Bandwidth-efficient Hardware-Based Volume Rendering for Large Unstructured Meshes

Thierry Carrard
CEA/DIF
DSSI
BP 12
France, 91680 Bruyères-le-Châtel
thierry.carrard@cea.fr

Manuel Juliachs
Laboratoire PRiSM
Université de Versailles-Saint-Quentin
45 avenue des Etats-Unis
France, 78035 Versailles
mju@prism.uvsq.fr

ABSTRACT

Recent advances in graphics processor architecture and capabilities have made the development of fast and efficient unstructured volume rendering methods possible. These techniques can be classified into two roughly delimited categories: cell projection based methods and GPU raycasting algorithms. However, both approaches are subject to limitations, respectively due to the main memory-to-GPU bandwidth for the former and due to the GPU per-fragment computation speed and memory size for the latter. These potential bottlenecks can be particularly limiting for large-size datasets, such as the ones produced by large-scale numerical simulation. In this work, we describe an enhancement to the cell-projection rendering method, allowing us to specify each tetrahedron with only 4 vertices and their associated data. By using a point sprite primitive, instead of a set of 4 triangles, we significantly reduce the amount of data transferred from the main memory through the graphics port for each frame rendered. We evaluate the impact of the different rendering stages of our method on the overall frame rate.

Keywords

unstructured meshes, volume visualization, GPU-based rendering.

1. INTRODUCTION

Numerical simulations of unsteady physical phenomena yield datasets comporting a very large number of elements. We are interested in such datasets, usually sharing the following characteristics:

- a very large number of elements (typically ranging from 10^6 to 10^8 elements),
- unstructured meshes, with tetrahedra, hexahedra or other types of cells,
- a large number of different time steps,
- a high dynamic range, both in time and space.

Scientific visualization is a way to gain insight into the simulated phenomena. However, efficiently visualizing such datasets requires high performance techniques and methods. Volume rendering of unstructured datasets is an example of such an advanced visualization technique. Recently, performance and functionalities of commodity graphics processors have reached a point enabling the implementation of complex volume rendering algorithms, dramatically accelerating their performance with respect to previous software implementations. Current graphics processors (or GPUs) are able to process several hundred millions of vertices per second and several billions fragments per second [Nvi04a], allowing relatively complex user-defined programs to be applied to each processed vertex and fragment.

Taking advantage of this dramatic performance increase, several GPU-based unstructured volume rendering methods have been developed in the past years, allowing to render approximatively between 500k and 1 million tetrahedra per second, processing only relatively modest-sized datasets. However, given the current size of the datasets routinely produced by numerical simulations, GPU-based volume rendering methods should be able to render at least 10 million elements per second to meet the visualization needs of computational scientists.

The projected tetrahedra (PT) technique [Shi90] is a well-known hardware-based unstructured cell-projection volume rendering method. It requires cells to be rendered according to a correct visibility order (either front-to-back or back-to-front), which is typically determined on the CPU, prior to rendering, using a sorting algorithm. Volume cells are decomposed on the CPU into a set of transparent triangles, which in turn are sent to the graphics card and blended into the framebuffer.

Recently, several enhancements of this method have been proposed, taking advantage of the programmability features afforded by modern GPUs. These methods generally allowed to bypass the CPU tetrahedron decomposition phase, resulting in an additional performance gain. However, most of these methods require that each tetrahedron be transmitted as a set of four triangles, which can result in a very high bandwidth consumption between the main memory and the graphics card. The ability to specify volume primitives with existing graphics APIs, as suggested by King *et al.* [Kin00a] would allow to go beyond this restriction.

In this work, we present a GPU-based volume rendering method, building upon previously developed approaches. First, we describe related work in the field of GPU-based unstructured volume. In the following section, we describe in detail our point sprite-based GPU volume rendering method, which optimizes the bandwidth usage by reducing the amount of data sent for each tetrahedron. Then, we present an adjacency search method, partly executed on the GPU. After that, we present some experimental results and a comparison with previously published approaches. Finally, we discuss some limitations of our implementation.

2. RELATED WORK

The scope of this section is hardware-accelerated volume rendering. We classify the different approaches into object-space or image-space rendering, the OpenGL rendering pipeline being an example of such the former whereas raycasting is typical of the latter.

In order to compute a correct image, graphics rendering usually requires to depth-sort primitives according to their viewpoint distance. Surface rendering generally only requires to find to nearest primitive (except if transparent). However, volume rendering requires to find all the primitives intersected by a given view ray (except if opaque objects allowing early termination are encountered). Depending on the optical model used [Max95a], this requires a visibility ordering of the primitive set intersecting any given ray. This visibility ordering can be done either in objet-space or in image-space.

