Eurographics Symposium on Parallel Graphics and Visualization (2006), pp. 1–8
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

iRun: Interactive Rendering of Large Unstructured Grids

Paper ID: 1027

Abstract

We present iRun, a system for interactively volume rendering large unstructured grids on commodity PCs. Rendering arbitrarily large datasets has been an active area of research for many years. However, the techniques required for polygonal data do not directly apply to the more complex problem of unstructured grids. In this paper, we describe the data structures and algorithms necessary to store large datasets on disk, keep an active portion of the dataset in main memory, and render visible regions to one or more displays. Our system leverages a combination of out-of-core data management, distributed rendering, hardware-accelerated volume rendering, and dynamic level-of-detail. On a commodity PC, our system can preprocess a dataset consisting of about 14 million tetrahedra in about a half an hour and can render it interactively with one or more PCs.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems

1. Introduction

Interactive rendering of arbitrarily large datasets is a fundamental problem in computer graphics and scientific visualization, and a critical capability for many real applications. Interactive visualization of large datasets poses substantial challenges (see survey by Silva *et al.* [SCESL02]). Current systems for rendering large datasets employ many of the elements proposed by Clark [Cla76] including the use of hierarchical spatial data structures, level-of-detail (LOD) management, hierarchical view-frustum and occlusion culling, and working-set management (geometry caching). Systems along the lines of the one envisioned by Clark have been used effectively in industrial applications for scenes composed primarily of polygonal geometry.

For more complex scenes, such as those composed of tetrahedral elements, the problem is not as well studied and can be more difficult for several reasons. First, rendering tetrahedra is not natively supported by current graphics hardware. Thus, efficient algorithms for handling this type of data robustly are required. Second, tetrahedra must be projected in visibility order to accurately composite transparency. This requires special care to traverse the out-of-core hierarchy in the correct order. Finally, visibility techniques such as occlusion culling are not practical because the opacity of the volume is controlled by the user. iRun addresses these issues while still maintaining interactivity on extremely large dataset.

The visualization pipeline may be broken down into four major stages: retrieval from storage, processing in main memory, rendering in the Graphics Processing Unit (GPU), and display on the screen. The performance of each of these stages is limited by several potential bottlenecks (*e.g.*, disk or network bandwidth, main memory size, GPU triangle throughput, and screen resolution). iRun uses out-of-core data management and speculative visibility prefetching to maintain a working-set of the geometry in memory. Our rendering approach uses GPU-assisted volume rendering with a dynamic set of tetrahedra and uses an out-of-core LOD traversal. Finally, our system was implemented in VTK and allows distributed rendering for high-resolution displays. Using a single commodity PC, we show how our system can render datasets consisting of 14 million tetrahedra while maintaining interactive frame rates (see Figure 1).

The main contributions of this paper are:

- We present a system that can volume render tens of millions of tetrahedra at interactive frame rates on commodity PCs;
- We introduce new data structures for out-of-core management of large volumetric meshes;
- Our system uses a novel approach for dynamic level-of-detail traversal of our out-of-core data structures using speculative prefetching;
- We show how state-of-the-art, hardware-accelerated volume rendering can be used in a distributed environment to interactively render these large datasets to multiple displays.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 provides an overview of our volume rendering system, including preprocessing and traversal of out-of-core data structures, hardware-assisted volume rendering of the active set, and distributed rendering for high-resolution displays. The results of our algorithm are shown in Section 4. In Section 5 we discuss some of the issues we encountered while developing the iRun system and differentiate our approach from similar approaches for large polygonal models. Finally, in Section 6 we conclude and provide areas for future research.

2. Related Work

Walk-Through Systems. Pioneering work on rendering large models at interactive rates by Clark [Cla76] continues to be the basis for many systems used today. His original system proposed novel data management techniques such as hierarchical spatial data structures, level-of-detail (LOD) management, hierarchical view-frustum culling and occlusion culling, and working-set management.

The first system to handle models larger than main memory was introduced by Funkhouser *et al.* [FST92]. This system maintained interactive rates for large architectural models by swapping visible portions of the mesh into memory using speculative prefetching based on from-region visibility. The ROAM system [DWS*97] uses an alternative approach that applies split and merge operations to create view-dependent triangulations and texture maps to maintain a desired frame rate. Work by Aliaga *et al.* [ACW*99] interactively renders models with tens of millions of polygons by employing LOD management and visibility culling for near regions and pre-rendered image impostors for far regions. The drawback to this approach is that it uses a preprocessing step that requires user intervention and weeks to run. Recent work by Varadhan and Manocha [VM02] uses a parallel approach with workstations to perform hierarchical LOD rendering and update the working-set based view-frustum and simplification culling.

