where a subset of the original geometry is used to create a reduced representation. In particular, Callahan *et al.* noticed that an efficient approximation of the dataset can be created by computing the volume rendering integral between only the boundary fragments of the mesh. We use a more robust version of this approximation that can be used in future progressive steps.

Upon connection with the server, the client receives some initial data to begin the progressive rendering process. First, mesh parameters such as maximum edge length and scalar range are transferred for creating a pre-integrated lookup table. Immediately following these parameters, the boundary triangles of the mesh are transferred to the client. These boundary triangles are placed in arrays on GPU memory and remain there for the duration of the client-server connection. If the entire boundary cannot be maintained in memory, a subset of the boundary can be used instead.

During user interaction (*i.e.*, rotation, panning, or zooming), the following steps take place to create an approximate image.

1. Buffer $T_{k12}$ is attached to the FBO as well as a depth buffer. The depth buffer is cleared and set to `GL_LESS` for a first pass on the boundary geometry. The depth buffer is then cleared again and set to `GL_GREATER` for a second pass at the boundary geometry. This has the same effect as depth-peeling [11] the front and back fragments and placing them in the $k$-buffer.

2. Buffers $T_{approx}$ and $T_{k12}$ are attached to the FBO and a screen-aligned plane is rendered. Color and opacity for the ray-gaps between the front and back fragments from the boundary are looked up from $T_{k12}$ and composited into $T_{approx}$.

3. Buffer $T_{approx}$ is displayed to the screen.

These steps are repeated for every view change. When the user stops interacting, the entries in $T_{k12}$ are used for the progressive steps. The importance of using this depth-peeling approach instead of the $k$-buffer directly is discussed in more detail in Section 5.4.

### 5.2 Progressive Mode

After the boundary has been drawn in *Interactive Mode*, the internal triangles of the mesh are streamed in an approximate front-to-back order based on their centroids. For each portion of the geometry that arrives from the server, the progressive volume renderer takes the following steps:

1. Buffer $T_{approx}$ is cleared.
2. Buffers $T_{pro}$, $T_{k12}$, $T_{k34}$, and $T_{k56}$ are attached to the FBO and the incoming internal geometry is rendered. The $k$-buffer is used to sort the fragments and composite the results into $T_{pro}$. This step is similar to HAVS, but with only a portion of the geometry.
3. Buffer $T_{approx}$ is attached to the FBO, $T_{k12}$, $T_{k34}$ and $T_{k56}$ are bound as a read-only textures, and a screen-aligned plane is drawn. The fragment shader finds the $k$-buffer entry $f$ closest to the current view and the entry $b$ farthest from the current view, looks up the color and opacity for the ray gap between $f$ and $b$, and composites the result into $T_{approx}$.
4. Buffers $T_{pro}$ and $T_{approx}$ are attached to the FBO and a screen-aligned plane is drawn. The fragment shader composites $T_{pro}$ into $T_{approx}$.
5. Buffer $T_{approx}$ is displayed to the screen.

The $k$-buffer entries and progressive buffer $T_{pro}$ are then ready to be used in the next rendering pass. Figure 3 illustrates the textures used during a progressive step. This process is repeated until all the geometry has been streamed from the server. When rendering the last portion of geometry from the server, an additional step is taken to flush the $k$-buffer entries into the $T_{pro}$ by rendering $k-1$ screen-aligned planes after the second step. After which, the $T_{pro}$ buffer contains the full quality volume rendering.

### 5.3 Completed Mode

Once the stream of geometry has terminated and the progressive volume rendering is completed, we capture the $T_{pro}$ buffer and save it for subsequent browsing as shown if Figure 4. Our progressive volume renderer allows the user to browse any previously completed visualizations by selecting the corresponding thumbnail. Capturing this data for the user has important benefits. First, it prevents the user from losing important visualizations through interactions that could reset the results of the previous stream. Second, it allows a user to specify a set of camera positions to the progressive volume renderer so the user can easily capture an animation of the exploration process. This tool is useful for exploring previously generated results while the current view is being progressively rendered off-screen.

