On-the-fly Processing of Compressed Volume Data

Ph.D. Dissertation Proposal

Chuan-kai Yang
Department of Computer Science,
State University of New York at Stony Brook,
Stony Brook, NY 11794-4400.
ckyang@cs.sunysb.edu

November 6, 2001

Abstract

The size of volumetric data sets grows rapidly with advances in computing technologies. Volume compression, therefore, has been proposed as an effective technique to reduce the associated storage requirement. However, before rendering, a decompression step is usually needed. We explore several approaches to integrate compression and rendering in a single framework so that either rendering can proceed without decompression, or rendering can be performed on the fly during decompression. This framework greatly reduces the data loading time as well as the memory footprint. Sometimes the reduction in memory usage can lead to further performance improvement, especially for those data sets whose peak memory requirements exceed the physical memory size (out-of-core rendering). In addition, we have also developed methods to integrate volume simplification into this framework, thus making it possible to perform volume simplification and rendering on the fly during decompression. Finally, we propose two more extensions. The first one is to perform "just-in-time" rendering, which tries to complete a rendering task on a given delay budget, by adjusting the simplification ratio. The second one is to perform iso-surface extraction based on a "layer-based" structure, which is a different way to decompose a data set than the conventional "coordinate-based" structure. Each layer contains a collection of tetrahedral whose associated data values fall within a specific range, and is compressed independently to reduce storage requirement.

1 Introduction

Volume visualization has become more and more popular during the last decade. Due to the advances made in data acquisition technology, the size of volume data sets has increased tremendously. One common technique to address this issue is compression. However, without careful application, compression can only be used to reduce the storage requirement, but not necessarily to improve the rendering performance. A naive implementation writes the decompressed data sets to the disk and then reads it back from the disk again before proceeding to the rendering step. Although the write and read of the uncompressed data sets seem redundant, this pattern of use is not uncommon, and it is clear that this does not take full advantage of compression. From the renderer's point of view, the rendering latency (including decompression delay), data loading time and memory requirement still remain the same. One can eliminate the intermediate write and read of the uncompressed data sets by "bridging" the output of the decompressor and the input of the renderer, thus reducing the data loading time, but the problems of rendering latency and memory requirement still remain unaddressed. Better ways than this naive "decompress-then-render" strategy exist and here we have explored some of them. Perhaps the most elegant way is to perform rendering completely in the *compression domain*, i.e., rendering can be done without decompression at all. This can be done through the use of FPST (*Fourier Projection Slice Theorem*), which is provably faster in terms of time complexity, but it suffers from the loss of quality, i.e., only "X-ray" like images are possible. By partitioning data sets into sub-cubes, applying compression domain rendering to each sub-cube, and spatially compositing the sub-images together using their aggregate colors and opacities, we can achieve better image quality by simulating the *self-occlusion* effect that was missing in the whole-volume case. However, for non-orthographic views, even this sub-cube approach cannot completely avoid the *aliasing* or *ghost* effects.

The second strategy is to perform *on-the-fly* rendering during decompression. The renderer need not wait for the decompression of the entire data set but can proceed once a tetrahedron is decompressed. Data loading time is reduced because a data set is loaded from its compressed form. By incorporating a garbage collection scheme to deallocate a tetrahedron as soon as its rendering is done for sure, the memory footprint of the renderer can be reduced significantly. A smaller memory footprint in turn can further boost the system performance. In this system, which is called Gatun, basically the decompressor dictates the order in which a tetrahedral mesh is traversed, and the renderer needs to be modified to generate the correct rendering result.

Other than incorporating rendering into decompression, *volume simplification* can also be integrated. We have also modified Gatun so that decompression, simplification and rendering can be integrated into the same pipeline. Previously volume simplification is performed in a preprocessing mode, and at run time, according to the user specified simplification ratio, a tetrahedral mesh can be simplified on the fly. Only those tetrahedra that survive the simplification step should be sent to the renderer.

We are proposing two more extensions to this framework in this proposal. The first extension is to perform a "just-in-time" tetrahedral grid rendering, or essentially a *rendering on a fixed time budget* scheme. For example, according to the mouse movement (for rotation, translation, scaling), different simplification should be applied, i.e., the faster the mouse movement is, the more aggressive simplification could be applied to reduce the number of tetrahedra included in rendering and display. The second extension is to support iso-surface extraction in this framework. Instead of the traditional way of decomposing volume data sets spatially according to vertices' geometrical coordinates, we propose another decomposition scheme based on the scalar value associated with

each vertex. This is called "layer-based" decomposition, because a volume data set can be considered as layers of tetrahedra with similar data values. Each layer is also compressed to further reduce its storage requirement. One may also apply simplification to each layer on the fly during decompression.

The rest of the paper is organized as follows. Section 2 reviews some of the related work. Section 3 describes our past work on compression domain rendering for regular-grid volume data. Section 4 details the work on how to perform on-the-fly rendering of tetrahedra grids during decompression. Section 5 explains how to extend the work done in Section 4 to further integrate simplification into the pipeline. Section 6 proposes two new extensions to the whole system. Appendix A provides some mathematical details on some of the formulas we used for interpolation within a tetrahedron.

2 Related Work

Multiple fields are involved in our work, therefore the related work for each of them will be discussed.

2.1 Compression of Volume Data

Since there are basically two kinds of volume data: (see Speray et al. [SK90] for the classification of volume data) regular (in particular we mainly refer to volume data sets consisting of *Cartesian Grid* or so called *voxels*) and irregular, we list their related work separately.

2.1.1 Compression of Regular Volume Data

On compressing regular volume data, there are two types of approaches: lossy algorithms versus lossless algorithms. Since volume data sets tend to get bigger and bigger, lossy compression seems to be more attractive than lossless compression; however, when fidelity is crucial, such as in the case for medical diagnoses where we do not know how the data lossiness affects the rendered images, lossless compression may prove its use. Surprisingly to our knowledge there have been rare studies on this. Fowler et al. [FY94] generalize the idea of DPCM (Differential Pulse-Coding Modulation) and applies it to 3D volume data set to capture the 3D coherence that is difficult to explore using traditional window-based approach, such as *gzip*. Together with Huffman coding, this algorithm can achieve about 50% compression ratio, though not a very impressive result compared with typical lossy compression algorithms, is claimed to be very close to the "entropy", or the theoretically obtainable lower bound size to encode those tested data sets.

For lossy volume compression, there are relatively more studies. Among them, we will discuss several methods: vector quantization, fractal compression, Laplacian compression, Fourier compression and wavelet compression. In general, most of the compression efficiency comes from doing a scalar quantization, which approximates the original data values by a much smaller number of values.