More recently, the iWalk system introduced by Correa *et al.* [CKS02] uses efficient data structures to render large out-of-core polygonal models on a commodity PC. iWalk uses a multithreaded approach which couples speculative prefetching with from-point visibility to manage the working-set while rendering the scene. The system reduces the preprocessing step to minutes while remaining fully automatic. The iRun design is inspired by iWalk, but requires a more complex solution for volume rendering on unstructured grids. Due to many similarities in the two approaches, we defer comparisons to Section 5.

Visibility. For polygonal meshes, visibility algorithms such as view-frustum and occlusion culling are important for maintaining interactive frame rates (see [COCSD03] for a recent survey). For volume rendering, occlusion culling is not feasible due to transparency. Therefore, we are more interested in view-frustum techniques that allow efficient prefetching of visible geometry. El-Sana *et al.* [ESSS01] describe a system that efficiently combines LOD traversal with occlusion culling for interactive rendering. Correa *et al.* [CKS03] describe a multithreaded from-point visibility approach which is used in the iWalk system and relies on occlusion and view-frustum culling to maintain interactivity.

Out-of-Core Algorithms. External memory data structures are an important component for dealing with models too large for main memory. Out-of-core structures have been used for memory sensitive applications such as large model simplification [CRMS03, Lin00] and isosurface extraction [CSS98]. For rendering large models, hierarchical external data structures are frequently used. El-Sana and Chiang [ESC00] build out-of-core, view-dependent trees that are used to maintain interactive LOD. Our system uses a similar approach to Cignoni *et al.* [CFM*04] which uses an out-of-core octree [Sam90] to store a multi-resolution or LOD representation of the full mesh for volume rendering. However, instead of storing progressive refinement operations, we explicitly store LOD geometry in the octree nodes.

Hardware-Assisted Volume Rendering. Leveraging the performance of graphics processing units (GPUs) for direct volume rendering has received considerable attention (for a recent survey, see [SCCB05]). Shirley and Tuchman [ST90] introduces the Projected Tetrahedra (PT) algorithm, which splits tetrahedra into GPU renderable triangles based on the view direction. For correct compositing, the neighborhood information of the original mesh is used to determine an order dependence. More recent work by Weiler *et al.* [WKME03] performs ray-casting on the mesh by storing the neighbor information on the GPU and marching through the tetrahedra in rendering passes. As with PT, the hardware ray caster requires the neighbor information of the tetrahedra for correct visibility ordering. A more flexible approach was introduced by Callahan *et al.* [CICS05], which operates on the triangles that compose the mesh and requires no neighbor information. This makes immediate mode rendering, working-set management, LOD, and preprocessing much simpler than it would be by using tetrahedra directly. Because of this flexibility and the speed of the approach, we use an extended version of the HAVS algorithm in iRun.

Level-of-Detail. Many approaches have been developed for LOD [LE97,LRC*02] rendering of polygonal models. However, work on volume rendering for unstructured grids is not as well studied. Leven *et al.* [LCCK02] sample the unstructured grid as a structured one and use texture-based multiresolution techniques on the resampled data. Museth *et al.* [ML04] render the volume as opaque points and incorporate CSG operations to explore the internal regions of the mesh. Cignoni *et al.* [CFM*04] use a progressive hierarchy which captures simplification steps in a data structure which can be traversed at a desired LOD. More recently, Callahan *et al.* [CCSS05] introduced *sample-based* simplification, which samples the original geometry to create a LOD.

iRun extends this approach to use a hierarchical representation for out-of-core traversal. This approach works well with our out-of-core data structures and can be efficiently used for hardware-assisted volume rendering.

Distributed rendering. Though our algorithm can efficiently render large unstructured grids on a single commodity PC, a cluster of PCs can be used in a distributed manner to improve image quality. Chromium [HHN*02] is a system that was introduced to perform parallel rendering on a cluster of graphics workstations. Distributed rendering can also be used to visualize the data on larger displays. Moreland and Thompson [MT03] describe a parallel rendering algorithm that uses Chromium and an image-composite engine (ICE-T) built with VTK for visualizing the results on a display wall. A major distinction between iRun and Chromium is that while Chromium *pushes* data through the pipeline to the render devices, iRun *pulls* data from a geometry-cache to the render devices. This *pulling* approach results in a conceptually simpler framework for parallel rendering where the CPU and GPU are tightly connected and data is fetched to the geometry-cache as needed before being transformed into graphics primitives.

