LA-UR-07-5141

*Approved for public release;
distribution is unlimited.*

Title:

A Modular Extensible Visualization System Architecture For Culled Prioritized Data Streaming

Author(s):

J. Ahrens, N. Desai, P. McCormick, K. Martin, J. Woodring

Intended for:

Visualization and Data Analysis 2007, Proceedings of the SPIE - The International Society for Optical Engineering, Volume 6495, Jan. 2007

A Modular, Extensible Visualization System Architecture for Culled, Prioritized Data Streaming

James P. Ahrens, Nehal Desai, Patrick S. McCormick[a], Ken Martin[b] and Jonathan Woodring[c]

[a]Los Alamos National Laboratory, Los Alamos, New Mexico, USA; [b]Kitware, Inc., Clifton Park, New York, USA; [c]The Ohio State University, Columbus, Ohio, USA

ABSTRACT

Massive dataset sizes can make visualization difficult or impossible. One solution to this problem is to divide a dataset into smaller pieces and then stream these pieces through memory, running algorithms on each piece. This paper presents a modular data-flow visualization system architecture for culling and prioritized data streaming. This streaming architecture improves program performance both by discarding pieces of the input dataset that are not required to complete the visualization, and by prioritizing the ones that are. The system supports a wide variety of culling and prioritization techniques, including those based on data value, spatial constraints, and occlusion tests. Prioritization ensures that pieces are processed and displayed progressively based on an estimate of their contribution to the resulting image. Using prioritized ordering, the architecture presents a progressively rendered result in a significantly shorter time than a standard visualization architecture. The design is modular, such that each module in a user-defined data-flow visualization program can cull pieces as well as contribute to the final processing order of pieces. In addition, the design is extensible, providing an interface for the addition of user-defined culling and prioritization techniques to new or existing visualization modules.

1. INTRODUCTION

Scientists use computational simulations and sensors to model, study, and understand complex processes. Visualization systems are used to process and extract meaningful information from the extremely detailed datasets generated by these tools. The massive sizes of these datasets can make processing difficult or even impossible on a desktop computer. One solution to this problem is to divide a massive dataset into smaller pieces on disk and then stream these pieces through memory, running the visualization algorithms on each piece. This technique, called out-of-core processing, assumes that the data is larger than memory and therefore must be incrementally processed. The process of *data streaming* places an emphasis on incremental processing, regardless of where the data is stored. A key aspect of out-of-core and streaming algorithms is that once the data is divided into pieces, both approaches allow decisions to be made about which pieces are operated on and in what order.

The research contribution of this paper is a novel culling and prioritized data-flow streaming visualization architecture that supports a broad range of visualization operations and programs. Culling and prioritization improve a user's visualization experience by producing results faster (by skipping the processing pieces that do not contribute to the final result) and by progressively ordering the display of pieces based on a user-defined prioritization metric.

2. RELATED WORK

2.1. Out-of-core algorithms and systems

Researchers have developed out-of-core visualization algorithms for streamlines on unstructured grids, surface simplification, and isosurfacing.[4, 17, 25] In addition, Vitter presents a general introduction to out-of-core/streaming algorithms.[27] These algorithms typically read the data from disk, execute the algorithm, and write the data back to disk incrementally. The resulting data is then rendered from disk. One downside to this approach is the additional I/O required. Ideally, these algorithms would be incorporated into an out-of-core/streaming infrastructure that avoids intermediate I/O.

The algorithms and systems described above focus primarily on the visualization of scientific data. Other researchers have focused on out-of-core rendering for architectural walkthroughs and the display of massive terrain datasets. There are many techniques for culling and ordering data for rendering. Culling techniques are typically based on the view frustum and object occlusion, while ordering techniques often include view dependent display.[6] The architecture presented in this paper supports culling and ordering of pieces using these techniques. Funkhouser *et al.* present an out-of-core rendering system for interactive walkthroughs.[11] Correa *et al.* describe the iWalk system, an out-of-core rendering system that uses multi-threading to render, compute visibility, and read from disk simultaneously in order to achieve interactive performance.[7] The iWalk system builds an octree representation of the model on disk for faster access. Lindstrom et. al built a terrain system that supports a data indexing scheme for fast access and user-defined error metrics.[16] Out-of-core rendering systems are typically applied to static models and can therefore use a preprocessing optimization step to organize the data for efficient access and rendering. This allows portions of the data to be excluded from processing by identifying their visibility from selected viewpoints. Out-of-core visualization systems, on the other hand, are used to render dynamic data, such as interactive requests for a specific isocontour value; this often makes the optimizations used for static models unworkable.