Discrete Fourier, Hartley and Cosine Transforms Discrete Fourier transform has long been used in the signal processing world. The basic idea is to get a different distribution, preferably a more biased one in the transformed domain, thus making the key compression technique, quantization, more effective. It is known that applying quantization directly in the original domain (often referred to as the spatial domain) often leads to poor quality, but could offer astonishing results in a properly chosen *transformed* domain. One drawback of Fourier

transform is that it maps real numbers to complex numbers therefore it doubles the input size thus making its use for data compression less appealing. Cosine transform, on the other hand, not only keeps most of the merits from Fourier transform but also maps real numbers to real numbers. An alternative is to use the discrete Hartley transform, which also maps real numbers to real numbers, and in some case better than cosine transform because the so called FPST (Fourier Projection-Slice Theorem) can hold for Hartley transforms. However, in terms of computation cost and energy compaction efficiency, it is still better to perform cosine transform than Hartley transform, so in practice, cosine transform is widely used than both Fourier transform and Hartley transform. For example, the famous JPEG and MPEG algorithms apply cosine transform on still images or videos to obtain reasonably well compression efficiency in most cases. Further observation can reveal some redundancy in the computation of these transforms, therefore oftentimes FFT (fast Fourier transform), FHT (fast Hartley transform) and FCT (fast cosine transform) are used in place of the original transforms to speed up the computation. An implementation of FFT and FCT can be found in Press et al. [PTVF97], while some ideas on implementing FHT were reported by Bracewell et al. in [Bra84, BBHV86] and Yang in [Yan89]. In addition to images and videos, cosine transforms can also be applied to volume data sets, as being done by Chan et al. [CCM91] who applied the 3D DCT on the whole volume data set, while Yeo et al. [YL95] partitioned volume data sets into "sub-blocks" and applied 3D DCT on each of them to compress the volume data.

Vector Quantization A general introduction to vector quantization can be found in Gray [Gra84]. This is essentially a generalization from a one dimensional scalar quantization to n dimensional vector quantization; or equivalently, a relatively few number of vectors are used to approximate the the original vectors. The associated error for a particular quantization is often measured in terms of MSE (Mean Square Error). Those small number of representative vectors form what is often called the "code-book". One application of vector quantization is image compression, where each image is partitioned in multiple "sub-blocks" of size say, $m \times m$, then equivalently, it is using a m^2 dimensional vector quantization approach. Similar idea can be applied to volume compression, as being done by Ning et al. in [NH92].

Fractal Compression The theory behind fractal compression is non-trivial, and some good reference could be found in Jacquin [Jac89] and Barnsley et al. [BDM$^+$88] In a nutshell, there is much resemblance between fractal compression and vector quantization. In both schemes, we search the code-book and try to find and use one vector that best represent a given vector. However, they differ in at least two aspects. First, in fractal compression, "similarity" is defined with respect to results of a finite number of "contractive", "affine" (see [Jac89] or [Yan00] for a detailed explanation) transforms on the given vector. Or to put it more precisely, for a given input vector, we will find a vector from the code-book, which, under any contractive, affine transforms, will best approximate the given vector. Second, unlike in vector quantization, in fractal compression, the code-book need not be kept separately but can be generated dynamically during the decompression process. Fractal image compression, as shown in Fisher [Fis95] is one of such applications. It has also been extended to deal with volume compression as well, as done in Cochran et al. [CHF96]. Although the saving of a code-book and the easiness of decoding process make fractal compression an attractive scheme, its extremely time-consuming encoding process and oftentimes non-exact restorations

render it an impractical approach in most cases, at least for volumetric data.

Laplacian Transform Laplacian decomposition and wavelet decomposition, are two alternatives to build a "pyramid-like" data hierarchy, or to perform a so called MRA (multi-resolution analysis). Through this MRA representation, we can progressively refine 1D signals, 2D images, 3D volumes, or other possible objects to suit our needs for data transmission, interactive manipulation, and so on. The basic idea of Laplacian transform is very simple. Take 2D images as an example, the Laplacian transform first low-passes and down-samples the input image to get an image with half the resolution, then takes the difference image between the expanded image obtained from the previous sub-sampled image, and the original image. The same process applies iteratively by treating the sub-sampled image as the input image again. The collection of the original input image plus all the sub-sampled version at each stage compose the "Gaussian pyramid" while the collection of different images from each stage form the "Laplacian pyramid". For reconstruction, only the Laplacian pyramid and the smallest (toppest) one from the Gaussian pyramid are needed. As suggested by most of the cases, very aggressive quantization scheme can be applied on the Laplacian pyramid to achieve good compression ratio. Chan et al. [GY95] also applied this idea on volume compression.

Wavelet Transform Wavelet transform is one of the best transforms to provide high compression ratio. Some excellent references can be found from Chui [Chu92a, Chu92b] and Daubechies [Dau92] from a mathematical point of view, and Mallat [Mal89] from a signal processing point of view, together with some applications on image decomposition/reconstruction and processing. In practice, we just need to choose a suitable wavelet function and its corresponding scaling function, which will determine the corresponding high-pass filter and low-pass filters used for convolution during the decomposition process. The part generated by the high-pass filter is the detail signal while the part by low-pass filter the approximation signal at a lower resolution and input signal for the next (coarser) level of decomposition. The choice of the scaling function and wavelet function, consequently the choice of filters will affect the performance or the reconstruction quality. There have been some popular wavelets: Haar, Daubechies, and Battle-Lemarié. The First two families have finite support (or finite filter length), and are non-symmetric. Battle-Lemarié wavelets are symmetric but have an infinite support. Finite support wavelets lead to a better performance, but truncation in the wavelet domain usually leads to "blocky" reconstruction, while infinite support wavelets tend to provide more smooth reconstruction, at the expense of slower performance; and a truncation in the filter length, which always leads to an imperfect reconstruction, is inevitable. There are several advantages of applying wavelet transforms. First, compared to Fourier transform, wavelet transform benefits from the "local support" property. Even in the case where theoretically infinite filter length is needed, the impulse response decreases rapidly within a certain range, therefore a truncation or general quantization in the wavelet domain only affects "locally" in the original spatial domain. Unlike the wavelet transform, a tiny change in the Fourier frequency domain, will often affect the spatial domain "globally", as shown in some cases where the reconstructed image gets blurred due to a change in a frequency domain coefficient. Second, wavelet has the property called "vanishing moment" [Chu92a], which is defined to be the largest value to make all integrals vanish, and this is why wavelet means "little" wave. Applying wavelets of bigger vanishing moments in the transforms results in a large number of zero coefficients, which is excellent for data compression. Third, the orthogonality of the scaling function and

wavelet function guarantees the best approximation in the corresponding vector space, which means for multi-resolution analysis, the approximation at each resolution level is the best, at least in the mathematical sense. Moreover, sometimes this also makes the estimation of error possible at some sample point and at some image resolution, as done in Westermann [Wes94]. Fourth, the progressive approximation and hierarchical setup are good for interactive use or data transmission, where users can first see an approximation of the data, then more and more details are added as time or memory budget allows. Muraki [Mur93] extended the work of Mallat [Mal89] to apply 3D wavelet decomposition on volume data sets and use simple truncation (a special case of quantization) to retain only part of the wavelet transformed coefficients according to their magnitudes for volume rendering applications.