In unstructured volume rendering, object-space methods first sort primitives according to the distance to the viewpoint, then render each primitive individually in the visibility order, accumulating their contributions into the frame buffer. Image-space methods process each view ray sequentially. For a given ray, all the intersecting primitives are determined, ray segments are computed and sorted into the correct visibility order and finally accumulated to determine the final pixel color.

Shirley *et al.* [Shi90a] developed the first hardware-accelerated unstructured volume rendering method. Their projected tetrahedra (or PT) method used the alpha-blending capabilities of the then existing graphics boards to accelerate the rendering of tetrahedral cells. In this method, each tetrahedron is projected on the view plane and decomposed into a set of (up to four) non-overlapping triangles, according to a projection class depending on the point of view, with a thick vertex at the longest ray segment-tetrahedron intersection. The triangles are then transmitted to the graphics accelerator and rendered into the frame buffer using alpha-blending to implement transparency. The thick vertex color and opacity are determined by a transfer function and hardware linear interpolation is used to compute color and opacity at each rasterized fragment. However, opacity usually does not vary linearly across a tetrahedron, resulting in approximations.

Wylie *et al.* [Wyl02a] developed a GPU-based implementation of the projected tetrahedra method. They developed a vertex program algorithm which executed the triangle set determination step of the original PT method. As a vertex program performs the same computations on every vertex, they used a fixed topology graph, mapped to each tetrahedra and corresponding to a triangle fan primitive. By using a look-up table, their algorithm then determines the correct projection class, generating zero-area triangles in certain projection cases. Using a triangle fan allowed to transmit only the data relevant to the four tetrahedral vertices, allowing to render a 1000k tetrahedra mesh at 500k tetrahedra per second with a GeForce 4 GPU. However, they used an approximation to compute the thick vertex color.

Weiler *et al.* [Wei02a] developed a ray-casting based rendering method of individual tetrahedra on the GPU, performing projection in a view-independent way. They used a vertex program to compute the ray-segment exit intersection parameter with each face plane at each vertex. The intersection parameter and vertex scalar values are then linearly interpolated by the graphics hardware to provide at each fragment the coordinates addressing a pre-computed volume integral texture. They were able to render a 220k mesh at 480k tetrahedra/s, on an nVidia GeForce 4. However, computing ray intersections per-vertex instead of at each fragment (due to GPU limitations) resulted in artifacts, especially along tetrahedra edges.

More recently, Kraus *et al.* [Kra04a] reviewed the major causes of artifacts in the PT method and related algorithms. They identified incorrect ray segment length perspective interpolation as being one of the main causes of artifacts and implemented the correct interpolation method, using vertex and fragment processing programs. They also identified linear mapping between ray length and the third coordinate of the pre-integrated transfer function texture as a source of artifacts, due to an insufficiently accurate sampling of the volume integral for small ray lengths. Using a logarithmic mapping allowed them to reduce the sampling interval for small ray lengths, eliminating edge artifacts due to the use of linear mapping. Coupling this to the use of floating-point alpha-blending virtually eliminated all major sources of rendering errors.

The previously described methods require an object-space visibility sorting. Several object-space sorting methods have been developed, the best-known being the adjacency graph sorting method MPVO (Meshed Polyhedra Ordering Visibility) [Wil92a], using either a Depth-First Search or Breadth-First Search algorithm. Cook *et al.* [Coo04a] improved MPVO, allowing it to generate an image-space correct visibility ordering. For each different graph connected component, their method executes the MPVO topology ordering. Then, it rasterizes every boundary face on the CPU, storing the coordinates of every boundary face fragment into an A-buffer. This allows to define new adjacency relationships, between two consecutive fragments in a pixel list, belonging to the boundary faces of two different cells. These adjacency relationships are then used to extend the MPVO adjacency graph. The authors showed that a depth-first search of this extended graph generates an image-space correct visibility ordering. Adjacency graph sorting methods are typically executed sequentially on the CPU prior to rendering, which can be costly, especially for large datasets. Reducing this cost might increase the overall rendering performance.