Some algorithms and systems combine specific culling and prioritization functionality. For example, Livnat and Hansen[18] describe an algorithm for combining view-dependent rendering and efficient isosurface extraction. Chiang et. al describe a system that supports both out-of-core visualization and rendering functionality by combining a parallel out-of-core isosurface extraction algorithm with an out-of-core volume rendering technique for unstructured grids.[3] Gao and Shen present a parallel, view-dependent isosurface extraction algorithm that incorporates occlusion culling.[12] In contrast to the algorithms and systems described above, our work goes beyond implementing a specific combination of culling and prioritization functionality by creating an general-purpose visualization architecture that supports combining culling and prioritization functionality.

2.2. Importance-driven systems

The QSplat rendering system, which uses different size bounding spheres to provide a multi-resolution, graphical representation of an object, has been extended to progressively render data streamed over a network.[21,22] The system orders the streaming of data based on a measure of its importance. This measure incorporates the granularity of the data (i.e. the current level of detail), the projected screen size, and presentation order. Viola, Kanitsar and Gröller present system that modifies the volume rendering process such that user selected objects are visible despite view changes and occlusions.[26] Objects are assigned an importance value before program execution. In comparison to the fixed metrics in the systems described above, the architecture presented allows any algorithm in the visualization pipeline to independently contribute to a dynamic, global metric of the importance of each individual piece to the final result.

2.3. Data-flow based visualization systems

In general terms, our architecture is similar to data-flow based visualization systems such as IBM Data Explorer(DX),[1] AVS[9] and SCIRun.[20] Support for data streaming (i.e. the ability to incrementally process pieces of a dataset), a key building block of our architecture, is not supported by these systems. Moran and Henze present a visualization system called DDV that supports demand driven creation and visualization of derived fields.[19] DDV uses a field data structure to identify data elements that are required by the associated algorithm. These data structure/algorithm pairs are composed into a program as a data flow graph. Childs *et al.* describe the Visit system[5] and introduces the idea of a contract between the modules in a demand-driven data-flow visualization program. Modules can modify the contract (i.e. a data structure – essentially a name to data element map), during execution, to coordinate system features such as the generation of ghost data, to cull data and when to use static or dynamic load balancing. Ahrens *et al.* and Law *et al.*[2,14] present extensions to the Visualization Toolkit (VTK)[23] to support data streaming for structured and unstructured grids. Our work shares the demand-driven data-flow streaming features of DDV, Visit and VTK. Our architecture and the Visit architecture share a similar general approach for culling data. Our architecture is distinct from DDV, Visit and the existing VTK streaming architecture by offering the prioritization of streamed pieces. It also provides a single framework for incorporating culling and prioritization methods; including spatial culling, data culling, occlusion culling and view dependent prioritized rendering all in the same general-purpose visualization architecture.

3. DESIGN

3.1. Overview of a demand driven data-flow streaming based architecture

Our design builds upon a demand driven data-flow streaming architecture. In such an architecture, a program is created by composing functional modules together into a data-flow pipeline. The program is executed in a demand-driven manner, that is, data requests are propagated up through the modules in the pipeline to the data source, and the computed results flow back down the pipeline to create the final result. Data streaming requires:[*]

- *The data is separable.* This means the dataset can be broken into pieces. Ideally, each piece is coherent in geometry, topology, and/or data structure. The modules in this architecture must be able to correctly process pieces of data.
- *The data is mappable.* This means we must be able to determine what portion of the input data is required to generate a given portion of the output.
- *The results be output invariant.* That is, the results should be the same, independent of the number of pieces, and independent of the execution mode (i.e., single- or multi-process). This means the proper handling of pieces that may overlap on their boundaries.

To implement these requirements, a demand-driven preprocessing step is run when data is requested. This step gathers and computes information about the input and output data of each module in the pipeline. A wide range of information is returned. Typically, properties such as the native data type, number of scalar components, and the scalar and spatial ranges are computed. After the preprocessing step completes, a demand-driven execution step is run. Using the information gathered during the preprocessing step, data is instantiated and then the modules are executed.

When using streaming, the demand-driven preprocessing step is run for each piece. The piece extents and boundary data are computed for each piece by the preprocessing steps. For structured data, the piece data is represented by its spatial extents. For unstructured data, the data is represented as piece number M of N possible pieces. Modules that process structured data and output unstructured data use a partitioning algorithm to divide the structured data into the requested number of pieces. Ghost cells are used to handle cross-piece communication and the number of layers of ghost cells is computed as part of the preprocessing step. Figure 1 presents a collection of pseudo-code descriptions of how a standard demand-driven data-flow architecture is modified to cull and prioritize pieces.

For example, suppose we have a program with an image reader, contour, and render modules. If the number of pieces is set to four within the renderer, then four preprocessing steps will be run. During each step the renderer propagates a request for an unstructured piece, in the form of a piece number, up the pipeline. The contour module, which receives structured data and outputs unstructured data, converts from the given piece number to a specific spatial extent (e.g. one quadrant of the output image) using a data partitioning algorithm. During the associated execution step, the contour module receives a piece from the reader, and processes the data creating contours within the assigned extent. This process continues until the remaining three pieces have been completed.