2.1.2 Compression of Irregular Volume Data

The compression algorithms we will discuss here are all lossless algorithms while lossy algorithms, although exist, usually appear in the form of "simplification". Some simplification algorithms on irregular volume data will also be discussed later.

Grow and Fold Szymczak et al.'s "grow and fold" approach [SR99] represents an irregular-grid data set as a tetrahedral spanning tree. Starting from an arbitrary tetrahedron, the algorithm *attaches* the next tetrahedron to one of its four faces, if that face is *attachable*. From then on, each tetrahedron has at most three possible "external" faces to attach further tetrahedra. Therefore three bits are used to decide which of the external face is used to *grow* the tetrahedral spanning tree. For each external face, a 2 bit *fold string* is used to indicate which edge or none to use to fold this face with the corresponding face. It is possible that a face does not fold with its neighboring triangles but with a non-adjacent face. A typical case is a data set with holes, and when this happens an index of the triangle is needed to uniquely identify how to *glue* these two faces together. Fortunately in practice, this extra bit representation only happens with a fairly low probability, thus making the average bit number for representing a tetrahedron just about 7 bits.

Cut-Border Gumhold et al. [GGS99] basically extended their previous work [GS98], which is an compression algorithm for triangular meshes, to compress tetrahedral meshes. In [GS98], it is observed that given a border of the enumeration, initially set to be the edges of an arbitrary triangle, there are three ways to form a new triangle from each edge in the "cut-border". First, this edge is already on the boundary of the mesh, so no new triangles can be formed. Second, this edge can form a new triangle by paring with an edge on the cut-border. Third, this edge, together with a new vertex, can form a triangle. These cases are called, in order, as *border*, *connect* and *new vertex*. For the *connect* case, it needs to specify which edge on the cut-border to be used, and therefore an offset is needed. It turns out over 95% of the time, the new triangle is formed by the *new vertex* and *connect forward* cases, where the latter one means the pairing edge is the next edge on the cut-border, therefore the compression result is very good (less than 2 bits/per triangle). For tetrahedral meshes, the same idea applies, with the definition of cut-border modified from collection of border edges to collection of border triangles. Now a new tetrahedron is formed in three ways from each face (triangle) on the cur-border. First, it is already a border face. Second, it forms a new tetrahedron with another face(s) from the cut-border. Third, it forms a new tetrahedron with a new vertex. Except in the second case, they are very similar to the triangular mesh case. However, it is

exactly the second case which makes things more complicated. As a result, in [GGS99], totally 10 cases are distinguished. Furthermore, unlike in the case of triangular mesh compression, where the *new vertex* cases accounts for half of the cases, the *connect* cases appear most often here. Fortunately a careful enumeration/traversal order makes only several of them most dominant. Together with the arithmetic coding, the result is just a slightly more over 2 bits per tetrahedron. A vertex's data is put into the compressed representation whenever it appears as a pairing vertex in a *new vertex* operation, since each vertex will be touched only once this way. Also, the vertex order may be permuted/reordered due to this encoding process. The decoding process is straightforward, therefore we omitted it here.

Gatun In Yang et al. [YMC00], a different approach for compressing tetrahedral mesh is proposed. Because this method is one of the fundamental algorithms of this proposal, we will explain it in more details later in Section 4.

Implant Sprays Pajarola et al. [PRS99] used a different approach than other tetrahedral mesh compression algorithms. Instead of operating the compression algorithm on a single resolution, their algorithm encodes the progressive tetrahedralization process, which will also be discussed later in the volume simplification sub-section. Two basic primitives: edge-collapse and vertex split, are used in the encoding and decoding process separately. To encode an edge collapse, the *orbital surface* that will be affected by this operation should be identified. However, the enumeration of such orbital surface is not trivial and an ad hoc method is used. Some special care needs to be taken when this surface is non-manifold or when dealing with the border of a tetrahedral mesh. To speed up the decoding process, their algorithm allows the vertex split operations to proceed simultaneously to the next refined *level of detail* (LOD) as long as these operations are "independent" from each other, i.e., their orbital surfaces are disjoint. More constraints are given to maximize these independent sets but we omit the details here. One interesting aspect of this approach is its integration of compression and progressive tetrahedralization; however, it seems the compression efficiency is not particularly high compared to some other reported tetrahedral mesh compression algorithms that can achieve less than 3 bits per tetrahedron by only working on a single resolution.

2.2 Rendering of Volume Data

There have a been a great deal of researches on volume rendering. In general two different types of approaches are used. The first type is called *direct volume rendering*, and the second type is called *iso-surface extraction*. Unlike the first type of approaches, the second type of approaches extract the surface in terms of a polygonal model according to a given iso-value, and sends the model for traditional graphics rendering. We will defer iso-surface extraction to the next sub-section and discuss direct volume rendering first.

2.2.1 Rendering of Regular Volume Data

Because of the different structures of volume data sets, the rendering algorithms also look quite different. Algorithms for rendering regular volume data are reviewed here while those for rendering irregular volume data will be reviewed next.

The use of different optical models could generate very different looking images. Williams et al. [WM92] and Sabella [Sab88] treated volume data sets as particle models and each voxel a density

emitter. Max [Max95] reviewed several more sophisticated optical models including (multiple) scattering and shadowing, in addition to absorption and emission, and their combination. However, in practice, usually most rendering methods adopted a very simple optical model: light source is only from outside of the volume, and there is no scattering, no refraction, and only partial absorption and/or attenuation. This optical model, although very simple, could often provide very good visualization results, especially when dealing with data sets containing transparent materials, which was more difficult to represent using the approach of iso-surface extraction. Although nowadays we can use the technique of *α blending* to show multiple surfaces at a time, direct volume rendering methods still have the benefits of providing more flexibility and images of higher quality.

In addition to the optical model involved, there are several terms need to be clarified.

interpolation If a sample point is not on a voxel's location, then an interpolation scheme is needed to "derive" the value at this sample point. Usually a *tri-linear* interpolation is employed to make use the 8 closest voxel neighbors to interpolate the value at the desired sample point location. Other lower or higher order of interpolation schemes are also possible. In our system, we use the tri-linear interpolation scheme for regular grids and the quad-linear (see Appendix, similar to tri-linear) interpolation scheme for irregular (tetrahedral) grids.