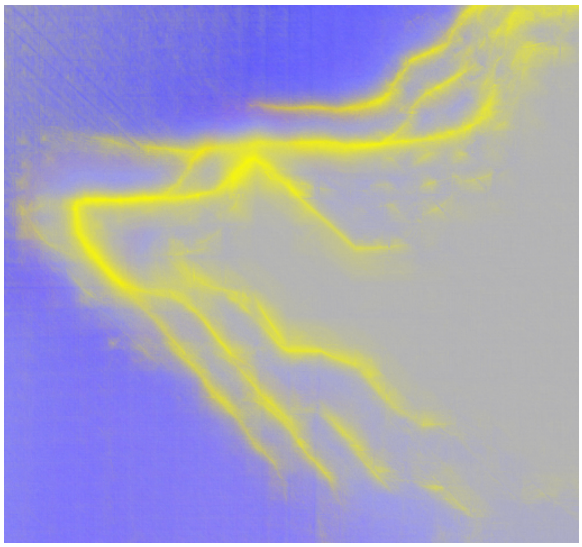

Figure 1: *A close view of the earthquake simulation data Sf1 with 14 millions of tetrahedra. The image is rendered in iRun using a full 512MB RAM geometry cache. iRun allows interactive exploration of datasets too large for main-memory.*

3. Interactive Out-of-Core Volume Rendering

iRun interactively renders large unstructured grids in several stages. First, a preprocessing step prepares the data for hierarchical traversal. Second, our algorithm interactively traverses the out-of-core data structure and keeps a working-set (geometry cache) of the geometry in memory by using visibility culling, speculative prefetching, and LOD management. Finally, the contents of the geometry cache are rendered using a hardware-assisted visibility sorting algorithm (see Figure 2). For improved image quality or large display capability, we describe how our algorithm can be distributed to a cluster of PCs.

3.1. Out-of-Core Preprocessing

iRun utilizes an efficient and fully automatic preprocessing algorithm that operates out-of-core for large datasets. The preprocessing comprises the following steps:

1. Extract the unique triangles and mark boundaries
2. Add internal triangles to the octree leaves, splitting nodes when necessary
3. Fill internal nodes with LOD triangles
4. Add simplified boundary triangles to internal nodes
5. Filter vertices and clip triangles to node bounding-box

For rendering, we begin by extracting the unique triangles that compose the tetrahedral mesh. This is done out-of-core by writing the four triangle indices of each tetrahedron into a file and using an external sort to arrange the indices from first to last. The resulting file contains adjacent duplicate entries for faces that are on the inside of the mesh and unique entries for boundary triangles. A cleanup pass is performed over the triangle index file to remove duplicates and to create a similar file that contains a boundary predicate for each triangle.

iRun employs an out-of-core, hierarchical octree [Sam90] in which each node contains an independent renderable set of vertices and triangles, similar to iWalk. Because the number of vertices in a tetrahedral mesh is generally much smaller than the connectivity information, for simplicity, we keep the vertex array in-core while creating our out-of-core hierarchy. This allows us to keep the global indexing of the triangles throughout our preprocessing which facilitates filtering in the final stages. Our octree is constructed by reading the triangle index and boundary predicate files in blocks and adding the triangles one-by-one to the out-of-core octree structure.

While adding triangles to the octree, we perform triangle-box intersection to determine one or more nodes that contain the triangle. If the triangle spans multiple nodes, we replicate and insert it into each. This can lead to the insertion of a triangle into a node where any or all of the triangle vertices lie outside the node. When a node reaches a preset capacity (*e.g.*, 10,000 triangles), the node is split into octants and the triangles are redistributed. For each block of triangles processed, the octree is flushed and merged into out-of-core nodes to enable the processing of the next block of data. The result of the triangle insertion phase is a hierarchical directory structure that represents the octree with only the leaf nodes containing any actual data and a *hierarchy structure* file that contains the octree structure information (see [CKS02]).

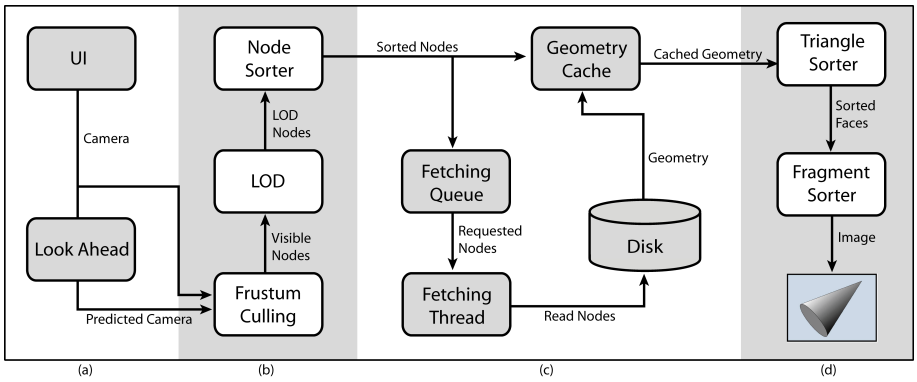

Figure 2: *iRun overview. (a) The user interacts with the UI by changing the camera from which our system predicts future camera positions. (b) Our octree traversal algorithm selects the octree nodes that are in the frustum, determines the appropriate LOD, and arranges the nodes in visibility order. (c) The geometry cache keeps the working-set of triangles while a separate thread is used to fetch additional geometry from disk. (d) Finally, the geometry is sent to the renderer for object-space sorting followed by a hardware-assisted image-space sorting and compositing which is performed using a modified version of the HAVS algorithm.*