### 5.4 Considerations

The $k$-buffer algorithm efficiently handles streaming geometry because it simultaneously reads and writes from textures at each pass. Due to the highly parallel nature of GPU architectures, this may result in a race condition for overlapping triangles. Because the HAVS algorithm sends geometry sorted by centroid, it effectively layers the geometry in the depth direction and thus avoids these errors. However, since the depth compexity of the boundary is generally small and we want to keep the interaction as fast as possible, we do not sort the boundary faces in object-space before rendering. This may result in some noticeable artifacts in the first pass that would remain in subsequent progressive steps. Therefore, we perform the depth-peeling of the front and back fragments before inserting them into the $k$-buffer. This resolves the race condition, improves the quality of the rendering, and maintains interactive rates (see Section 6).

Fig. 5. A zoomed-in view of the STP dataset (about 25M tetrahedra) during interaction. Significant performance improvements are made by frustum-culling the geometry on the server. Here, 50% of the geometry is culled during the progression.

The depth-peeling as described has the unfortunate side-effect that it removes any non-convexities in the boundary. This can propagate through the progressive steps and cause the empty space to be composited into the final image. Since storing the back fragments in the $k$-buffer effectively reduces $k$ by one during the progressive steps, storing all boundary information for non-convex objects can severely impact sorting capabilities. A solution to this problem is to transmit boundary and internal faces during progressive steps. This would introduce redundant fragments only in the front and back and allow other boundary fragments to be used in the progression steps to avoid compositing empty space, as in [6]. To completely remove the storage overhead of the back boundary in the $k$-buffer, an extra texture can be used to store the back fragments during *Interactive Mode* and can be bound as a read-only texture during *Progressive Mode*.

## 6 Results

Our experimental results were measured on a thin client (IBM T41 laptop) running Windows with a 1.7 GHz Pentium M processor, 1.5 GB RAM, and an ATI Mobility Fire GL T2 graphics card with 128 MB RAM. The server machine was running Linux with two Dual core Opteron 2.25 GHz processors, 8 GB RAM, and an NVidia GeForce 7800 GTX graphics card. Performance timings are measured with a $512 \times 512$ viewport on a 100 Megabit/sec ethernet network with regular network load. Our experiments include timing results for the progressive rendering with local and remote configurations, as well as

| Dataset | Tetrahedra | Preprocess | Size |
|---|---|---|---|
| Fighter | 1.4 M | 15 s | 117 MB |
| F16 | 6.3 M | 81 s | 531 MB |
| SF1 | 13.9 M | 110 s | 1165 MB |
| STP | 25.0 M | 458 s | 2087 MB |

Table 1. Experimental datasets with initial tetrahedra count, time to preprocess tetrahedral mesh to binary triangle format, and resulting size of the dataset.