shading and classification Shading means assigning color values based on the scalar (or so called "density" or "intensity") values while classification means assigning opacity values based on the scalar values. The conversion process is defined by using some user-specified *transfer functions* to "select" or "focus" only the region of interest. Depending on when this conversion takes place, there are usually two possible choices: pre-shading (pre-classification) or post-shading (post-classification). Pre-shading (and pre-classification) means all the colors (and opacities) are determined in the pre-processing time (based on some transfer functions) before any visualization of the data sets. When viewing parameters are given, colors and opacities are interpolated if sample points are not on the voxel grid locations. Post-shading (post-classification) means there is no color (and opacity) conversion in the pre-processing time and at rendering time only density values are interpolated if necessary and then converted to colors (and opacities) afterwards. If *directional shading* is to be concerned, for pre-shading model, the gradient vector, usually obtained by taking a *central difference* from neighboring voxels, could be derived and used as the local normal vector thus a simple *Phong shading* lighting model may be applied to obtain the final color. For post-shading model, special care should be taken to determine the normal vectors, as done by Mueller et al. [MMC99]. We use post-classification and post-shading in our implementation, without directional shading.

compositing For the same pixel, the contribution from two contiguous sample points needs to be combined to take into account the proper occlusion and attenuation effect. There are usually two possible orders: "front-to-back", as in Sabella [Sab88] and Levoy [Lev88], or "back-to-front", as in Drebin et al. [DCH88] and Levoy [Lev90a]. Later we will use the following front-to-back compositing formulas:

$$C_{out} = C_{in} + (1 - O_{in})C_v O_v \tag{1}$$

$$O_{out} = O_{in} + (1 - O_{in})O_v \tag{2}$$

Here C_v and O_v are the color and opacity values for the current sample point v, C_{in} and O_{in} are the accumulated color and opacity values before visiting v, and C_{out} and O_{out} are the accumulated color and opacity values after visiting v.

Direct rendering methods can be further classified as image-order (or backward mapping) algorithms, object-order (or forward mapping) algorithms, and hybrids of the two. There are also some algorithms that can not be readily classified in this taxonomy. A general review on volume rendering methods can also be found in [Kau91, Kau94] written by Kaufman.

Image-Order Algorithms Basically this type algorithm shoots rays from the image plane, which is oriented by user-specified viewing parameters. Sample points are taken on each ray and interpolation is often necessary if they are not at the exact voxel grid locations. Color and opacity contribution given by two contiguous sample points are composited using front-to-back or back-to-front formula depending on the ray traversal direction.

MIP and XRay rendering MIP (maximal intensity projection) means for each ray, only the sample point containing the highest interpolated density will be projected to the image plane, therefore no compositing is required; whereas an XRay rendering sums up (a simplest form of compositing) all the interpolated density values of all the sample points along each ray.

Ray-Casting Ray-Casting is one of the most famous image-order algorithms. Levoy [Lev88] proposed one of the earliest forms of raycasting where he employed the pre-shading and pre-classifying approach and the tri-linear interpolation, front-to-back compositing scheme.

Object-Order Algorithms Object-order algorithms render image in a different way. Voxels or their affected region are projected on the image plane. To have correct results, voxels must also be traversed in order, i.e., in a front-to-back or a back-to-front order. The time-consuming tri-linear interpolation part presented in raycasting can be saved in this kind of approaches.

Splatting Splatting, first proposed in [Wes89] by Westover is the most visible object-order rendering approach. Instead of calculating the contribution made by multiple voxels or 3D sample points to a given pixel on the rendered image, one calculates the contribution of each voxel to the pixels on the rendered image. For this reason, a *3D reconstruction kernel* centered at each voxel is used to reconstruct the 3D continuous data. To know the contribution of a particular voxel to the final image, its *footprint* on the image plane is calculated. The calculation of the footprint involves integrations which may be time consuming. However, at least for parallel projection, the footprint is independent of the voxel location, therefore these integrations need to be done only once, and the computation overhead is not important. A footprint of a voxel is then composited with the resulting image computed so far. To have the correct order, instead of sorting, Westover proposed the idea of using the *sheet buffer* in [Wes90], where a volume data set is processed sheet by sheet, and a sheet is defined as a plane through the data that is *most parallel* to the image plane. Notice here footprints are not necessarily aligned with the pixels on the image plane, therefore an interpolation/re-sampling process is usually required.

Hierarchical Splatting To speed up rendering further, Laur et al. [LH91] used Gouraud-shaded RGBA polygons to approximate the footprints to take advantage of modern graphics hardware. This implementation also features a hierarchical approach, which decomposes a data set into an octree of "cells" according to the colors. Cells of different

sizes have "splats" of different sizes, or footprints. Each cell stores the average RGBA of its children and the associated average error if this region is to be approximated by the average splat. This allows users to specify the desired error, and the algorithm searches for the appropriate level in the pyramid, projects the splats with different sizes according to their levels, and composites them in a back-to-front order. This progressively refinement approach is particularly suitable for interactive browsing.

Quality Issues Because splatting eliminates the sample-value interpolation, which is usually the dominant cost in raycasting, it is usually faster than raycasting. However, the footprint table size needs to be carefully tuned: smaller table offers faster performance but blocky images while bigger table may smooth out details and also slow down the rendering. Moreover, the original methods proposed by Westover could create *color-bleeding* or "popping" effect, as pointed out and fixed by Mueller et al. [MC98b] through the use of sliced interpolation kernels. Furthermore, compared to raycasting, splatting tends to generate more blurred images. Mueller et al. [MMC99] applies a post-shading model splatting, together with the *iso-thresholding* to clip off some blurred image components so that generated images becomes much sharper.

Hybrid and Miscellaneous There are other visualization methods. They may be a combination of both image order approach and object order approach or do not fall into either one of them.

V-Buffer Proposed by Upson et al. [UK88], to divide a data set into cells, and render it cell-by-cell, which makes this approach an object order method. However, within each cell a raycasting is used, and therefore this is a hybrid approach.

Ray-driven Proposed by Muller et al. [MY96] to make *early ray termination* (will be explained later) possible by raycasting the volume of "suspended" splats around each voxel.

Shear-Warp Proposed by Lacroute et al. [LL94] to render a volume data set by first shearing the the volume slice to make ray traversal trivial, and then warping the intermediate image to the final image. Run-length encoding is also applied to efficiently skip transparent voxels and opaque image pixels to further speed up the rendering process.

Texture-Mapping Proposed by Gelder et al. [GK96] to take advantage of the hardware 3D texture map support from some graphics workstations by converting the volume data into 3D textures and then performing compositing in the texture space.

Opacity-Weighting Wittenbrink et al. [WMG98] argued that shading/classification before interpolation may produce artifacts in some case and therefore proposed an *opacity-weighted* color interpolation scheme to address this problem.

Compression Domain A very different school of thoughts explores the possibility of rendering directly from the frequency/transformed domain, as shown in [DNR90, Lev92, Mal93, TL93, CYH+97, Gro96]. These methods, although are asymptotically faster, usually fail to produce high quality images. We will discuss more on these algorithms later.

Acceleration techniques There are many acceleration techniques available to speed up the volume rendering, as it is usually a lengthy process.