Weiler *et al.* [Wei03a] implemented ray-casting of a convex tetrahedral mesh entirely on the GPU, by using fragment processing programs. Using floating-point textures, they were able to store the whole mesh (vertices, face normals and connectivity) in graphics memory, after convexification during a pre-processing phase. They implemented a multi-pass raycasting rendering method. During a given pass, each ray traverses a single cell, accumulates the cell contributions into the framebuffer and proceeds to the exit point adjacent cell. They were able to render between 500k and 600k tetrahedra/s, on an ATI Radeon 9700 graphics card with 128 MB of memory. However, the maximum size of the mesh was limited by the graphics memory size (up to 600k tetrahedra with a 512^2 frame buffer).

More recently, Bernardon *et al.* [Ber04a] improved Weiler *et al.*'s approach by using a depth-peeling technique in order to correctly render non-convex meshes, eliminating the need to perform a costly convexification pass. Their technique extracts successive boundary faces layers, starting the raycasting phase afresh from the currently extracted layer. Furthermore, using a static screen tiling scheme, they were able to reduce the number of raycasting passes. They reported to be able to render up to 1.3 Mtetrahedra/s on a 187 ktetrahedra non-convex dataset.

Callahan *et al.* [Cal04a] developed a hybrid image-space/object-space method, performing a coarse per-primitive sorting step on the CPU then a subsequent per-fragment refined step on the GPU. They used the multiple output buffer capability of modern GPUs in order to implement a fixed-depth sorting network, storing, for each pixel, an unsorted depth sequence of up to 4 fragments. Fragments determined to be the closest to the viewpoint are used to compute the contributions of a corresponding ray segment. They reported to be able to render between 1 and 2 million tetrahedra per second. However, in the case where the difference of the submitted fragment unsorted order and the correct one exceeds 4, an incorrect visibility order is determined and artifacts will appear, which can be frequent for unstructured data sets, where cells can vary greatly in size and shape.

Weiler *et al.* [Wei04a], improving on their previous work, were able to store unstructured meshes into graphics memory in a compressed form using tetrahedral strips and pre-rendering stripification algorithms. Furthermore, using a depth peeling technique akin to the one described in [Ber04a], they were able to correctly render non-convex meshes. They reported to be able to store up to 17 million tetrahedra on a 256 MB graphics card, and up to 3 million with speed optimizations.

3. TETRAHEDRA PROJECTION WITH POINT SPRITES

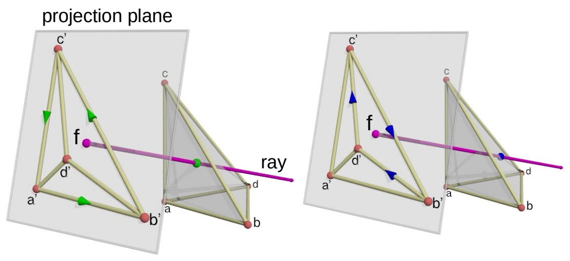
As we saw earlier, most tetrahedron projection implementations describe each tetrahedron by a set of four triangles, which is required by graphics APIs, which are not designed for the rendering of unstructured volume primitives. As a consequence, for a given tetrahedron, there is an overall duplication factor of up to 3 concerning per-vertex data (vertex coordinates, scalar field value) and per-face data (e.g. face plane equations). As the graphics port downstream bandwidth is limited (approximately 2.1 GB/s for AGP8x), this can result in a potential bottleneck and decrease the rendering performance. Ideally, for a given tetrahedron, its associated data should be transmitted without any duplication.

Here, we propose an enhancement to GPU-based tetrahedra projection methods allowing to transmit only the required data. To each tetrahedron, we associate a point sprite primitive instead of four triangles. For each of a given tetrahedron's four vertices, we specify the vertex (x, y, z) coordinates and scalar field value as point sprite vertex attributes, amounting to a total of 16 floating-point values per tetrahedron.

To each rasterized fragment of the sprite, we associate a ray. We use a vertex processing program in order to perform per-tetrahedron constant computations, such as edge line equations, whereas we use a fragment program to compute ray-tetrahedron intersection point depths, using the intermediary results computed by the vertex stage.

As a point sprite has fixed maximum dimensions, we assume for the remainder of this section that the tetrahedron projection's axis-aligned bounding box is included within the point sprite footprint. Also note that we consider only the case of orthographic projections.

The two vertex and fragment processing programs describe each tetrahedron as a set of four faces, each face being represented by a set of three co-edges, connected into a loop. For any given front-face (in world space), its projection's co-edges normals point towards the center of the face, whereas for any given back-facing face, the normals point towards the exterior of the face. Using this property, we can determine for any fragment whether it belongs to a given back-face or front-face, by computing fragment position-face edge equations dot products. We can then determine the two faces (one front-facing and one back-facing) whose projections cover the fragment, giving the ray entry and exit points. Figure 1 shows how intersected faces are determined.