| Dataset | Server Preprocess | | Server | | Data Transfer | Client | | Total |
|---|---|---|---|---|---|---|---|---|
| | Load | Octree | Traverse | Sort | | Interactive | Progressive | |
| Local | | | | | | | | |
| Fighter | 0.25 s | 0.92 s | 0.79 s | 0.78 s | 1.32 s | 0.01 s | 0.21 s | 1.53 s |
| F16 | 1.10 s | 6.12 s | 1.17 s | 4.18 s | 4.17 s | 0.01 s | 0.39 s | 4.56 s |
| SF1 | 2.53 s | 7.87 s | 9.04 s | 7.90 s | 16.16 s | 0.01 s | 0.43 s | 16.59 s |
| SF1 (25%) | 2.53 s | 7.87 s | 2.32 s | 1.49 s | 3.10 s | 0.01 s | 0.71 s | 3.81 s |
| STP | 36.55 s | 18.46 s | 8.62 s | 18.89 s | 21.80 s | 0.38 s | 9.82 s | 31.62 s |
| STP (25%) | 36.55 s | 18.46 s | 2.13 s | 2.96 s | 3.65 s | 0.38 s | 0.36 s | 4.01 s |
| Ethernet | | | | | | | | |
| Fighter | 0.25 s | 0.92 s | 0.79 s | 0.78 s | 13.12 s | 0.10 s | 10.77 s | 23.89 s |
| F16 | 1.10 s | 6.12 s | 3.95 s | 4.68 s | 102.51 | 0.10 s | 128.06 s | 230.57 s |
| SF1 | 2.53 s | 7.87 s | 10.05 s | 9.17 s | 237.62 s | 0.24 s | 501.15 s | 738.77 s |
| SF1 (25%) | 2.53 s | 7.87 s | 2.41 s | 1.70 s | 87.73 s | 0.24 s | 32.37 s | 120.10 s |
| STP | 36.55 s | 18.46 s | 51.74 s | 17.26 s | 727.86 s | 0.45 s | 551.98 s | 1279.84 s |
| STP (25%) | 36.55 s | 18.46 s | 11.19 s | 2.73 s | 208.05 s | 0.45 s | 121.65 s | 329.70 s |

Table 2. Performance analysis of the preprocessing and one step of the progressive volume rendering. Measurements were obtained for a client running on the server (Local) and on a laptop over the network (Ethernet).

error measurements for the progressive images. Table 1 shows the tetrahedra count of our test datasets, the one-time penalty to reformat them into our binary format used by the server, and the resulting size of the binary files.

Our timing measurements are shown in Table 2 for four large meshes. The Fighter and F16 datasets are simulations of jets, the SF1 dataset is an earthquake simulation, and the STP dataset is a simulation of a sphere going through a plate. The measurements can be broken into four important sections: server preprocessing, server, data transfer from the client to the server, and client. The preprocessing step occurs on the server and includes loading the file from disk, and building an octree, and transferring the mesh information to the client. By extending our binary file format to include the octree structure, the server preprocessing time could be decreased even further. The timing results for the server, client, data transfer, and total time represent one progressive rendering of the dataset from views that include the whole mesh. In addition, for the larger datasets, we chose a view showing only 25% of the mesh, which takes advantage of frustum culling (see Figures 5, 7, and 8). Because our client uses a thread for rendering and another for fetching data from the server, much of the data transfer and rendering work is done in parallel. Therefore, we measure data transfer as the total client time for a progressive step minus the rendering time. In our experiments, the interactive manipulation of our progressive renderer was able to achieve interactive rates for all but the largest dataset on the thin client.

| Stream Size | Local | Ethernet | Wireless |
|---|---|---|---|
| 1K Triangles | 0.082 s | 0.050 s | 0.051 s |
| 10K Triangles | 0.085 s | 0.090 s | 0.130 s |
| 100K Triangles | 0.574 s | 1.042 s | 1.893 s |

Table 3. Latency analysis of different server settings on the Fighter dataset with an octree depth of seven, a 802.11 wireless network at 54 Mbps, and an ethernet network at 100 Mbps.

Since data transfer over the network is one of the main bottlenecks of our progressive renderer, we generated experiments to tune the stream parameters. One important consideration is the latency of the network. Several settings on the server effect the round-trip latency of the system — the time for the client to send a packet and receive a response. An obvious consideration is the compression of the geometry during transmission. For our network rendering, we used full compression of the stream and achieved about a 60% compression rate, which dramatically improved performance. Other important considerations include the octree resolution and stream size (number of triangles sent on each progressive step). Finer octree resolution and higher stream size improves performance, but decreases the number of progressive steps and increases memory usage on the client. Table 3 shows the effects of these parameters on the latency for the Fighter dataset. For our experiments with the thin client (see Table 2), we were limited to a stream size of 100K triangles and an octree resolution of 1K triangles per octree node.