A subsequent LOD propagation stage works by populating a parent node with a subset of the triangles that are not on the boundary from each of the children (*e.g.*, 1000 from each child). Again, the triangles are replicated as they move up the octree to create self-contained nodes. The subset is selected based on a dynamic LOD strategy introduced by [CCSS05]. The idea is to sample the full resolution geometry to achieve a subset that minimizes the image error. We choose to select the triangles that are the largest to maximize node coverage and thereby minimize image error for that node. To ensure that there are no holes in the LOD structure, the boundary triangles are simplified to a reduced representation (*e.g.*, 5%) of the original and inserted into every intersecting node except the leaf nodes.

Finally, a cleanup pass on the octree is performed which inserts the referenced vertices into each octree node and clips the triangles based on the bounding box of the node. The use of global vertex indices simplifies the filtering process and the end result is an octree in which each node contains a dataset with unique vertices and triangles.

3.2. Hardware-Assisted Rendering

An important consideration for the interactive rendering of unstructured grids is the choice in volume rendering algorithms. Three aspects need to be considered: speed, quality, and the ability to handle dynamically changing data. By using a hardware-assisted volume rendering system, we can address both the speed and quality issues. However, most solutions are not setup to handle dynamically changing data because they require topological information for the active set. In our system, we use the Hardware-Assisted Visibility Sorting (HAVS) algorithm of Callahan *et al.* [CICS05] because it combines speed, quality, and most importantly, it does not require topological information.

HAVS operates by sorting the geometry in two phases. The first is a partial object-space sort that runs on the CPU. The second is a final image-space sort that runs on the GPU using a fixed A-buffer implemented with fragment shaders called the *k*-buffer. The HAVS algorithm considers only the triangles that make up the tetrahedral mesh, thus it does not require the original tetrahedra nor the neighbor information of the mesh. This allows us to render a subset of the octree nodes independently without merging the geometry.

Unlike rendering systems for opaque polygonal geometry, special care needs to be taken when rendering multiple octree nodes to ensure proper compositing. At each frame, our algorithm resolves the compositing issue by sorting the active set of octree nodes that are in memory in visibility order (front-to-back). When octree nodes of different sizes are in the active set, we sort by the largest common parent of the nodes. The original HAVS algorithm has also been modified to iterate over the active set of nodes in visibility order and perform the object-space and image-space sort on each piece. To ensure a smooth transition between octree nodes, the *k*-buffer is not flushed until the last node is rendered.

3.3. Out-of-Core Dataset Traversal

iRun uses an out-of-core traversal algorithm that has been extensively optimized for volume rendering (Figure 2). For each camera received from the user interface, we apply frustum-culling on the octree to find all nodes that are visible in this view and mark them as visible nodes. Depending on whether or not the user is interacting with iRun, the LOD will decide which nodes are to be rendered next. Next, everything is passed to the visibility sorter and only those that have been cached in the geometry cache are sent to HAVS for rendering while the others are put onto the fetching queue. iRun also does camera prediction for each frame by linearly

```
LOD (Camera C, Node R, PriorityFunction P, int MaxTri)
  PriorityQueue Q;
  R.Selected = true;
  Q.Push(P(C, R), R);
  Total = 0;
  while !(Q.Empty())
    Node N = Q.Pop();
    if N.HasChildren
      TC = the total number of triangles in N's children
      if (Total − N.NumberOfTriangles + TC) < MaxTri
        Total = Total − N.NumberOfTriangles + TC;
        N.Selected = false;
        for i = 0 to 7
          if N.Children[i] is not empty or culled
            N.Children[i].Selected = true;
            Q.Push(P(C, N.Children[i]), N.Children[i]);

SORT (Camera C, Node R, List SortedNodes)
  if R is not culled
    if R.Selected
      SortedNodes.Push(R);
    else if R.HasChildren
      SC = R's children sorted ascendingly by distances to C
      for each node N in SC
        SORT(C, N, SortedNodes);
```

Figure 3: *Pseudo-code for the octree traversal algorithm.*

extrapolating previous camera parameters. All of the nodes selected in the predicted camera will also be put on the fetching queue.

The LOD management of iRun is a top-down approach working in a priority-driven manner. Given a priority function $\mathbf{P}(C,N)$ which assigns priority for every node N of the octree with respect to the camera C, the LOD process starts by adding the root R to a priority queue with the key of $\mathbf{P}(C,R)$. Next, iterations of replacing the highest priority node of the queue with its children are repeatedly executed until such refinement will exceed a predefined number of triangles (Figure 3).