•f is left to a'b', b'c' and c'a'
•abc is intersected and is a front face

•f is right to b'd', d'c' and c'b'
•bdc is intersected and is a back face

Figure 1. Intersecting faces determination

The pseudo-code below describes vertex stage computations:

- Transform the tetrahedron four vertices into window space
- Compute the point sprite bounding box dimensions
- Compute the line equations (in window coordinates) of each co-edge pair (corresponding to each projected tetrahedron edge)
- For each face vertex, compute the reciprocal of the result of its opposite co-edge line equation applied to the vertex

This amounts to the computation of six line equations, and twelve reciprocals (3 for each of the 4 faces). The second co-edge equations are deduced from the first ones simply by changing their signs. As the result of these computations is constant for any fragment, they can be done during the vertex processing stage instead of the fragment stage. The computed values are then written to output.

The following pseudo-code describes fragment stage computations:

- Compute the unnormalized window-space 2D distance of the fragment to each of the twelve co-edges
- For each of the 4 projected faces:
    - Compute the interpolated fragment depth
    - Compute the interpolated scalar value
    - Determine if the fragment is inside the projected face and whether it is a front-face or a back-face
- Determine the respective identities of the front-face and the back-face (if any)
- Compute the ray-segment length, subtracting the entry face z coordinate from the exit face's one
- Determine color and transparency values using the ray-segment length and write them to output
- If no intersected faces are found, write color and transparency values (0, 0, 0, 1) to output

Normalized Barycentric (NB) coordinates are computed, as illustrated in Figure 2, in order to determine the interpolated z and scalar values at the entry and exit points.

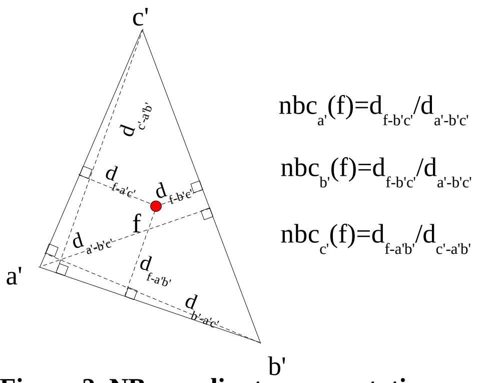

c'
a'
b'
f
$d_{c'-a'b'}$
$d_{f-a'c'}$
$d_{f-b'c'}$
$d_{a'-b'c'}$
$d_{f-a'b'}$
$d_{b'-a'c'}$
$nbc_{a'}(f)=d_{f-b'c'}/d_{a'-b'c'}$
$nbc_{b'}(f)=d_{f-b'c'}/d_{a'-b'c'}$
$nbc_{c'}(f)=d_{f-a'b'}/d_{c'-a'b'}$

Figure 2. NB coordinates computation

The unnormalized distance from the fragment position to each edge e is computed. It is then multiplied by the reciprocal of the relevant edge-opposite vertex v distance to produce a normalized barycentric coordinate $nbc_v(f)$. NB coordinates are then used to compute the interpolated depth and scalar values at the fragment: $z_{int} = nbc_{a'}\ z_{a'} + nbc_{b'}\ z_{b'} + nbc_{c'}\ z_{c'}$, where $nbc_{a'}$, $nbc_{b'}$, $nbc_{c'}$ are the NB coordinates, $z_{a'}$, $z_{b'}$, $z_{c'}$, the 3 face vertices window-space z coordinate and z_{int} the fragment interpolated z coordinate. Note that this equation is correct only for an orthographic projection. A perspective-correct interpolation formula should be used with perspective projection in order to get a correct result (see [Seg03a] and [Kra04a]).

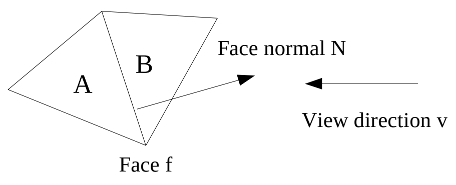
We use the same formula in order to interpolate the per-vertex scalar values across the entry and exit faces: $s_{int} = nbc_{a'}\ s_{a'} + nbc_{b'}\ s_{b'} + nbc_{c'}\ s_{c'}$. The two scalar values at segment extremities and the segment length are used as a 3D texture coordinate to perform a lookup into a pre-integrated volume integral texture, giving the color and transparency contributions of the ray segment.