Our final experiment was to analyze the quality of the progressive steps. To measure this metric, we used root mean squared (RMS) error to compare incremental steps with the final rendering for all of our experimental datasets. Figure 6 shows a plot of these errors as the progression refines. Since the quality of the approximation is directly related to the transfer function, we used the transfer functions shown in our figures that highlight the more relevant portions of the data. These results show that the image quality steadily converges to the full quality image, which is important for allowing the user to explore the dataset efficiently.

## 7 Discussion

Our progressive volume rendering system is unique because it efficiently handles data of arbitrary size on a thin client. This is done using a novel technique which keeps only image data and boundary geometry in GPU memory on the client for each step. The amount of memory used on the client can be bounded by adjusting the stream size and the size of the boundary. In our experiments, the boundary was not large enough to adversely effect performance or expend memory constraints. In fact, even with a 25 million tetrahedron dataset, the boundary can be volume rendered with our algorithm on a thin client in *Interactive Mode* at about two frames-per-second. This interactivity depends heavily on the boundary complexity of the dataset. If a dataset has a boundary too large to fit in GPU memory or render at acceptable rates, our algorithm would work efficiently by using only a random subset of the boundaries for an approximate rendering during interaction. This would have the effect of lightening the general appearance of the approximation. The remaining boundary triangles could then be streamed before the rest of the internal faces.

An important consideration for a progressive renderer is the depth complexity and structure of the dataset. The client rendering is fill-bound and thus depends more on the view selected or screen size then on the depth complexity. However, the depth complexity of the dataset may adversely affect performance of our geometry processing on the server because more culling and sorting passes are required. Our approach for culling by depth on the server assumes an even distribution of triangles. For most of the experimental datasets, this results in few triangles selected in some ranges and many in others. This is balanced by accumulating triangles on the server until a target packet size is reached. A better approach may be to avoid using fixed depth ranges by traversing the octree incrementally from the front to the back, rather than doing a hierarchical culling. In our experiments, the aspect ratio or depth complexity of the dataset seems to impact overall performance only slightly if the server parameters are properly selected

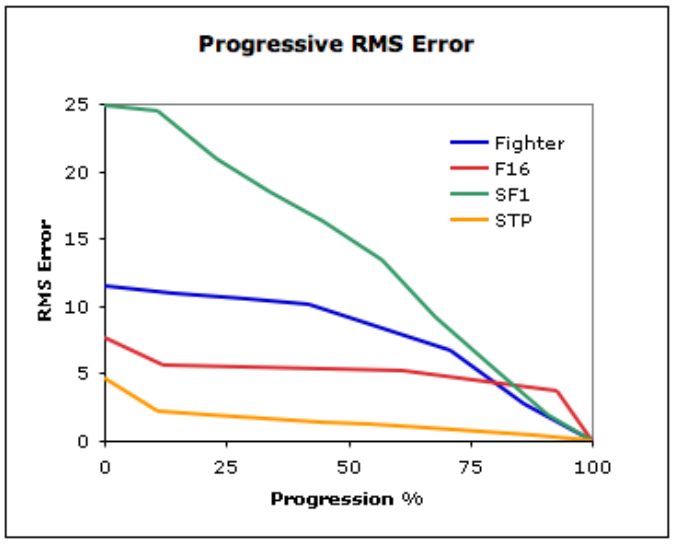


Fig. 6. Root Mean Squared Error (RMS) of the progressive images going from only the boundaries (0%) to the full quality (100%).