In order to stream data from disk, the reader module needs to be able to read and return individual pieces. One option is to build a custom reader that can extract each piece from a single file. This option has the advantage that the number of pieces can be changed at runtime. The main disadvantage is that writing such a reader is complicated, especially for complex data structures such as unstructured grids. [†]

[*]Note that these requirements are met by the streaming architecture of the Visualization Toolkit. Most imaging, visualization and rendering modules in VTK currently can stream data as part of a visualization pipeline. In this paper, we extend the VTK streaming architecture to support culling and prioritization. See[2,14] for more details on VTK's streaming architecture.

[†]Parallel data partitioning algorithms provide similar functionality, splitting up a dataset stored in a single memory unit into multiple memory units for distributed processing on multiple processors. Modifying existing data partitioning algorithms to solve the file splitting problem we described above is an avenue we will explore in future work.

```
for piece = 1 to number_of_pieces
    preprocessing_step()
    execution_step()

Streaming
```

```
for piece = 1 to number_of_pieces
    preprocessing_step()
    cull_value = culling_step()
    if (cull_value > 0)
        execution_step()

With culling
```

```
for piece = 1 to number_of_pieces
    preprocessing_step()
    cull_pr_value = cull_and_pr_step()
    if (cull_pr_value > 0)
        push(prioritized_list,
                cull_pr_value, piece)

while length(prioritized_list) > 0
    piece = pop(prioritized_list)
    execution_step()

With culling and prioritization
```

Figure 1. Evolution of psuedocode showing a streaming architecture that includes culling and prioritization

The option used in this paper, is to read the data and save each piece to its own file before the visualization process starts. In addition to storing the piece data, the data range and spatial extents of the piece are computed and stored as meta-data associated with each file. This step avoids the need to compute the range during the execution of the pipeline. This option has the advantage of being simpler to implement, but has the disadvantage of having to select a specific number of pieces before the program is run. Another possible disadvantage is that I/O performance can significantly degrade when reading and writing many small files. This process needs to occur only once, before the visualization pipeline is run.

3.2. Adding culling and prioritization to the demand driven data-flow streaming based architecture

For each piece, once the preprocessing step is complete, all the information necessary to run the execution step is known. The key idea in this paper is that the preprocessing information can be used throughout the visualization pipeline to cull and order pieces. While culling and prioritizing are unified in this architecture we will first consider culling.

3.3. Adding a culling step

In order to cull pieces an additional step is added that runs after the preprocessing step. The culling step uses the preprocessing information to set a value to zero if a piece is to be discarded and one if it is not (Our particular use of zero and one as culling flags will be discussed in Section 3.4.) Each module can provide its own unique culling method. For example, a clipping module can cull pieces whose spatial extent does not intersect the clip area, or a rendering module can cull pieces not within in the view frustum.

As with the preprocessing step, the culling step is demand-driven. A culling request is propagated up through the modules in the pipeline to the data source and culling information flows back down the pipeline. Specifically, for each module, an independent culling value is calculated. If any module in the pipeline chooses to discard a piece, all modules in the pipeline will discard it as well. This is accomplished by propagating a zero culling value down the pipeline. All modules that do not define a specific culling method use a default method that does not affect the culling process. This culling method always returns a value of one unless it is propagating a zero value down the pipeline. The pseudocode in the center box of Figure 1 shows the addition of a culling step.

Culling methods are based on the spatial location or data values of a piece. In the list below, we describe a collection of culling steps that are spatial or value based. It is important to note that this list is not meant be exhaustive, rather it simply illustrates the types of culling operations that are possible.

1. **Based on spatial location**

    (a) *Spatial Clipping*: A clipping module clips a dataset with a surface, retaining any cells that fall on the retaining side of the surface. In this case, the module's culling method removes pieces whose spatial bounds fall outside of the retained region.

(b) *Cutting*: A cutting module cuts through a dataset with a surface and then displays the interpolated values on the surface. The culling method discards pieces that do not intersect with the cutting surface.

(c) *Probing*: Probing samples a dataset with a set of points (possibly a single point, points along a line, on a plane, or in a volume). The resulting points can then be graphed or visualized. For this case, the culling method will remove pieces that do not contain any probe points.

(d) *Frustum Culling*: The frustum culling method removes pieces that are not within the viewing frustum. This method is part of the rendering module.

(e) *Occlusion Culling*: The occlusion culling method discards pieces that are not visible because they are occluded by other objects in the scene. This step makes use of OpenGL's hardware-accelerated, occlusion culling extension.[15] To run the occlusion test, writes to the color and depth buffers are disabled. Next, the bounding box of the piece is rendered. If the bounding box is not visible, none of the geometry representative of the piece will be visible, and therefore the piece is culled. After the test completes, the color and depth buffers are re-enabled.