Raycasting Many techniques were developed for raycasting, and some of them in fact can be used in other contexts, as can be seen in the following discussion. Levoy [Lev90a] assumed a pre-shading/pre-classification model. A data set is decomposed into an octree according to the opacity values. By skipping those "empty" regions, his method speeds up the ray traversal process, this is so-called *presence acceleration*. The second optimization technique, called *early ray termination* or *α-termination*, is applicable to a front-to-back compositing where as soon as a ray corresponding to an image pixel accumulates enough opacity–close enough to 1 or to some pre-specified threshold, then the traversal for that ray stops. His later paper [Lev90b] further exploited the homogeneity of the data sets. The rendered image is first partitioned into squares and for each square one ray is cast into the data set. A square is recursively partitioned into smaller squares if the image pixel value difference among the original squares is bigger than a threshold. The final image is formed by interpolations/re-sampling from the results of these sparse rays on the image space. Danskin et al. [DH92] generalize the idea of presence acceleration to *homogeneity acceleration* by building a multi-resolution hierarchy and performing traversal at the highest possible level to skip homogeneous regions. Another generalization is called *β-acceleration* which basically sample less and less points as the opacity value accumulates by similarly moving the traversal up to higher levels in the hierarchy. Compression domain rendering, as mentioned before, can provide asymptotically faster rendering. However, the generated XRay-like images keep them from being used in practice. A different way to speed up the rendering, proposed by Yang et al. [YC01b] focused on minimizing the *end-to-end* rendering time, which mainly includes the I/O time and the rendering time. The main argument is, as data sets get bigger and bigger, the data loading time becomes non-negligible. Data sets are partitioned into sub-blocks and two threads are used to mask the disk I/O time with the computation time. By observing that most data sets may have many empty cells outside of region of interest, Wan et al. [WKB99] run-length encoded the boundary cells in a compact way to skip empty region and sped up the ray intersection process for each viable ray. Knittel's UltraVis system [Kni00] made use of MMX and SSE capabilities provided by Pentium III processors and employed a careful data arrangement to achieve high hit ratio of the cache to speed up rendering significantly. Pfister et al.'s VolumePro chip [PHK+99] built a special-purposed chip capable of doing raycasting in hardware.

Splatting As mentioned before, Mueller et al. [MY96] applied the ray-driven approach to make the implementation of early ray termination possible, which is known as a very effective speedup mechanism for raycasting. Another idea proposed by Mueller et at. [MSHC99b] is to build so called *occlusion map* where each splat, before being projected to the image plane, first checks to see whether its projected region falls within a region whose occlusion map is saturated or not; if so, skip this splatting, otherwise proceed as usual and corresponding occlusion map is updated accordingly.

Point-Based Rendering In recent years, there is a trend in the graphics world to use point as a rendering primitive to perform surface rendering, instead of using the traditional polygonal models. It was originally proposed by Levoy et al. [LW85] in 1985, and later, it is observed that for a huge polygonal data set, there seems to be no need to keep the connectivity information because most of the projected triangles will be smaller than an image pixel size. Instead, a multi-resolution hierarchy of points is built during the

preprocessing time, and at run time, only the level that is comparable to the image resolution is retrieved for display. Amazingly interactive speed can be achieved, as can be seen from the work done by Rusinkiewicz et al.'s Qsplat [RL00] and Pfister et al's Surfel [PZvBG00]. Although this work is mainly used to address traditional polygonal rendering, by treating each each voxel in the volume data sets as a sample point and through the use of *α-blending*, presumably volume rendering can also be done in a similar fashion. The only obstacle left is the intrinsic problem of ordering, which is most important factor in volume rendering to guarantee correct results. Much work along this line may be able to narrow the traditional gap between surface rendering and volume rendering.

IBR-Assisted Rendering Up to now the main discussion is focusing on generating static images from a volume data set. If an animation or navigational view of a volume data set is desired, then apparently there may be significantly potential coherence that one can explore. Mueller et al. [MSHC99a] and Baoquan [Che01] have done some researches along this line.

2.2.2 Rendering of Irregular Volume Data

Most of the algorithms for rendering regular grids can not be readily applied to irregular grids, because the lack of regularity (see Speray et al. [SK90]). However, most of the methods shown here still bear much resemblance to the previous methods used for regular volume rendering, as some of them can be generalized to handle irregular grids as well. Perhaps the simplest approach is to "re-sample" a irregular grid into a regular grid, as done by Wilhelms et al. [WCA^{+}90] and Fruhauf [Fru94]. But to accommodate the finest details, it may result in an exceedingly large number of sample points, not to mention that there is still the possibility of high loss of accuracy. For curvilinear grids, Fruhauf [Fru94] also proposed a method to "bend" the cast rays according to the deformation of the grids, by performing a raycasting in the computational domain, and mapping the sampled locations into the spatial domain. However, this approach cannot deal with general unstructured grids. In general, there are three types of methods that can render general irregular grids: Projection (including Splatting), Plane-Sweep and Image-Order algorithms.

Projection (Object-Order Algorithms) Two *projection-based* methods were proposed independently by Max et al. [MHC90] and Shirley et al. [ST90]. The basic ideas are similar: a tetrahedron or a polyhedron is projected onto the image plane. In [MHC90], they assume the grid (cell) to be a polyhedron, and by scan-converting each polyhedron twice, one can get the z (depth) values and scalar (color or intensity) values for the *front* face and *back* face of the polyhedron. For each pixel, an integration process can be used to obtain the pixel color contributed by this polyhedron. Since this integration can be solved analytically, therefore most of this calculation can be done beforehand. However, to make the integration solvable, the color must be set constant throughout the integration, making it less "real" in general. Shirley et al. [ST90] proposed an even less accurate version, where hardware-assistance is available. And to reduce the number of possible outcomes of projecting a polyhedron, if a cell is not a tetrahedron, it is first decomposed into tetrahedra. For example, a hexahedron can be decomposed into five tetrahedra. Notice the decomposition should be done carefully to avoid possible cracking (an overlap or gap between cells). According to the viewing direction, the possible projection shape of a tetrahedron is then *classified*, as shown in Figure 1 (a). Most