In our experiments, we use two different priority functions to control the LOD of iRun. The first is a Bread-First-Search (BFS) based function that is used during user's interactions: $\mathbf{P_{BFS}}(C,N) =< l,d >$, where l is the depth of N and d is the distance of the bounding box of N to the camera C. In this case, each node's priority is primarily determined by how far it is from the root and subsequently by its distance to the camera when the nodes are on the same level. Briefly, our goal is to evenly distribute data of the octree on the screen to improve the overall visualization of the dataset. While interacting with iRun, the target frame rate can be achieved by setting a limit on the maximum number of triangles rendered in the current frame. This number is calculated based on the number of triangles that were rendered, and the rendering time, for the previous frame.

For increased image quality at a given view, iRun will automatically adjust itself to increase the LOD using as much memory as possible when interaction stops. Since we want to cover as much of the screen as possible, a priority function reflecting the projected screen area is necessary for the LOD. We define $\mathbf{P_{area}}(C,N) = \mathbf{A}$, where $\mathbf{A}$ is the projected area of

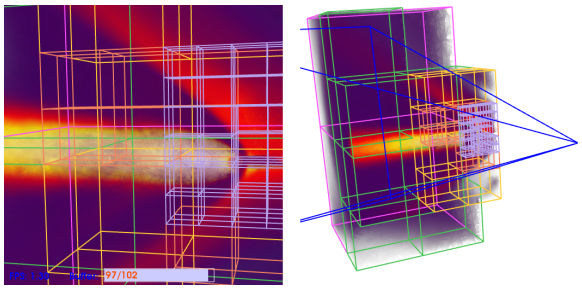

Figure 4: *A snapshot of iRun refining the LOD: The image on the left is rendered as the user would see it from the current camera position. On the right is a bird's-eye view of the same set of visible nodes. Different colors indicate different levels-of-detail. The geometry cache is limited to only* 64*MB of RAM in this case.*

the bounding box of N onto the screen. However, this function can be easily replaced by any other heuristic approaches, such as those reflecting nodes scalar ranges, transfer functions, *etc.*, to achieve the best image quality. The maximum number of triangles to be rendered at this higher image quality is limited to the amount of memory that has been dedicated to the geometry cache.

This approach, however, could raise a problem when the user begins interaction again and the geometry cache is already full. Our next frame would be displayed incompletely since a lower LOD is not available and the higher LOD is too large to be rendered at an interactive rate. To overcome this problem, before increasing the LOD, the current data on the screen will be locked; *i.e.*, it will not be flushed by the geometry cache while fetching new data from disk. When the camera is changed, the previous locked nodes will be unlocked. The trade-off in image quality is insignificant because the amount of memory used by this data is usually very small (*e.g.*, 1%) when compared to the total memory of the geometry cache.

The node visibility sorter ensures everything is in the correct order before compositing in HAVS. In fact, it only takes a single pass through the whole octree to sort all of these nodes (see Figure 3).

iRun separates the fetching from the building of sets of visible nodes. If the fetching queue is empty, the fetching thread will wait until new requests arrive. Otherwise, it will read the requested node from disk and move it to the geometry cache. If the geometry cache is full, the least recently used node will be flushed to provide space for the new request. It also ensures that nodes currently being displayed will not be flushed. As a result, the target frame rate of iRun is guaranteed to stay the same throughout user-interaction since the rendering process will never stall while waiting for nodes to be read in from disk. This improves interaction and does not introduce any significant degradation in image quality to the system. Because none of the visible geometry will

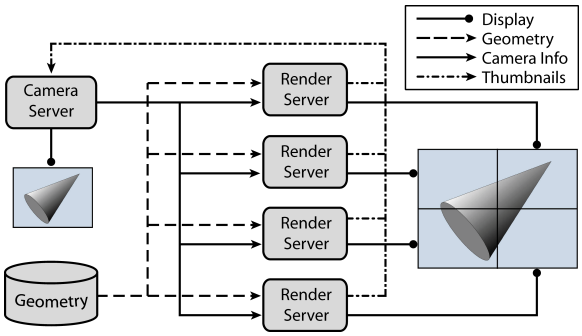

Figure 5: *Distributed rendering with a thumbnail client. The client receives thumbnails from the render-servers and composites them.*

be culled by occlusion, the amount of geometry shared between frames is very similar–every frame will have at least as much data as the previous frame. In the worst case, there will be at most one level of difference in LOD of the frame because of our BFS-based priority function.

3.4. Distributed Rendering

iRun can partition rendering across a distributed network. This feature is useful for driving a display wall, where each system controls a single tile of display. This approach can further improve performance when rendering scenes with complicated geometry.

In iRun, each display on the display wall is driven by a dedicated render-server implemented in VTK. The portion of the display wall that a render-server will be responsible for is specified to the render-server on startup. A skewed view frustum is calculated based on the region of responsibility, and this frustum allows the render-server to cull the geometry to only the set visible on its display. Each render-server has access to the full geometry, but only loads the portions that it needs or anticipates needing.