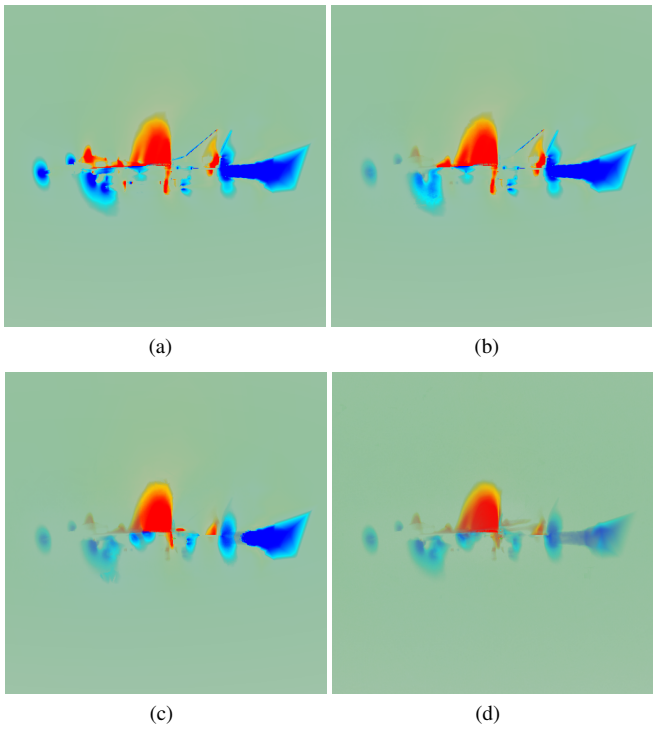


Fig. 7. A zoomed-in portion of the F16 dataset (about six million tetrahedra) shown progressively at (a) 0%, (b) 12%, (c) 61%, and (d) 100%.

(number of depth ranges and minimum packet size).

A client-server progressive volume renderer is advantageous because it allows the data to be stored in a central repository, while the rendering can be performed with the help of graphics hardware on a client. This efficiently splits the load of the client and server for datasets that may be too large to render locally. Since processing power continues to increase rapidly for both CPUs and GPUs, it is becoming increasingly important to develop algorithms that efficiently harness the computational power without being limited by memory constraints. Our algorithm uses this approach by acting as a stream renderer and allowing the interactive exploration of a dataset with only a portion of the geometry.

The main disadvantage of using this paradigm is that data transfer over the network incurs a substantial penalty. We reduce this penalty partially with the use of lossless compression of the stream. If some loss in quality is acceptable, quantization techniques could also be applied to reduce the bandwidth of the geometry. Because data transfer is a limiting factor in quickly visualizing a full quality rendering, the quality of the approximation is important. Unlike naïve approaches that render only an opaque boundary mesh or outline, our initial approximation gives excellent results with little overhead. With only a few iterations, our progressive volume renderer converges to the final image which facilitates dynamic exploration. We find this aspect useful because often in the exploration process, the user will not wait for the entire progressive volume rendering before moving on to another viewpoint. Because only the geometry within the current view frustum is transferred, efficiently exploring details of the dataset becomes easy. This feature also makes rapid transfer function exploration possible. With each update of the transfer function, the stream can be reset and the progression started. For datasets with more important features in the center, the boundary may not give a good approximation. A more advanced technique that uses multiple $k$-buffers could be employed to render several advancing progressions at once which results in an increased rendering overhead. A simpler approach would be the addition of user-controlled cutting planes that could cull geometry on the server, thus reducing the amount of data sent to the client and allowing the rapid visualization of internal structures. Along with the image capturing system, which keeps previously computed results, these features would provide a powerful data exploration tool for large datasets.

## 8 Conclusions and Future Work

Our algorithm provides a progressive volume rendering system for interactively exploring large unstructured datasets. We use a novel approach of balancing the load of the client and server while minimizing the memory constraints of the client. In fact, our algorithm can bound the memory usage of the client machine to allow a wide variety of client devices. We have introduced a novel progressive algorithm that efficiently uses the GPU to incrementally refine the visualization by retaining only image data. An interactive mode can be efficiently computed with the addition of boundary triangles that can be kept in GPU memory. To further improve interaction, our system keeps previously computed visualizations that can be interactively browsed while progressively rendering the visualization. Finally, we have provided detailed experiments and discussed the trade-offs of a client-server approach for volume rendering unstructured grids.