2. **Based on data value**

(a) *Contouring*: A contouring/isosurfacing module creates a boundary line or surface that follows one or more constant scalar values known as the contour values. The culling method compares the data range of the values in a piece with the contour values to see if they intersect. If there are no intersections, the piece is discarded. The data range of a piece is stored on disk as meta-data and is available as a result of the preprocessing step.

(b) *Thresholding*: A thresholding module selects the cells within a dataset based on whether a cell's data value falls within a given range. If the data range for a piece falls outside of the threshold range, the piece is culled.

3.4. Adding a prioritization step

Our architecture also allows for the prioritization of pieces as a simple extension of the culling process. Prioritization is useful to provide early feedback to the user about a visualization before it completes. Users can make decisions, such as stopping a visualization and changing the viewpoint, or changing a visualization parameter immediately instead of having to wait for completion. This is especially important for distance applications where visualizations may take significantly longer to complete due to data transfer times over a wide-area network.

Prioritization of pieces is implemented by extending the notion of culling from a boolean value of zero or one to a continuous value between zero and one. A culling value of zero is still used to indicate that a piece should be discarded. However, values greater than zero indicate the priority of a piece. In this case, pieces should be processed in the order given by this numeric value; with higher values indicating higher priority. For example, after the preprocessing step is complete, the spatial location of a piece is known. This location information can be used to assign a view dependent priority P, where $0.0 < P \leq 1.0$, such that pieces are processed in order based on their distance from the viewer.

Priorities are multiplied together to provide a weighting of the priorities assigned by different modules. For example, the priority of a piece can be based on an estimate of the total number of polygons combined with a view dependence prioritization. Pieces that are estimated to generate a large number of polygons, and are closest to the viewer, would be processed first.

The default culling method for modules continues to work with prioritization as it assigns a value of one to all pieces. With a priority of one, by default, a module will not change the current priority assigned by other modules in the pipeline (i.e. multiplying the priority by one does not change the priority). Note that multiplying by zero propagates the message to cull a piece. The psuedocode in the box on the right hand side of Figure 1 shows the addition of a prioritization step to the algorithm.

The prioritization of pieces can be based on spatial location or the data values within the piece. In the list below, we describe a collection of prioritization steps that are spatial or value based. Once again, this list is not meant to be exhaustive, but instead a illustrative of the types of operations that can be used to assign priorities.

1. **Based on spatial location**
   (a) *View Dependent Ordering*: View dependent ordering prioritizes pieces based on their distance from the camera. In this situation, pieces located closer to the camera receive a higher priority.
   (b) *Cutting*: The culling algorithm presented in Section 3.3 can be extended to also provide a priority. The culling algorithm simply checks if the bounding box of the piece is intersected by the cutting surface. A prioritized algorithm prioritizes based on how much the surface intersects the bounding box.
2. **Based on data value**
   (a) *Color by Scalar Value*: A scalar color mapping module maps values to colors. For example, the opacity channel of the mapping could be used to order pieces (i.e. a measure of piece opacity could be calculated based on a histogram of the data values and the opacity mapping) and the estimate of the most opaque pieces could be rendered first. Similarly, a histogram could be used to display the piece values from the maximum to minimum scalar color mapping (e.g. from warm to cold temperature values).

The design of our architecture has similarities to Livnat and Hansen's work[18] on view dependent isosurface extraction. Our architecture differs because it supports the creation of programs that can include additional functionality beyond view dependence, hardware-accelerated occlusion culling and isosurface extraction. Specifically our architecture supports a broad range of culling and prioritization tasks; for example, user-defined programs using our architecture can include other visualization operations such as clipping, cutting and probing and combine alternative prioritization metrics such as coloring by scalar value. Our architecture also has similarities to Klosowski and Silva's work on the prioritized-layered projection algorithm (PLP).[13] In their work, a geometric dataset is partitioned into cells and a priority, based on how likely a cell is to be visible, is assigned to the cell. This priority is used to order the rendering of cells. As the rendering of cells progresses, hardware-accelerated occlusion culling is used to determine the visibility status of the geometry. Our architecture differs from PLP because it incorporates rendering prioritization methods with visualization and rendering culling algorithms in a general-purpose visualization architecture.

4. IMPLEMENTATION

Our culling and prioritization streaming architecture is implemented as part of the Visualization Toolkit[23] library and released as part of VTK version 5.0. To support these capabilities new meta-information keys were added to the key-value based architecture VTK uses for handling pipeline operations. The first of these new keys was added to create a new pass of the pipeline called REQUEST_UPDATE_EXTENT_INFORMATION. This pass occurs after requesting the overarching meta-information about the data and after an update extent (piece) has been specified. Its purpose is to gather meta-information specifically about a requested piece of data. One of the values that propagates downstream on this pass is the FIELD_RANGE key. This key provides the scalar range of the data arrays at each step in the visualization pipeline prior to actually reading or processing them. It is worth noting that these scalar ranges may change as the data passes from filter to filter. Another new key propagated during this pass is the priority of the piece specified using the PRIORITY key.