4. GPU-ENHANCED BREADTH-FIRST SEARCH

During a given breadth-first search pass, the successor determination of currently examined nodes is performed sequentially on the CPU, only one node's neighborhood being explored at the same time. However, as each node examination can be performed independently from the others, it is well suited to a parallel implementation, especially on modern GPUs, which are able to process many independent data elements at the same time.

We present a multi-pass CPU-GPU hybrid breadth-first search implementation, executing successor determination on the GPU and subsequently determining visited nodes on the CPU, using the previously GPU-computed successor list. A pre-search phase on the GPU determines the adjacency of each graph node.

- N points outwards A, defined as the « out » cell
- N points inwards B, defined as the « in » cell
- A is behind B, relative to face f (ordering)
- B is in front of A, relative to v (adjacency)



- $(N.v) < 0$
- $s_A > 0$, thus $s_A(N.v) < 0$, A's successor through f is B
- $s_B < 0$, thus $s_B(N.v) > 0$, B has no successor through f
- B has one entering edge through f

Figure 3. Ordering, adjacency and successor determination

An adjacency graph is an oriented graph defined as {N, E}, respectively a set of nodes and oriented edges, associated to an unstructured mesh. To each mesh cell corresponds a node in the node list N. Two adjacent cells share a common face f which is used to define an ordering relationship, relatively to the face normal vector N, as depicted in Figure 3.

This ordering relationship allows us to compute an adjacency relationship, that is, to determine which one of the two adjacent cells is in front of the other one, relatively to the viewing direction v. The whole set of adjacency relationships is required to compute a correct depth order of the mesh by a graph search. To each adjacency relationship corresponds an edge e in the graph edge list. Note that the adjacency graph must be determined at each view parameter change. The edge list E is determined using the node and mesh faces lists.

Adjacency determination phase

Each list is stored into graphics memory as two different floating point 2D textures (RGBA and luminance). For each list, the dimensions of the two textures are the same. A node list element i stores the node's 4 face indices f, addressing the face list, and the node identifier. A face list element j stores the face normal vector N^j and the indices of the two “in” and “out” nodes sharing the faces.

The adjacency information is determined by the fragment program described below, using the four textures as an input. For each graph node, the program determines its successor nodes and its number of entering edges (in-degree), relative to the viewing direction and writes them to output. A full-screen quadrilateral is rendered, each fragment rasterized corresponding to a single graph node i. The following pseudo-code describes the operations performed:

- Set entering edges number to zero
- Fetch the node face indices and identifier from textures 1 and 2
- Translate the 4 1D face indices f1D into 4 sets of 2D texture coordinates f2D
- For each face f
    - Fetch f's information from face list using f2D
    - Determine the successor node (if any) corresponding to face f relative to the viewing direction
    - If f is not a boundary face, add one to the total entering edges number
- Write successor information (cell indices) to first output color
- Write number of entering edges to 2nd output color

Figure 3 describes in greater detail the determination of successor nodes through a given face f.

After the end of this phase, we copy back the computed successor information from the first output buffer into a floating-point texture, the adjacency texture, that will be used as an input during the search phase, whereas the entering edges information is read back from the second output buffer into main memory.

Breadth-First Search phase

During the search, the graph node list is stored in main memory, storing each node's state (unvisited or visited) and number of entering unvisited edges. The list of nodes currently examined is stored into a list on the GPU, which also maintains a search buffer. The pseudo-code below gives an outline of the search algorithm:

- Compute unvisited nodes count
- While unvisited nodes count greater than zero
    - Perform a search pass
    - Subtract visited nodes count from unvisited nodes count

Source nodes (nodes having no entering edges) are initially set as visited whereas all the other graph nodes are initially unvisited. They are also put into the list of currently examined nodes and subtracted from the unvisited nodes count. The following pseudo-code describes the execution of a given pass, and the operations performed for the two GPU and CPU steps:

For each given pass:

GPU:

- For each currently examined node (fragment)
    - Read its identifier, translate it into 2D coordinates

```
        id2D
    • Use id2D to read the four successor nodes IDs from the adjacency texture
    • Write them to output
CPU:
• Read back the search buffer
• For each node found, decrease the number of unvisited entering edges in the main list
    • If number equals 0
        • Mark node as visited with the pass number
        • Push node into the sorted list of nodes
        • Push node into the list of the nodes to be examined during the next pass
• Send the list of nodes to be examined back to the GPU
```

The GPU successor determination phase is implemented by a fragment processing program (as outlined by the pseudo-code above). During this phase, we render a quadrilateral such as each fragment rasterized corresponds to a node in the currently examined node list. Successor information, read from the previously computed adjacency texture, is written to output into the search buffer.