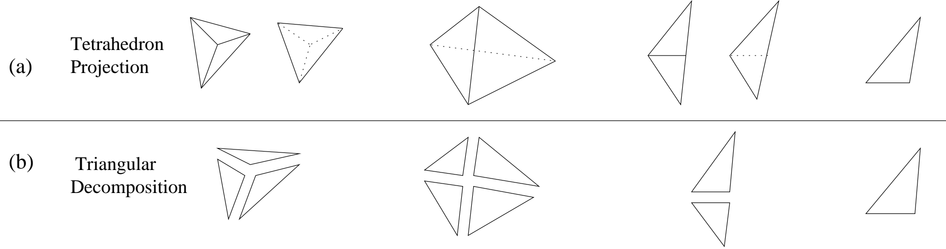

Figure 1: *Decomposition into triangles for each shape*

of the time along the viewing direction, there exists a "thickest" vertex. For those vertices around the silhouette, because of their zero thickness, their opacity values are set to zero. For the thickest vertex, an approximated integration, again by assuming a constant color, is performed to get its approximated color, as well as its opacity. After all the color and opacity information is calculated for each vertex, according to where the thickest point is, the projection is decomposed into several triangles, as shown in Figure 1 (b), and each of them has all of its three vertices classified and shaded. These triangles can now be treated as basic graphics primitives and sent to graphics engines for standard shading, where a presumably Gouraud-shaded RGBA model can then be easily used for hardware-assisted scan-conversion and compositing. Wittenbrink's CellFast [Wit99] based on Shirley et al.'s work and added several optimizations (visibility sorting, tetrahedra culling, and so on) to achieve nearly real-time rendering speed on moderately sized irregular grids. Mao et al. [MHK95] rendered curvilinear grids using Splatting. The basic idea is, for a fixed image resolution, enough sample points could be created from the original data set (a Poisson distribution is used when selecting points) for splatting, and these points should be dense enough so that from any viewing direction no holes should be spotted. Later Mao [Mao96] extended this work to handle tetrahedral mesh as well. However, it is not clear how the error analysis could be done and there is no theoretical bound on how many sample points should be created, given the general scenarios where the cell sizes may vary tremendously. To guarantee a correct result, most of these "projective" methods needs to project the objects (cells or points) "in order". This is so called "visibility ordering" problem has attracted a great deal of study. Based on the work by Edelsbrunner [Ede90], Max et al. [MHC90] and Williams [Wil92] developed similar methods on this. The latter one, called "MPVO", was extended to an "XMPVO" algorithm by Silva et al. [SMW98]. Comba et al. [CKM+99] proposed a follow-up work, called "BSP-XMPVO", to further improve the performance for visibility ordering.

Plane-Sweep Another completely different approach is to use so called *plane sweep* algorithm coming from computational geometry. The basic idea is to sort the vertices (or other primitives), according to their x, y, or z coordinates. A bounding box is then built from the previous min/max values. A data-structure, called *sweep line (plane) status*, are used to record some "active" objects (such as edges and faces) currently intersecting the sweeping line (plane). Another term is called an *event* (such as the occurrence a new vertex) and it represents the time or place where the sweep line status needs to be updated. As the sweep line (or plane) sweeps through the whole data set, whenever an event is encountered, using the pre-stored adjacency information (for edges, triangles, or tetrahedra, and so on) one can update the active objects accordingly. For example, given a collection of 2D segments, finding

out if there exists any intersection among them, and if so, how many of them, is a typical such problem that can be efficiently solved using a plan sweep algorithm, which takes advantage of the coherence between sweeping lines.

Giertsen's First Sweep Giertsen [Gie92] first applied this idea to the rendering of irregular grids. All the vertices of a data set are first transformed so that the viewing direction is aligned with the Z axis. A sorting is then performed to sort all the vertices by their y coordinates. An ACL (active cell list) is used to store all the cells (tetrahedra) that are currently intersecting with the sweeping plane. A sweeping plane that is parallel to the $X - Z$ plane is used to sweep the data set. According to whether a new vertex or a scan-line in the image plane is encountered, the algorithm does different things. In the first case (event), the ACL data structure is updated to insert/remove the cells incident to this vertex. In the second case (event), since the intersections of the grids with the sweeping plane can be derived, a 2D raycasting is then performed to get the result for this scan-line. However when processing a scan-line, a fixed-sized two-dimensional scan-plane buffer is used. All the intersecting polygons are broken into triangles, and further broken into segments in such a way that each segment is associated with only one interval from the scan-plane buffer. This means along the Z direction some accuracy is sacrificed but a straightforward raycasting scheme can be used.

Lazy Sweep Inspired by the Giertsen's work, Silva et al's *Lazy Sweep* [SM97] proposed several techniques to optimize this approach. No global transformation is needed, and only the vertices on the boundary of the grids are sorted. During the sweeping, whenever a new vertex is encountered, for each cell that contains this vertex, its adjacency information can be used to retrieve all the vertices incident to this cell and inserted to the sorted vertex list (usually a priority queue). According to the y coordinate of this new vertex, we know whether it is a topmost vertex, a bottommost vertex, or neither, then we can update the active data-structure (called AEdge here, meaning the active edges that intersect with the sweeping plane) accordingly. The rendering for the scan-line again employs a 2D sweep algorithm, using a sweeping line parallel to the z axis. It updates the event queue whenever a new vertex is encountered and stops at each scan-line to render a particular image pixel. Along each scan-line, a ray is cast, and interpolation together with compositing are performed at each intersection point, Therefore this algorithm, although more complicated, produces a better result. Moreover, this approach in general requires much less memory because most of the geometry is discovered/constructed/deleted on the fly. Notice in these two methods, by making use of the coherence between scan-planes and scan-lines, the calculation of intersection points can be done incrementally, not from scratch, thus making these approaches attractive.

Hardware-Assisted Yagel et al. [YRL+96] proposed a different sweeping scheme. Their sweeping plane is parallel to the image plane. After intersecting the data set with the sweeping plane, graphics hardware can be used to to scan-convert the intersecting polygons into a single "slicing image", and this image is composited with the rendered image accumulated so far. Sorting of all the vertices along the z direction can create a high quality image but may require too many scan-conversions and compositing work. Instead, an approximation may be made by stepping a constant Δz size, and performing the above work, thus making a tradeoff between the size of Δz and the image quality.

Without the need of storing the adjacency information, this scheme is simple to implement but requires accesses to high-performance workstations with huge memory, if targeting for high quality images.

ZSweep Farias et al. [FMSW00] implemented a full-blown plane sweep algorithm whose sweeping plane is also parallel to the image plane. Intersecting cells are inserted for compositing according to their correct order with respect to the image plane. To further reduce the memory requirement for storing all the intersecting cells, a "windowed" approach is used to limit the compositing scope. This so called *ZSweep* approach gives better performance than *Lazy Sweep* at the expense of consuming more memory, when facing bigger data sets and/or higher image resolutions.

Image-Order This type of approaches applies basically the raycasting idea to cast rays into the data sets. However, due to the irregularity of the data sets, the ray traversal part bears much higher complexity.

Garrity's Algorithm Garrity [Gar90] followed this image order paradigm, where a pre-processing step was performed in the object space. First the data set is decomposed intro tetrahedra, if necessary, and then pre-processed to find the adjacency information, such as the vertices for a given tetrahedron and tetrahedra for a given triangle. Next the boundary triangles are identified, which are the triangles that are used by only one tetrahedron. To perform raycasting, each ray needs to identify its "first cell", when it first shoots from the image plane or when it "re-enters" the mesh through a boundary triangle after previously exiting from another boundary triangle. Some spatial indexing is used to reduce the searching range for this. This *first cell* problem attracts a large amount of research, and we refer the interested readers to [SM97] for an in-depth discussion. Once the problem of first cell is resolved, the ray traversal continues by using the adjacent information to retrieve the next tetrahedra to proceed until the ray touches another boundary triangle, where the first cell finder is consulted again, or the ray completes its traversal.