The render-servers are coordinated by a single system that controls the camera. They receive camera description asynchronously to the render cycle, and wait for the render cycle to complete before updating VTK's camera. The update mechanism is implemented as a `vtkInteractorStyle` to allow for seamless integration.

The camera-server allows render-servers to establish and break TCP connections arbitrarily. Camera descriptions are sent out periodically, regardless of whether the camera has changed, to allow recently connected render-servers to quickly synchronize with the rest of the display wall.

Two clients have been written to run on the camera-server, both implemented in VTK. The first is nearly identical to the render-servers, except that it broadcasts camera coordinates instead of receiving them. This is accomplished by listening for the timer event which the interactor styles use for motion updates. The camera is read inside the event handler

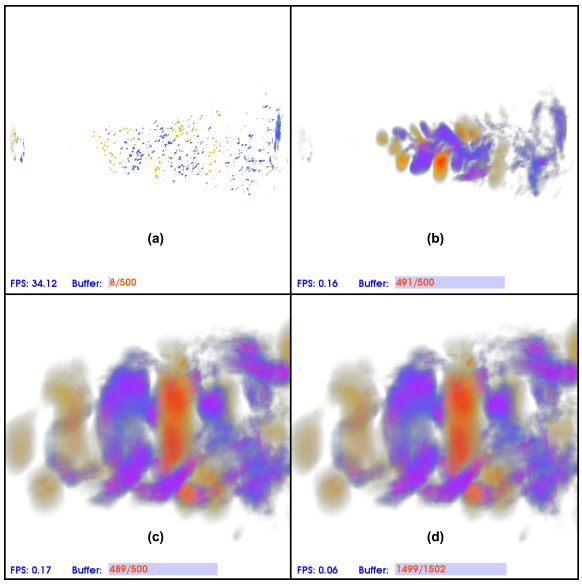

Figure 6: *iRun visualizes the Turbo Jet model consisting of over 10 millions tetrahedra with multiple levels-of-detail. The geometry cache size of (a), (b) and (c) is 512MB while (d) is 1.5GB. (a) In interactive mode, iRun only displays 40K triangles. (b) In still-rendering mode, most of the geometry cache is used. (c) The user zooms in to a particular region. (d) The geometry cache size is tripled.*

and provided to the broadcast code. This client has access to the same geometry and renders it on its own. It requires an adequately functional GPU, and relies on the automatic LOD adjustment to maintain interactive frame rates.

The second client (Figure 5) is able to run on a less capable system. It also runs as a VTK implementation, but uses VTK's interaction styles with no geometry loaded. In this mode, the render-servers additionally frame capture their rendered output and transmit downsampled versions back to the camera-server. The camera server receives and assembles the snapshots asynchronously to the render cycle, and periodically requests the message loop to execute code to display the composite.

4. Results

We generated all of our results on a 3.0-GHz Pentium D machine with 2.0 GB of RAM and a 500 GB SATA hard-disk with an nVIDIA 7800 GTX. Table 1 shows timing results and data sizes before and after preprocessing. We were able to preprocess the largest dataset, which contained 14 million tetrahedra or 28 million triangles, in just over half an hour. For all models in this paper, we target the output octree to have at most ten thousands triangles per leaf. Due to the triangles added during simplification and clipping, the final number of triangles per leaf is slightly higher. The rest of the section will use these datasets unless otherwise stated.

Table 1: *Preprocessing Results*

Data Set	Input				Timing (m:s)		Output			
	Vertices	Tetrahedra	Triangles	Size	Total	Clipping	Tris/leaf	Vertices	Triangles	Size
SPX	2896	12,936	27,252	1 MB	00:01	00:00	10,301	13,059	48,957	1 MB
Torso	168,930	1,082,723	2,168,505	34 MB	01:45	01:12	12,602	3,239,468	6,673,469	158 MB
Fighter	256,614	1,403,504	2,848,760	50 MB	02:27	01:28	18,305	3,640,470	7,791,567	182 MB
Rbl	730,273	3,886,728	7,935,936	143 MB	06:13	04:08	42,527	7,840,753	17,953,898	410 MB
Mito	972,455	5,537,168	11,176,096	206 MB	10:59	05:00	30,406	11,060,893	22,751,951	538 MB
Turbo Jet	1,730,664	10,125,312	20,336,128	344 MB	15:49	07:58	27,009	12,275,915	31,677,170	697 MB
Sf1	2,461,694	13,980,162	28,202,581	516 MB	37:41	25:56	58,390	24,137,623	63,322,069	1,425 MB