After the GPU phase, a sequential search of the readback search buffer is done on the CPU, examining each successor node and updating the list of nodes to be examined during the next pass. This list is then transmitted back to the GPU by updating the corresponding texture.

When the search main loop is over, the sorted identifier list is used as an element buffer in order to specify the mesh cells in the computed depth order.

5. RESULTS

We present results of our breadth-first search implementation and then of our point sprite-based tetrahedron rendering method. We performed measurements on an Intel 3.0 Ghz Xeon-based workstation with 1 GB RAM, an Nvdia NV40-based GeForce 6800 GT graphics board with 256 MB RAM on an AGP8x graphics port, and Redhat Linux Enterprise 4 as an operating system. We used the 1.0-7664 version of the Nvidia Linux graphics drivers. The OpenGL library was used for all our implementations.

Breadth-first search

We implemented the GPU part of our GPU-CPU BFS method with the ARB_fragment_program OpenGL extension assembly language, using specific instructions made available by the NV_fragment_program2 OpenGL extension, such as conditional execution or return instructions. In order to evaluate potential gains, we also made a software implementation of the breadth-first search algorithm, including the adjacency determination method and the actual search algorithm.

To test our implementations, we generated a simple dataset corresponding to a regular hexahedra grid that we converted into tetrahedra as described in [Shi90a], each tetrahedron corresponding to a single graph node. We used several grids of increasing size. For each grid, we performed a sequence of several searches, alternatively specifying two different viewing direction vectors. Table 1 gives adjacency computation and search times as well as the number of sorted nodes per second as a function of grid size, for the CPU-GPU and software-based methods.

Grid size	**40^3**	**50^3**	**59^3**	**70^3**
Nodes	$0.32x10^6$	$0.63x10^6$	$1.03x10^6$	$1.72x10^6$
CPU-GPU search results				
$t_{adjacency}$ (s)	0.05	0.08	0.11	0.95
t_{search} (s)	0.05	0.09	0.14	0.25
Sorted nodes/s	$2.6x10^6$	$2.8 x10^6$	$3.4x10^6$	$1.3x10^6$
Software search results				
$t_{adjacency}$ (s)	0.04	0.08	0.13	0.22
t_{search} (s)	0.02	0.05	0.11	0.23
Sorted nodes/s	$5.1 x10^6$	$4.6 x10^6$	$4.1 x10^6$	$3.7 x10^6$

Table 1. Graph search performance

We remark that the CPU-GPU search performance, in sorted nodes/s, increases as the grid size grows. However, for a grid size of 70^3, it decreases dramatically to $1.3x10^6$ nodes/s. This might be due to the size of the data textures, consuming nearly all the available graphics memory. On the contrary, software search performance decreases as the grid size grows, as for each pass more nodes have to be examined, still in a sequential way, therefore increasing the quantity of work to do.

Point-sprite Based Tetrahedron Rendering

We used the Cg graphics programming language in order to implement the algorithms described in 3, using an FP40 rendering profile.

In order to perform the object-space pre-rendering cell sort, we used the software breadth-first search implementation we mentioned above. Rendering was made using immediate mode, specifying each tetrahedron with 1 glVertexAttrib3f call for each of the 4 vertices (x, y, z) coordinates sets and one glVertexAttrib4f call for the tetrahedron's 4 per-vertex scalar values.

The dataset used for our tests, a tetrahedrized rectilinear grid, stems from a numerical simulation of the propagation of seismic waves. We used the magnitude of displacement vector as the scalar field. Volume rendering reveals the spatial structure of wave amplitude (Figure 4). For each single measurement, we rendered a sequence of 100

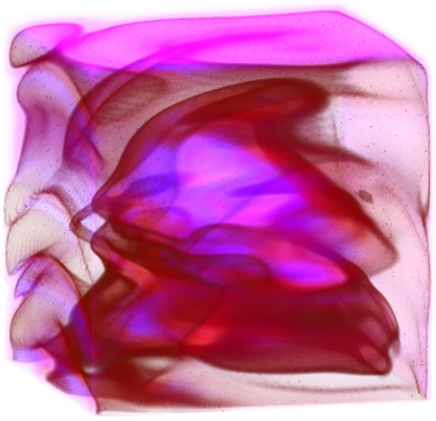


Figure 4. 1250k tetrahedra dataset, 512x512 pixels

images, rotating the view point rotation about the center of the dataset. We measured the average sorting (including adjacency determination), rendering, and total frame time, the latter being used to compute rendering performance. We activated linear filtering of the pre-integrated transfer function texture, which was computed separately by an offline process. The number of tetrahedra rendered per second was measured as a function of resolution.