Hong et al.'s Algorithm Hong et al. [HK99] applied Garrity's idea, and solved the *first cell* problem by back projecting each "visible" boundary face to find out all penetrating rays. Here "visible" boundary face means a boundary face with its normal pointing at the opposite direction as the viewing direction. When a ray can intersect with multiple visible faces, all the intersection points are sorted and chained according to their *depths* or *distance* to the image plane. If the current intersecting face is not a boundary face, then raycasting proceeds by finding the next intersecting face and calculating the contribution made from the current intersecting face to the next. Due to the properties of curvilinear grids, there are only a few cases to consider on how to find out the next intersecting face. On the other hand, if the current intersecting face is a boundary face, the previously sorted chain of intersection points is used to retrieve the next intersecting face thus continuing the ray traversal.

Bunyk et al.'s Algorithm Bunyk et al. [BKS97] combined the advantages of the previous two methods: the simplicity of tetrahedron cells for finding the next cell, and the back projecting idea to solve the *first cell* problem. For calculating the contribution made between the current face and next face, an approximation is used to simulate a very simple lighting model.

Gatun Based on Bunyk et al.'s work, Yang et al. [YMC00] modified it in that it is also able to perform the unit-distance sampling. The enabling technique is the ability to do interpolation within tetrahedra accurately and efficiently (see Appendix). Furthermore, [YMC00] is mainly designed to be integrated with a lossless irregular volume grid compression algorithm therefore some other modifications are also made. This is one of the main themes of this proposal, so we will discuss it in more details later.

2.3 Volume Simplification

Many of the ideas for volume simplification came directly from surface simplification algorithms, therefore the latter ones are also briefly covered for completeness. Speaking of volume data, the main concern here is about tetrahedral grids, which is the target domain for our proposal.

2.3.1 Surface simplification

The given input data sets for surface simplifications consist of polygons, and in many contexts, only triangles. The output will be another polygonal data sets, with presumably much less number of polygons or triangles, but still preserving the original shape approximately. Essentially this is a tradeoff between allowed quality/accuracy and consumed storage requirement.

Flattening and Straightening Schroeder et al. [SZL92] described a method which tests the neighboring properties of a vertex. If it is not on the boundary and its adjacent faces form a "flat enough" face, then this vertex is removed; otherwise if it is a boundary vertex and it is close enough to the line segment defined by its two neighboring vertices, it can also be removed. The resulting mesh is then patched or *re-triangulated* and fed into the next iteration for further possible simplification. Hinker et al. [HH92] and Kalvin et al. [KT96] used a similar approach by merging faces that are flat enough or edges that are straight enough and then applied a new triangulation, with the latter one to be in a better time complexity and a better error control mechanism.

Re-Tiling According to the user specified number of vertices to be presented in the final mesh, Turk [Tur92]'s re-tiling method first uniformly added these number of vertices to the old mesh, and let each of such new vertices repel each other to reach in an "equilibrium" state, then applied a *mutual tessellation* scheme to build a finer tessellation on top of the old tiling. Old vertices are then removed one by one, accompanied by re-triangulations along the way.

Wavelet Eck et al. [EDD$^+$95] first constructed a continuous parameterization of the original mesh, then re-mesh/re-sample the original mesh to create a closely approximated mesh. Wavelet multi-resolution technique is then applied to the latter mesh to get the corresponding approximated mesh according to the user specified error tolerance. However, the first step of re-sampling may introduce some error, especially when facing some "ill-formed" meshes, such as ones having creases at arbitrary angles.

Vertex Clustering Rossignac et al. [RB93] introduced a different simplification method based on *vertex clustering*. A bounding box is found for a given data set, then the whole bounding box is partitioned into blocks. All the vertices within a block collapse into one "representative vertex". Triangles are then formed from these representative vertices. This method is very

general but the topology of the original mesh may be changed, and even a non-manifold mesh may be resulted.

Mesh Optimization Hoppe et al. [HDD+93] associated each mesh with a pre-defined energy. The energy function penalizes on several terms including the number of vertices, the distance from vertices to the mesh, and the length of edges in the triangulation. The surface bearing the minimal energy is achieved through insertion/removal of vertices, collapsing/splitting/swapping of edges. Although slow, this method tend to create a very good approximation to the original mesh in terms of quality.

Progressive Mesh Hoppe [Hop96] later restricted the allowed operations for simplification to just *edge collapse* operations to speed up the process while claiming the resulting mesh is at least as good as the work in [HDD+93] or even better. The idea of building a simplification hierarchy was also proposed and used later for other work, including its generalization to tetrahedral grid simplification.

Simplification Envelope Cohen et al. [CVM+96] proposed to construct a *simplification envelope* to guarantee the difference between the approximation and the original is always within a user specified ϵ, and the simplification is done within that envelope.

View-Dependent and Out-of-Core Simplification In addition to the general ideas about surface simplification, there have also been some works which can adapt the simplification to the viewing parameters or perform the simplification when the required memory exceeds the physical memory capacity. Xia et al [XV96]'s method is to build a simplification hierarchy in a bottom-up way by merging collapsible edges. At run time, viewing parameters are used to select *active* vertices, therefore their related triangles, thus simplifying the rest vertices and triangles. Hoppe [Hop97] proposed a similar idea to build the hierarchy in a top-down manner by selectively refining regions of interest. His scheme is more general and less constrained, therefore the resulted hierarchy is usually much shorter. Jihad et al. [ESV99]' approach, also similar to [XV96], added the capability to deal with topological change of the original data set by considering a subset of the *Delaunay edges*, and some dependency information. This was originally used to avoid *fold-overs* during the simplification process, but is found to help the run-time memory access efficiency significantly when nearby related vertices are stored. Based on this work, Jihad et al. [ESC00] extended it to perform out-of-core simplification as well by partitioning the mesh into *sub-meshes* carefully (still preserving the simplification semantics) and then selectively perform the simplification on only a number of sub-meshes each time. A different out-of-core simplification used by Lindstrom [Lin00] applied the vertex clustering approach but with a different strategy about the placement of the representative vertex in each cluster. By assuming that a simplified mesh can be held in memory, this algorithm just requires one pass of scanning of all the triangles and simplifies each of them on the fly.

2.3.2 Volume Simplification

As mentioned before, the main target data sets concerned here are tetrahedral data sets, and they can be easily converted otherwise.

Renze et al. Using the idea of vertex removal on tetrahedral grid, Renze et al. [RO96] proposed to try to patch the hole by a local *Delaunay tetrahedralization*. If such a tetrahedralization does not exist, the vertex will not be removed.

Trotts et al. Trotts et al. [THJW98] applied a piece-wise linear spline function that is defined over the data values on the input tetrahedral mesh as the basis of the error metric for volume simplification. The algorithm associates each tetrahedron with such an error metric and favors the removal of those tetrahedra that causes the least change in the spline function. Removing a tetrahedron is carried out by a sequence of edge collapses operations.