In practice, when data is written out into pieces using VTK's XML file format, the files will automatically include meta-information about the ranges of each piece's data arrays. When opening a data file the number of pieces is read in first. Then during the next phase of pipeline processing, information about specific pieces can be requested. This information is read from the data file and then adjusted and manipulated by each filter in the pipeline. For example, the contour filter in VTK considers the FIELD_RANGE of the data array it is contouring. If the contour value is not included in the range then the priority for that piece is set to zero. Even if the contour value does intersect the range a subsequent filter downstream may lower or zero out the priority based on a different metric. In this manner priorities and culling are performed not independently by each filter, as most out of core techniques are limited to, but by all the filters negotiating together. While only a few implementation keys have been discussed, the key-value nature of the architecture supports easily adding new meta-information that can be taken into account in determining priority and culling without any changes to the core code base.

Although the extensions to cull and prioritize data appear straightforward (i.e. as shown in Figure 1) figuring out how to add this functionality to the VTK architecture with a minimal impact was not. For a full-featured and complex architecture like VTK it is of great benefit to extend the system in significant way (i.e. allowing VTK programs to cull data and progressively prioritize the display of results) without adding significant additional complexity.

5. RESULTS

In this section, we report results of using our architecture for data culling and prioritized streaming for different types of input data: structured, out-of-core and unstructured datasets. In each case, the same basic methodology is applied to each type of data. The underlying architecture is not modified only how the input dataset is subdivided into pieces is changed. All tests were run on a Dell Precision 450 with a 2.8 GHz Intel Xeon processor, 512 KB of cache, 2 GB of RAM, and an NVIDIA Quadro FX 1000 graphics card.

5.1. Structured Dataset

The dataset used in this section is a simulation result from the Terascale Supernova Initiative (TSI).[24] The TSI team is modeling the mechanisms responsible for driving core collapse supernovae. The simulation runs on a three-dimensional structured grid with dimensions of $320 \times 320 \times 320$ and generates multiple variables including entropy, density and energy. The supernova data is used to illustrate how the architecture performs on a reasonably large data set that a scientist would process on their desktop computer. We did not use a dataset that exceeded the memory of the test machine because we wanted to compare the execution times with the standard VTK architecture.

5.1.1. Data culling

The first test reports the performance improvement achieved when using the new streaming architecture for data culling. A VTK program was created to read the entropy and density fields, contour the entropy field with a contour value of 0.07, color the contour by density, clip the resulting contour geometry, and render this result. Table 5.1.1 shows the total program time in seconds as well as individual times to read, contour, clip, and render the data using the standard architecture versus using the streaming architecture described in this paper. This visualization pipeline makes use of the culling method in the contour module (Section 3.3, list 2, item a), the clipping module (Section 3.3, list 1, item a) and the render module (Section 3.3, list 1, item e). In addition, view dependent ordering is used (Section 3.4, list 1, item a).

	Read Time	Contour Time	Clip Time	Render Time	Total Time
Stnd. Arch.	21.65	28.55	1.66	0.23	52.29
Strm. Arch.	5.41	7.21	1.18	0.59	16.25
Spdup	4.00	3.95	1.41	0.39	3.22

Table 1. Performance in seconds on a 320^3 dataset with the standard and streaming architectures. For the streaming architecture, the dataset is split into 512 pieces.

Number Of Pieces	Read Time	Contour Time	Clip Time	Render Time	Total Time
32	17.61	23.26	1.33	0.34	42.85
64	12.56	16.65	1.37	0.35	31.28
128	10.98	14.54	1.40	0.47	28.09
256	7.83	10.38	1.23	0.50	21.07
512	5.41	7.21	1.18	0.59	16.25
1024	4.73	6.38	1.26	0.86	16.73

Table 2. Performance in seconds with the streaming architecture when varying the number of pieces. The visualized result is the same regardless the number of pieces used and is shown in the upper left corner of Figure 2.

Table 1 shows that total program performance is improved by a factor of three. The modules are able to cull away 395 out of 512 pieces. The contour culling step culls 336 pieces, the clipping culling step culls an additional 48 pieces and the occlusion culling step culls an additional 11 pieces; leaving 117 pieces out of 512 that are read from disk, contoured, clipped and rendered. Notice that rendering time increases since we are rendering many intermediate images (i.e. each time a piece is generated it is composited to the final result) but the standard architecture renders only once. The visualized result is shown in the upper left corner of Figure 2 and a sequence of intermediate results is shown in Figure 4.