Table 2 represents execution times of respectively the cell sorting step and the rendering step, the frame total time, and the number of tetrahedra rendered in the 10^6 unit per second.

Resolution	**256^2**	**512^2**	**768^2**	**1024^2**
$t_{sorting}$ (s)	0.33	0.33	0.33	0.33
$t_{rendering}$ (s)	0.36	0.86	1.74	2.92
t_{frame} (s)	0.7	1.21	2.09	3.29
Tetra/s (x10^6)	1.78	1.03	0.6	0.38

Table 2. Rendering speed as a function of resolution

We observe that for a resolution of 512^2 pixels, rendering speed is about 1.0x10^6 tetrahedra rendered/s. Sorting time is constant whatever the resolution, at it is only dependent on the mesh size and viewing direction.

We performed measurements in order to highlight the cost of fragment processing, disabling the sorting step and using a null fragment program doing no computations, only writing the color (1, 1, 1, 1) to output. Table 3 indicates the average rendering time and speed as a function of resolution. We also indicate the ratio between the rendering time shown in Table 2 and the rendering time with no fragment processing.

We observe that for a resolution of 256^2 pixels, rendering speed is about 6.3x10^6 tetrahedra/s. Rendering speed decreases with resolution whereas the rendering time ratio increases, only slightly decreasing from 512^2 to 768^2.

Resolution	**256^2**	**512^2**	**768^2**	**1024^2**
$t_{rendering}$ (s)	0.2	0.2	0.2	0.27
Tetra/s(x10^6)	6.27	6.31	6.14	4.64
Rendering time ratio	1.8	4.3	8.7	10.8

Table 3. Rendering speed with no fragment processing

The fact that the time ratio is high, even for low resolutions, indicates that the fragment processing cost seems to be the bottleneck of our rendering method. The increase of the rendering time is probably due to the increasing rasterization costs (including alpha-blending).

Finally, in order to evaluate the bandwidth used to transmit the whole mesh from main memory to the graphics card, we used a null vertex processing program, keeping only the vertex projection and sprite size computation. We also disabled sorting. Table 4 represents rendering time as a function of resolution.

Resolution	**256^2**	**512^2**	**768^2**	**1024^2**
$t_{rendering}$ (s)	0.2	0.2	0.19	0.2
t_{frame} (s)	0.2	0.2	0.2	0.2
Bandwidth (MB/s)	405	398	408	399

Table 4. Bandwidth with no vertex processing

We compute bandwidth as the amount of data transmitted over the rendering time, that is, the ratio between mesh size times tetrahedron size (64 bytes) and the average frame time. A glFinish command ensures that all data be transmitted between the start and the end of the rendering. Bandwidth does not vary with resolution, as fragment processing is disabled, and is lesser than 1/4th of the AGP8X theoretical maximum bandwidth. However, rendering time with no vertex and fragment processing is lesser than rendering time with processing activated (see Table 2), which seems to indicate that bandwidth is not a limiting factor. Further investigation is necessary in order to see if the fragment processing and data transmission times balance better with larger meshes.

6. DISCUSSION

As we precedently showed, our point-sprite based rendering method significantly decreases the amount of data transmitted through the graphics port with 64 bytes transmitted per tetrahedron. Furthermore, it decreases the number of vertices processed by the vertex stage to only one per tetrahedron, allowing to perform more complex computations. However, it appears to be limited by the fragment stage, as the quantity of computations done for each fragment is quite important. Computations that are done at each fragment, such as multiplying each face vertex z coordinate and scalar value by the reciprocal value of the distance to the opposite edge could be done in the vertex stage, saving a significant amount of computation. Interpolated scalar values and fragment depths have to be explicitly calculated by the fragment program whereas triangle-based view-independent methods by specifying them at each triangle vertex can determine them by using hardware linear interpolation. Also note that the size of any projected tetrahedron is limited by the point sprite maximum size. Tetrahedra with a larger footprint will be incorrectly processed. However, GPU fragment processing power has dramatically increased in the past few years, whereas graphics port bandwidth has undergone a much slower rate of growth. We think that this will still be the case for at least several years to come. Therefore, it might be likely that our method's performance will scale better with the increasing GPU fragment processing power that the performance of other projection methods which might become limited by graphics port bandwidth. Future software and hardware evolutions might also eliminate the point sprite size limitation. Moreover let us remind that GPU-based raycasting methods such as described in [Wei03a] or [Ber04a] require to store the whole geometry in graphics memory. Tetrahedron projection methods are not subject to this severe limitation since they allow streaming from main memory.