In the future, we would like to explore more efficient methods of reducing the data size for transmission from server to client. In particular, geometry compression algorithms (*e.g.*, [4]) could alleviate bandwidth constraints. We would also like to add a more advanced depth culling algorithm to avoid manual parameter tuning and add user-controlled cutting planes to facilitate exploration further. Another area of future research is to extend our system to render isosurfaces and to explore the visualization of time-varying data in a progressive manner.

## Acknowledgments

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. The authors acknowledge Neely and Batina (NASA) for the Fighter dataset, Tremel (EADS-M) for the F16 dataset, O'Hallaron and Shewchuk (CMU) for the SF1 dataset, and the Army Research Laboratory (ARL) for the STP dataset. The authors also thank the reviewers for insightful comments and suggestions. Steven P. Callahan, Louis Bavoil, and Cláudio T. Silva are funded by the National Science Foundation (grants CCF-0401498, EIA-0323604, OISE-0405402, IIS-0513692, and CCF-0528201), the Department of Energy, an IBM Faculty Award, Sandia National Laboratories, Lawrence Livermore National Laboratory, the Army Research Office, and a University of Utah Seed Grant.

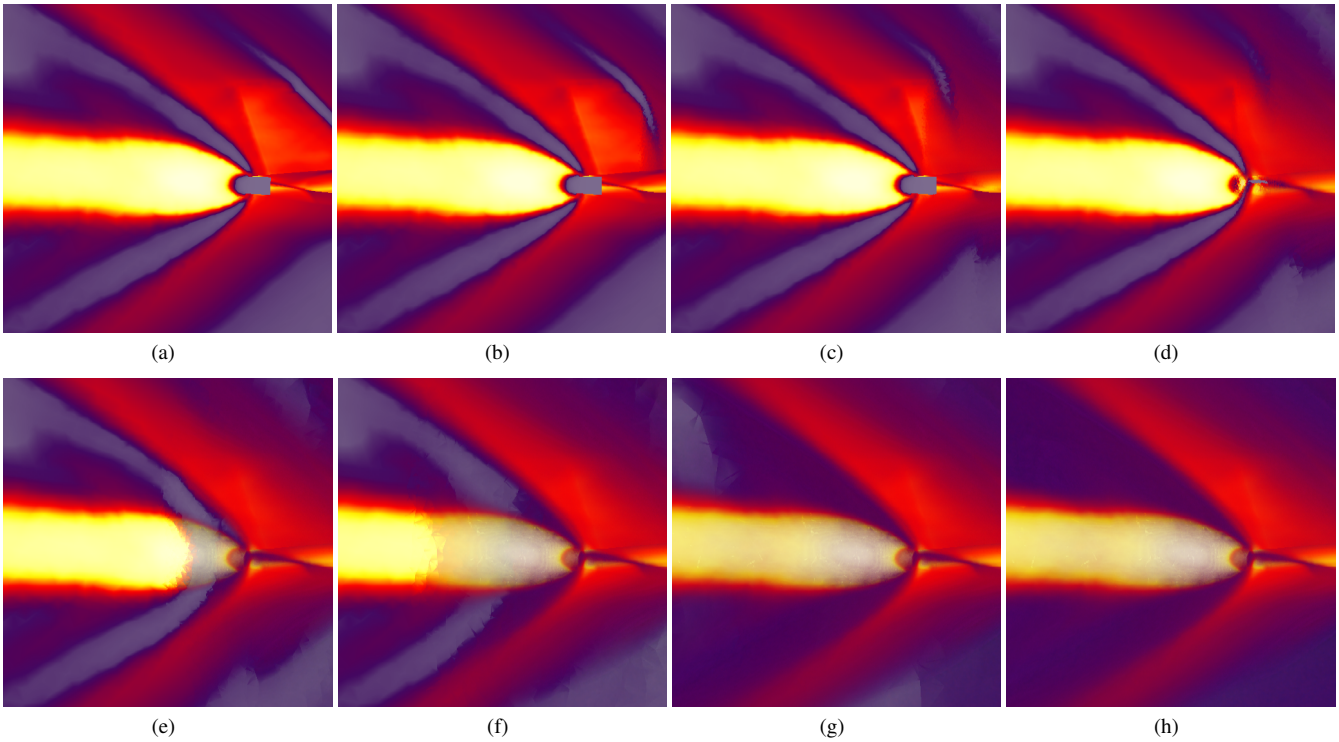


Fig. 8. A zoomed-in portion of the Fighter dataset (about 1.4 million tetrahedra) shown progressively at (a) 0%, (b) 14%, (c) 29%, (d) 43%, (e) 57%, (f) 71%, (g) 86%, and (h) 100%. View-frustum culling on the server removed 75% of the original geometry for this view.

## REFERENCES


[1] F. F. Bernardon, S. P. Callahan, J. L. D. Comba, and C. T. Silva. Volume rendering of unstructured grids with time-varying scalar fields. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 51–58, 2006.

[2] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. GPU-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, to appear. Also available as SCI Institute Technical Report UUSCI-2004-006.

[3] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Supercomputing '00*, page 28, 2000.

[4] U. Bischoff and J. Rossignac. Tetstreamer: Compressed back-to-front transmission of delauney tetrahedra meshes. In *Data Compression Conference*, 2005.

[5] S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva. Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05*, pages 199–206, 2005.