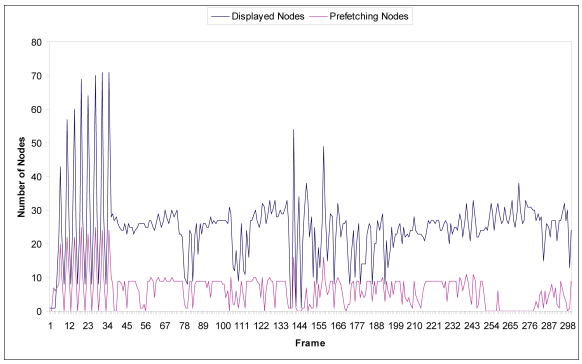

Figure 7: *The difference between the displayed nodes of the geometry cache and the ones being fetched during interactions of the Turbo Jet at 2 fps.*

In Figure 6, we show an example of how iRun can give users various levels-of-detail on demand. Assume a user has a machine with approximately 512MB of RAM available for rendering, and wants to use iRun to explore different features of the Turbo Jet dataset using a 512×512 viewport. The user can use iRun to load the dataset with 512MB of RAM dedicated to the geometry cache and a target frame rate of 5 (for example). Interactions at this frame rate only give the user a low LOD representation of this dataset since iRun can only render as many as 40 thousands triangles at each frame. It is almost impossible to get a coherent overview of the dataset at a resolution less than 0.1%. Thus, the user would want to stop interactions for iRun to refine its visualization. With more detail, the user can now select a specific region of interest to explore. For example, the user may wish to take a closer look at the *orange* part in the center of the dataset. This can be accomplished by zooming, and the LOD will be automatically adjusted by iRun. With a full use of the geometry cache, the quality of our image is very close to the full resolution at that same view using 1.5GB of RAM.

Figure 7 is a plot showing the difference between the displayed and fetched nodes while performing interactions with the Turbo Jet as in the case study above. We selected the target frame rate of 2 frames per second to increase the readability of the plot while the interaction speed is fairly high (mouse-movement of over 30 pixels/frame). Our set of movements contains excessive rotations, zooms, and translates. As shown in the figure, the total number of nodes that need to be loaded for the next frame stays relatively low compared to those that are rendered.

5. Discussion

Limitations. There are issues in our current implementation that could use improvement. First, due to clipping and boundary triangles, node size can grow larger than desired. Bounding this limit would be a useful feature. Another limitation is that our current LOD strategy may not be suitable for all datasets, therefore a more automatic way of determining the LOD triangles would improve image quality. Finally, even though the number of vertices is generally much less than the number of tetrahedra, our current method of keeping the vertices in-core during preprocessing would not be feasible on a PC for extremely large datasets with hundreds of millions of tetrahedra.

VTK. In term of coding, we found VTK to be very helpful when implementing our system. By leveraging existing functionality provided by this framework, we were able to focus on algorithmic instead of engineering contributions. For example, the simplification and clipping phase of the preprocessing are all done using VTK classes. Additionally, the client-server and user interface are built on top of VTK. However, using VTK provided the following challenges:

- Because VTK is for general use, we were forced to modify existing classes to get desired functionality. For example, we added our own timers and modified the rendering pipeline order. In addition, current VTK data structures for geometry are incompatible with OpenGL, which makes using fast display structures such as triangle arrays difficult.
- VTK is not thread-safe. iRun required solving many synchronization problems among threads. For example, one condition was occurring when the visibility sorter received the camera slower than the rendering window, causing compositing artifacts in the resulting image.
- To manage the client-server architecture for parallel rendering, modifications to the VTK interactors were necessary. Specifically, the addition of an asyncExec method to `vtkRenderWindowInteractor` and its derived classes was necessary to queue commands for later execution.

iRun Versus iWalk. iRun shares many ideas with iWalk [CKS02]. However, due to the complexity of volume rendering tetrahedral data, many of the algorithms presented in iWalk were not suitable for iRun. In iRun, each node of the octree contains a self-describing `vtkUnstructuredGrid` of varying levels-of-detail that can be rendered independently. iWalk only keeps triangles in the leaf nodes and depends on occlusion culling instead of LOD to remain interactive. This also affects the traversal algorithm, which is more sophisticated in iRun because LOD as well as screen coverage need to be considered due to the transparent nature of the nodes. The fetching thread also works differently. iRun separates fetching from building visible sets of nodes because of the added complexity of visibility sorting for the nodes.

Another difference is that due to compositing problems that occur with overlapping triangles from neighboring nodes, we require a more exact triangle-box intersection that is based on the triangle, not the vertices. iRun also requires the clipping of triangles that extend beyond the node's bounding box to avoid incorrect visibility ordering across neighboring nodes. Unlike iWalk, we require boundary triangles (simplified or full resolution) at each level of the tree to avoid holes in the rendered image. Finally, we note that the implementation of both systems is completely disjoint. As we note previously, our choice to use VTK has simplified some tasks and while making others more difficult.