The partly GPU-based BFS method we described takes advantage of the multiple fragment units of GPUs in order to speed up the successor determination. However it performs a readback of successor information into main memory in order to update the global node list, which costs graphics port bandwidth and CPU time. Nevertheless, for a $1.7x10^6$ node graph, the search time for the GPU-CPU based method and the purely CPU-based one are about the same, indicating that GPU successor determination might be fast enough to compensate for the read-back and re-send penalty as well as the additional CPU cost. Eliminating the readback and performing the CPU work on the GPU might increase the CPU-GPU BFS performance in a significant way, allowing it to sort large meshes (with more than $2x10^6$ cells).

7. CONCLUSION AND FUTURE WORK

After this first implementation of point sprite-based tetrahedron projection method, we intend to test the rendering of larger datasets, with at least 10^6 cells, and compare its performance with other GPU-based tetrahedron projection method. We also plan to improve our partly GPU-based BFS method by eliminating the CPU visited-node determination, using “render to vertex array” capabilities.

8. ACKNOWLEDGMENTS

We would like to thank Jean-Philippe Nominé for his thorough and insightful reviewing of this work, as well as the anonymous reviewers for their useful comments. We also would like to thank Dominique Rodrigues for PRODIF datasets.

9. REFERENCES

[Ber04a] F.F Bernardon, C.A. Pagot, J.L.D. Comba, C.T. Silva, GPU-based Tiled Ray Casting using Depth Peeling, SCI Institute Technical Report, No UUSCI-2004-006, University of Utah, 2004

[Cal04a] S.P. Callahan, M. Ikits, J.L.D. Comba, C.T. Silva, Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering, SCI Institute Technical Report, No UUSCI-2004-003, University of Utah, 2004

[Coo04a] R. Cook, N. Max, C.T. Silva, P.L. Williams, Image-Space Visibility Ordering for Cell Projection Volume Rendering of Unstructured Data,, IEEE Transactions on Visualization and Computer Graphics, Vol 10, N° 6, 2004

[Kin00a] D. King, C.M. Wittenbrink, H.J. Wolter, An Architecture for Interactive Tetrahedral Volume Rendering, HP Labs Technical Report, HPL-2000-121R3, 2000

[Kra04a] M. Kraus, W. Qiao, D.S. Ebert, Projecting Tetrahedra without Rendering Artifacts, Proceedings of IEEE Visualization 2004, pp. 27-34, 2004

[Max95a] N. Max, Optical Models for Direct Volume Rendering, Transactions on Visualization and Computer Graphics, Vol. 1, N° 2, pp. 99-108, 1995

[Mor04a] K. Moreland, E. Angel, A Fast High Accuracy Volume Renderer for Unstructured Data, Proceedings of IEEE Symposium on Volume Visualization and Graphics 2004, pp. 9-16, 2004

[Nvi04a] NVIDIA Corporation, Programming Graphics Hardware, Eurographics 2004, 2004

[Seg03a] M. Segal, K. Akeley, The OpenGL Graphics System: A Specification (Version 1.5), Silicon Graphics, Inc., 2003

[Shi90a] P. Shirley, A. Tuchman, A Polygonal Approximation to Direct Scalar Volume Rendering, Proceedings of San Diego Workshop on Volume Visualization, Computer Graphics, vol. 24, N° 5, pp. 63-70, 1990

[Wei02a] M. Weiler, M. Kraus, T. Ertl, Hardware-Based View-Independent Cell Projection, Proceedings of IEEE Symposium on Volume Visualization 2002, pp. 13-23, 2002

[Wei03a] M. Weiler, M. Kraus, M. Merz, T. Ertl, Hardware-Based Ray Casting for Tetrahedral Meshes, Proceedings of IEEE Visualization 2003 , pp. 333-340, 2003

[Wei04a] M. Weiler, P.N. Mallon, M. Krauss, T. Ertl, Texture-Encoded Tetrahedral Strips, Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics, pp. 71-78, 2004

[Wil92a] P.L. Williams, Visibility-Ordering Meshed Polyhedra, ACM Transactions on Graphics, Vol. 11, N° 2, pp. 103-126, 1992

[Wyl02a] B. Wylie, K. Moreland, L.A. Fisk, P. Crossno, Tetrahedral Projection using Vertex Shaders, Proceedings of IEEE Volume Visualization and Graphics Symposium 2002, pp. 7–12, 2002