[6] S. P. Callahan, M. Ikits, J. L. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[7] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, volume 18, pages 103–108, July 1984.

[8] P. Cignoni, L. D. Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):29–45, 2004.

[9] K. Engel, O. Sommer, C. Ernst, and T. Ertl. Progressive isosurfaces on the web. Late Breaking Hot Topics, IEEE Visualization '98, 1998.

[10] K. Engel, O. Sommer, and T. Ertl. A framework for interactive hardware accelerated remote 3D-visualization. In *EG/IEEE TCVG Symposium on Visualization (VisSym)*, 2000.

[11] C. Everitt. Interactive order-independent transparency. White paper, NVIDIA Corporation, 1999.

[12] M. Garland and Y. Zhou. Quadric-based simplification in any dimension. *ACM Transactions on Graphics*, 24(2), Apr. 2005.

[13] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02*, pages 693–702, 2002.

[14] R. Kaehler, S. Prohaska, A. Hutanu, and H.-C. Hege. Visualization of time-dependent remote adaptive mesh refinement data. In *IEEE Visualization '05*, pages 175–182, 2005.

[15] J. Leven, J. Corso, J. D. Cohen, and S. Kumar. Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics*, pages 37–44, 2002.

[16] L. Lippert, M. H. Gross, and C. Kurmann. Compression domain volume rendering for distributed environments. *Computer Graphics Forum*, 16(3):C95–C107, 1997.

[17] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann Publishers, 2002.

[18] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 85–92, 2001.

[19] K. Museth and S. Lombeyda. Tetsplat: Real-time rendering and volume clipping of large unstructured tetrahedral meshes. In *Proceedings of IEEE Visualization 2004*, pages 433–440, 2004.

[20] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization '98*, pages 233–238, 1998.

[21] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Proceedings of San Diego Workshop on Volume Visualization*, 24(5):63–70, 1990.

[22] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon. A survey of GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA)*, 12(2):9–29, 2005.

[23] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *IEEE Visualization '03*, pages 333–340, Oct. 2003.

[24] M. Weiler, P. N. Mallón, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proceedings of Symposium on Volume Visualization 2004*, pages 71–78. IEEE, 2004.

[25] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transaction on Graphics*, 11(3):201–227, 1992.

[26] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, Apr. 1992.