It is important to recognize that the effectiveness of each culling step depends on how many pieces are culled by previous modules. For example, only a small number of pieces are occlusion culled in the previous example. This is because the contour and clipping module were very effective at culling pieces. We ran a set of tests that use the same data but vary the clip plane value such that it is placed at equidistant points along its normal from one side of the dataset's bounding box to the other side. The streaming architecture consistently improves performance over the entire range of clip values, but for each clip value different culling modules are more effective than the others. For example, when the dataset is completely clipped away the clipping module culls all the pieces. When none of the dataset is clipped, the occlusion culling module culls many pieces on the far-side of the supernova contour. This result highlights the benefit of having multiple modules in a visualization pipeline independently culling pieces.

The second test reports the performance difference when using the streaming architecture for data culling with a varying number of pieces. The same program used to generate the results in Table 1 is used to generate the results in all figures and tables in this section.

Notice in Table 2 that as the number of pieces increases from 32 to 512 pieces, the total execution time decreases. We expect that increasing the number of pieces tightens the range of the spatial extents and range of data values in each piece allowing for more culling to occur. For the supernova dataset, we measured the mean size of the scalar range and the spatial bounding volume for 32 to 1024 pieces. As the number of pieces increases by a factor of two the average mean size of the scalar range and bounding volume decreases by approximately a factor of two confirming our expectations. At some point, increasing the number of pieces will not provide additional performance improvements because of the overhead associated with processing additional pieces. For example, the total execution time increases at 1024 or more pieces due to this overhead.

In the streaming architecture the total program time is proportional to the number of pieces actually processed. Many other modules described in Section 3.3, such as the cutting (Section 3.3, list 1, item b) and probe modules (Section 3.3, list 1, item c), will also significantly reduce the number of pieces that need to be processed. For example, consider a volume consisting of n pieces on each side for a total of n^3 pieces. After the culling step of the cutting module runs there will be approximately n^2 pieces left: the pieces that intersect the cut plane. If a line probe module processes the data volume, the culling step would then reduce the number of pieces from n^3 to approximately n; the pieces that intersect the probe line.

5.1.2. Prioritization

The next test examines the use of the streaming architecture to display results progressively. This is accomplished by displaying an intermediate image after each piece is processed. An image-accuracy metric is used to measure the quality of these intermediate images. This metric counts exact pixel matches between each intermediate image and the final result image created by the standard architecture. This result is then divided by a count of exact pixel matches between the starting image (a blank image composed of all background pixels) and the final result image to scale the metric between 0 and 100. A value of 0 for the metric means that there are no non-background pixel matches between the test image and final image and a value of 100 means that the images are an exact match. Values in between describe a percentage of the number of non-background pixels that are an exact match. Figure 3 plots a value for each intermediate result where the x axis is the elapsed time for the image generated and the y axis is the image accuracy. This is done for six different camera views with a prioritized ordering shown in dark blue and a random ordering shown in pink. Figure 2 shows the rendered results for the six camera views. A prioritized ordering provides an increase in image accuracy over a random ordering for four of the six views. The two views that have similar behavior to the random ordering are for azimuth values of 240

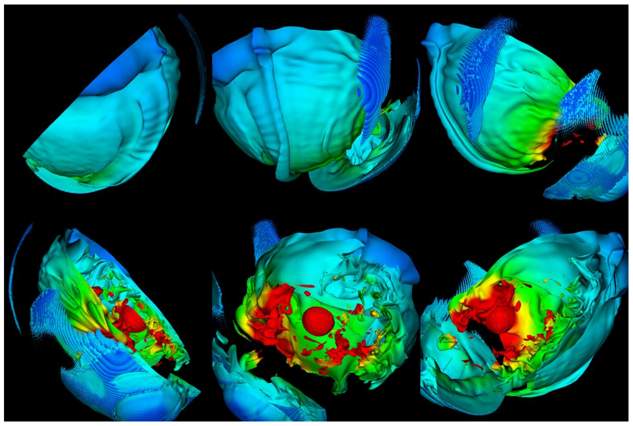

Figure 2. Six views of the supernova from left to right, top to bottom the camera azimuth values are 0,60,120,180,240 and 300 degrees.

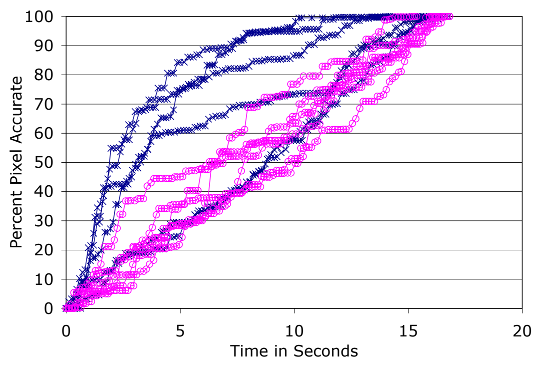

Figure 3. Image accuracy versus time for six views shown in Figure 2 with prioritized ordering (in dark blue with X icons) and a random ordering (in pink with O icons).

and 300 degrees. These views do not have much depth complexity and therefore a view dependent ordering does not provide an improvement.