6. Conclusion

We have introduced the iRun system, which is capable of volume rendering tens of millions of tetrahedra at interactive rates on a commodity PC. The preprocessing required by our algorithm occurs completely out-of-core and is light, fully automatic, and does not result in a large increase in data size. This enables our system to start up and render the geometry immediately. We have shown how hierarchical level-of-detail, parallel prefetching, and hardware-assisted volume rendering can be combined to maintain interactivity in an environment where occlusion culling is not suitable. Finally, we have shown how out-of-core volume rendering can be applied in a distributed manner to improve mesh quality and increase display size.

In the future, we would like to explore more efficient rendering techniques for keeping high image quality on even larger datasets. For example, point-based rendering may be more suitable for interactive rendering at lower levels-of-detail.

References

[ACW*99] ALIAGA D., COHEN J., WILSON A., ZHANG H., ERIKSON C., HOFF K., HUDSON T., STÜRZLINGER W., BAKER E., BASTOS R., WHITTON M., BROOKS F., MANOCHA D.: MMR: An interactive massive model rendering system using geometric and image-based acceleration. *1999 ACM Symposium on Interactive 3D Graphics* (1999), 199–206.

[CCSS05] CALLAHAN S. P., COMBA J. L. D., SHIRLEY P., SILVA C. T.: Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05* (2005), pp. 199–206.

[CFM*04] CIGNONI P., FLORIANI L. D., MAGILLO P., PUPPO E., SCOPIGNO R.: Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics 10*, 1 (2004), 29–45.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L., SILVA C. T.: Hardware-assisted visibility ordering for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295.

[CKS02] CORRÊA W. T., KLOSOWSKI J. T., SILVA C. T.: *iWalk: Interactive Out-Of-Core Rendering of Large Models*. Technical Report TR-653-02, Princeton University, 2002.

[CKS03] CORRÊA W. T., KLOSOWSKI J. T., SILVA C. T.: Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics)* (2003), pp. 1–8.

[Cla76] CLARK J. H.: Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (Oct. 1976), 547–554.

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 1 (2003), 3–15.

[CRMS03] CIGNONI P., ROCCHINI C., MONTANI C., SCOPIGNO R.: External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics 9*, 4 (2003).

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

[DWS*97] DUCHAINEAU M. A., WOLINSKY M., SIGETI D. E., MILLER M. C., ALDRICH C., MINEEV-WEINSTEIN M. B.: ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization '97* (1997), IEEE, pp. 81–88.

[ESC00] EL-SANA J., CHIANG Y.-J.: External memory view-dependent simplification. *Computer Graphics Forum 19*, 3 (Aug. 2000), 139–150.

[ESSS01] EL-SANA J., SOKOLOVSKY N., SILVA C. T.: Integrating occlusion culling with view-dependent rendering. In *IEEE Visualization '01* (Oct. 2001), pp. 371–378.

[FST92] FUNKHOUSER T. A., SÉQUIN C. H., TELLER S. J.: Management of large amounts of data in interactive building walkthroughs. *1992 ACM Symposium on Interactive 3D Graphics 25*, 2 (1992), 11–20.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02* (2002), pp. 693–702.

[LCCK02] LEVEN J., CORSO J., COHEN J. D., KUMAR S.: Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics* (2002), pp. 37–44.

[LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 1997* (1997), pp. 199–208.

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *SIGGRAPH 2000* (2000), pp. 259–262.

[LRC*02] LUEBKE D., REDDY M., COHEN J., VARSHNEY A., WATSON B., HUEBNER R.: *Level of Detail for 3D Graphics*. Morgan-Kaufmann Publishers, 2002.

[ML04] MUSETH K., LOMBEYDA S.: Tetsplat: Real-time rendering and volume clipping of large unstructured tetrahedral meshes. In *IEEE Visualization '04* (2004), pp. 433–440.

[MT03] MORELAND K., THOMPSON D.: From cluster to wall with vtk. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), p. 5.

[Sam90] SAMET H.: *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[SCCB05] SILVA C. T., COMBA J. L. D., CALLAHAN S. P., BERNARDON F. F.: A survey of GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing (RITA) 12*, 2 (2005), 9–29.

[SCESL02] SILVA C., CHIANG Y., EL-SANA J., LINDSTROM P.: Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization '02* (2002). Course Notes for Tutorial 4.

[ST90] SHIRLEY P., TUCHMAN A.: A polygonal approximation to direct scalar volume rendering. *Proc. San Diego Workshop on Volume Visualization 24*, 5 (Nov. 1990), 63–70.

[VM02] VARADHAN G., MANOCHA D.: Out-of-core rendering of massive geometric environments. In *IEEE Visualization '02* (2002), pp. 69–76.

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based ray casting for tetrahedral meshes. In *IEEE Visualization '03* (Oct. 2003), pp. 333–340.