For the view shown in Figure 2 with an azimuth value of 0, image accuracy increases very quickly. This is because with a view dependent ordering the surface of the supernova closest to viewer is computed first and immediately increases image accuracy. This surface also occludes additional details in other pieces. Figure 4 shows results that are 25%, 50% and 75% accurate. These views are computed in 2.4%, 3.8% and 7.7% of the time it takes the standard architecture to generate the final image.

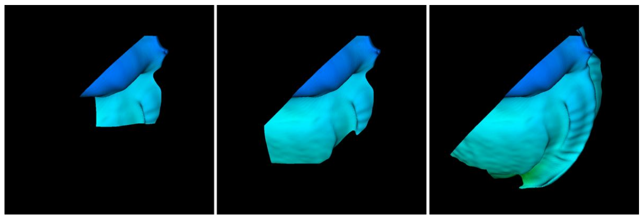

Figure 4. Visualized supernova results that are 25%, 50% and 75% accurate.

5.1.3. Overhead Costs

It is important to consider the total cost of the streaming architecture as compared to that of the standard architecture. There are three sources of overhead:

1. *The cost of splitting the files into multiple pieces before program execution.* This occurs once before the visualization program is run. It takes three minutes to split the file into 512 piece files. We consider the initial cost of a few minutes reasonable since many visualizations are typically run on one dataset and significant performance improvements result from this file organization.

2. *The cost of running the culling and prioritization step.* For the test program the overhead of the culling and prioritization steps is only 6.5% of the total execution time. This time is primarily due to reading the scalar range and spatial extent meta-data from disk.

3. *The cost of running multiple pieces through the pipeline.* We ran a test to compute the cost of running multiple pieces through the pipeline. Specifically, the worst case for the streaming architecture occurs when every culling and prioritization step was run and every piece was processed. The total time is only increased by about four seconds, or a 7% increase over the total time for the standard architecture. In practice, we expect the need to process all pieces is rare and therefore the overhead costs will be amortized by the performance gains from culling pieces.

5.2. Out-of-core Dataset

In this section we test the ability of our architecture to handle out-of-core data. We used the Visible Woman data set from the National Library of Medicine to test our system. In its original form, the Visible Woman dataset is 864 MB, but to stress the architecture we upcast the dataset from 16 bit to 64 bit integers, effectively quadrupling the size of the dataset to 3.375 GB. At this size, standard VTK is unable to the load the data into core memory. It is important to note that the input dataset is structured and the techniques used in the previous section were used to partition the data.

In our out-of-core test, we isosurface the data, clip the model in half and render the data for two common isosurface values 600.5 (skin) and 1224.5(bone). Table 3 shows the performance results and Figure 5 the resulting visualization of the skin isosurface.

Number of Pieces	Time (s) for 600.5	Time (s) for 1224.5
128	208.38	149.03
256	192.32	141.81
512	172.38	118.82
1024	119.79	102.90
2048	112.94	97.36
4096	104.65	94.81
8192	61.57	83.26
16384	103.72	76.73
32768	125.86	85.52

Table 3. Out-of-core timings for creation of the Visible Woman skin and bone isosurfaces with our architecture.

Number of Pieces	Pieces Culled	Total (s) Time
1	0	70.56
8	2	28.22
16	6	23.40
32	17	16.64
64	40	15.36
128	89	14.58
256	193	12.73
512	409	12.90
1024	861	13.11

Table 4. Timings of the atmospheric simulation data with density isosurface value of (1.01).

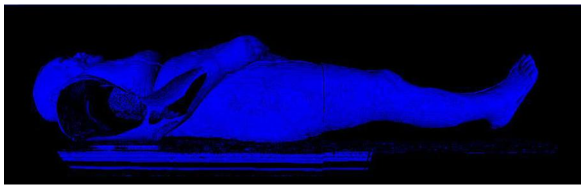
Figure 5. Visible Woman skin isosurface (600.5).

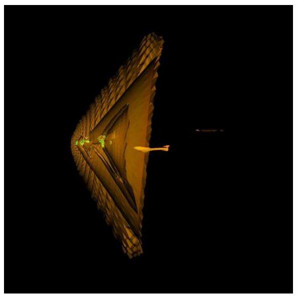
Figure 6. The shock cone that results from the space shuttle traveling at supersonic speeds. The isosurface is colored by stagnation energy.

As the piece count increases, program performance improves. The speedup results from a reduction in read time, rendering time, and isocontouring time. Culling does incur an overhead from reading metadata and running culling operations. There is a point of diminishing returns where the overhead increases significantly that additional speedup does not take place. This occurs at 8192 pieces in the skin isosurface and 16384 pieces in the bone isosurface. Increasing the piece count beyond that point does not create any additional speedup.

5.3. Unstructured Dataset

In this section we apply our architecture to the problem of efficiently processing an unstructured data. The unstructured dataset results from a simulation of a space shuttle traveling through the atmosphere. The resulting 2.7M tetrahedral mesh is partitioned into pieces using a recursive spatial bisection method. Table 4 shows the total time necessary to process (read, isosurface and render) the data.

6. CONCLUSIONS AND FUTURE WORK

This paper presented a general purpose visualization architecture that incorporates advanced culling and prioritization features based on data value and spatial location. This architecture also enables advanced rendering features such as view dependent ordering and occlusion culling. Significant performance improvements have been demonstrated for real-world visualization programs. This architecture provides a framework for researchers to incorporate additional advanced streaming features as they are presented in the literature.

There are a number of directions for future research with this architecture. The first is the incorporation and characterization of performance of new culling and prioritization steps. In this work we've focused primarily on optimizing performance. Prioritization steps that focus on content (i.e. getting the most relevant information to the user quickly) are also of interest. The second direction is the incorporation of more complex data structures into the architecture. Currently, work is in progress on incorporating overset and adaptive mesh refinement grids into VTK. This includes having these grids leverage the streaming architecture. Once these grids are fully integrated, additional efficiencies can be realized by adding culling and prioritization steps that are aware of the properties of these new data structures. For example, with an octree data structure, significant spatial portions of the grid can be culled quickly by culling data elements at the top of the octree.

REFERENCES

1. G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *Processings of Visualization '95*, pages 263–270. IEEE Computer Society Press, 1995.
2. James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, and Michael Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, 2001.
3. Yi-Jen Chiang, Ricardo Farias, Claudio T. Silva, and Bin Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *PVG '01: Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 59–66. IEEE Press, 2001.
4. Yi-Jen Chiang, Claudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Visualization 1998*, pages 167–174. IEEE Computer Society Press, 1998.
5. Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *IEEE Visualization 2005*, pages 191–198. IEEE Computer Society, 2005.
6. Daniel Cohen-Or, Yiorgos Chrysanthou, Claudio Silva, and Fredo Durand. A survey of visibility for walk-through applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.
7. W. Correa, J. Klosowski, and C. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *PVG '03: Proceedings of the IEEE 2003 Symposium on Parallel and Large-data Visualization and Graphics*. IEEE Press, 2003.
8. Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Visualization 1997*, pages 235–244. IEEE Computer Society Press, 1997.
9. C. Upson et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

10. Ricardo Farias and Claudio T. Silva. Out-of-core rendering of large, unstructured grids. *IEEE Computer Graphics and Applications*, 21(4):42–50, 2001.
11. Thomas Funkhouser, Carlo Sequin, and Seth Teller. Management of large amounts of data in interactive building walkthoroughs. In *1992 SIGGRAPH Symposium on Interactive 3D Graphics*, pages 11–20, 1992.
12. Jinzhu Gao and Han-Wei Shen. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *PVG '01: Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, pages 67–74. IEEE Press, 2001.
13. James T. Klosowski and Claudio Silva. Efficient conservative visibility culling using prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4), 2001.
14. C. C. Law, W.J. Schroeder, K.M. Martin, and J. Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *IEEE Visualization 1999*. IEEE Computer Society Press, October 1999.
15. Eric Lengyel. *The OpenGL Extensions Guide*. Charles River Media, Inc., 2003.
16. P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *IEEE Visualization 2001*, pages 363–370. IEEE Computer Society, 2001.
17. Peter Lindstrom. Out-of-core simplification of large polygonal models. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 259–262. ACM Press/Addison-Wesley Publishing Co., 2000.
18. Yarden Livnat and Charles Hansen. View dependent isosurface extraction. In *IEEE Visualization 1998*, pages 175–180. IEEE Computer Society, October 1998.
19. P. Moran and C. Henze. Large data visualization with demand-driven calculation. In *IEEE Visualization 1999*, pages 27–33. IEEE Computer Society, October 1999.
20. S. G. Parker, D.M. Weinstein, and C. R. Johnson. The SCIRun computational steering software system. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 1–40. Birkhauser Press, 1997.
21. Szymon Rusinkiewicz and Marc Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.
22. Szymon Rusinkiewicz and Marc Levoy. Streaming Qsplat: a viewer for networked visualization of large, dense models. In *SI3D '01: Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 63–68. ACM Press, 2001.
23. W.J. Schroeder, K.M. Martin, and W.E. Lorensen. *The Visualization Toolkit - An Object Oriented Approach to 3D Graphics*. Prentice Hall, 1996.
24. Anna Turnage. Modeling supernovae: Braving a bold new frontier. *IEEE Computer Graphics and Applications*, 23(6):6–11, 2003.
25. Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, 1997.
26. Ivan Viola, Armin Kanitsar, and Meister Eduard Gröller. Importance-driven volume rendering. In *IEEE Visualization 2004*, pages 139–146. IEEE Computer Society, 2004.
27. Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.