Gelder et al. Gelder et al. [GVW99] proposed a less computation-intensive approach for volume simplification, which aims to minimize the density or "mass" change due to an edge collapse operation. Boundary vertices are associated with an extra geometry-related error metric, in addition to the so called *data-based* error metric required for internal vertices. We will explain this method in more details since it is used as a static simplification algorithm in our system.

Staadt et al. Staadt et al.'s *progressive tetrahedralization* [OGS98] adopted different error metric and provided methods to check for self intersections due to boundary edge collapses.

Cignoni et al. Cignoni et al. [CCM+00] also used the edge collapse primitive while providing a more detailed error control mechanism over the *domain error*, i.e., the geometrical difference between the old mesh and approximated mesh, and the *field error*, i.e., the scalar field error associated with each vertex, instead of using a weighted scheme like others.

Farias et al. Farias et al. [FMSW00] proposed an somehow different tetrahedral mesh simplification algorithm which was based on the idea similar to vertex clustering. It built a kd-tree from the original data set, and in each kd-tree area, spheres are used to group vertices together and all vertices within a sphere are represented by the center of the sphere. The radius of the sphere is determined by the user-specified number of vertices retained in the approximated mesh. Boundary vertices require special care to handle possible concavity of meshes.

2.4 Iso-Surface Extraction

As mentioned before, an iso-surface extraction algorithm extracts surfaces from a volumetric data set when given an interested iso-value or iso-density. The given data sets can be either regular or irregular. There are also many different algorithms that have been proposed. We first give a general introduction, then discuss several acceleration techniques.

2.4.1 General Introduction

Marching Cubes Lorensen et al. [LC87] pioneered the iso-surface extraction research field with the proposal of *Marching Cubes* algorithm on regular data sets. A volume data set is viewed as a collection of slices, where cells are formed between two adjacent slices. Each cell is composed of eight vertices, and each of which has an associated density or scalar value. Given an iso-value, each cell is checked to see if it "intersects" or "contains" the iso-value; if so, an linear interpolation scheme is used to find the intersecting vertices along the intersecting edges, and the resulting triangles are sent for normal polygonal rendering. Depending on whether a vertex is "inside" or "outside" of the iso-surface, there are $2^8 = 256$ cases. However,

using symmetry and complementation, there are only 14 possible cases to consider where an iso-value can intersect with a cell, and therefore an efficient table lookup approach can be used. Normal vector estimation is done by linearly interpolating between the gradient vectors associated these vertices, where each gradient vector is obtained using the *central difference* method mentioned before.

Disambiguation Marching cubes algorithm can generate high quality surface, however, there is a well known *ambiguity* problem, which means for some cells, there are more than one possible cases where the iso-surface can intersect with these cells. Several methods are proposed to resolve this issue. Nielson et al.'s *asymptotic decider* [NH91] method performs tests in those ambiguous cases to see if some of the problematic vertices should be within the same region or not by checking them against some hyperbolas, while Montani et al. [MSS94b] modified the original lookup table proposed by Lorensen et al. by using different triangulations to get rid of ambiguity.

Marching Tetrahedra *Marching Tetrahedra* is a similar idea to marching cubes, but instead of operating on cubes, it operates on tetrahedra. There are two possible reasons to use marching tetrahedra. First, the original data sets are tetrahedral data sets. Second, it is one of the simplest methods to avoid ambiguity by decomposing an originally voxel data set into a tetrahedral data set. A cube can be decomposed into five tetrahedra, as done by Carneiro et al. [CSK96], into six tetrahedra, or into twelve tetrahedra, as done by Chan et al. [CP98], each with different concerns on the number, pattern (biased or not), or shape (too thin, too wide, etc.) of the tetrahedra generated.

Triangle Reduction Another concern of using marching cubes algorithm is its excessive number of triangles generated. The situation is usually worsened when marching tetrahedra is used. Shu et al.'s *adaptive marching cubes* [SZK95] algorithm tackled this issue by first running the original marching cubes algorithm on the coarsest resolution of cubes, and if the surface within a cell is not flat enough then the cell is divided and dealt with recursively. Notice this approach may create *cracks* on the faces where two different resolution of cubes meet, and therefore a patching of these cracks is usually necessary. Montani et al. [MSS94a] increased the chances of merging the triangles generated by adjacent cells through the version of *Discretized Marching Cubes* algorithm, which restricts the intersecting location to be always at the middle point of each intersecting edge. This basically provides a more "friendly" configuration for the ensuing simplification algorithm to work on. Of course a more naive approach would be to apply those surface simplification algorithms mentioned before to decimate the number of output triangles from an iso-surface extraction algorithm.

Dividing Cubes Cline et al. [CLL+88] had a different perspective to deal with the issue of excessive number of triangles. Instead of outputting triangles, those intersecting cubes are recursively divided until the their sizes are not bigger than a pixel size. Then each intersecting cell at the final resolution is approximated as a surface point, located at the center of that divided cell. Its normal vector can be interpolated from the original gradient vectors at the original resolution. Then these surface points can be sent to the graphics pipeline for rendering. Notice this is very similar to the point-based rendering we mentioned previously, and the basic observation is that the number of triangles generated by the marching cubes algorithm can greatly outnumber the number of pixels on the screen. Rendering more triangles

simply will not convey any new information. If the target surface is opaque, then the order that these points send to the graphics pipeline does not matter, because the standard *z-buffer* algorithm can take care of the visibility problem; otherwise, an correct order of sending these surface points to the graphics pipeline should be maintained.

2.4.2 Acceleration Techniques

There are essentially two phases in the iso-surface generation process. The first phase is the search for intersecting cells. The second phase is the generation of triangles. While the second phase can be sped up by using a table-lookup, and leaves little room for further improvement, the first phase can be accelerated significantly by using different data structures and/or searching algorithms. There are basically three approaches: Space-based, Range-based and Surface-based. The first approach performs decomposition based on the spatial information while the second approach on the scalar value associated with each cell. The last approach tries to grow iso-surface using some adjacency information from a much smaller *seed-set* which is found in the preprocessing time.

Space-Based As mentioned before, this approach performs decomposition based on the spatial information, although in the later searching stage it is still based on the data values stored within the constructed spatial structure. Wilhelms et al. [WG92] proposed an octree decomposition method for regular grid. Each node in the octree records the *min* and *max* within it. A given iso-value is checked against the coarsest level in the octree and recursively sent to finer levels if necessary. However the real efficiency, usually defined by how much percentage of cells get touched, is very data-dependent and this method cannot be readily applied to irregular grids.

Range-Based This approach builds the data structures based on the scalar values associated with each cell, and the cell here can be cubic cells or tetrahedral cells.

Span Filtering Gallagher [Gal91] divided the range of data values into sub-ranges, called *buckets*. For each cell, we identify its starting bucket and well as its *span*, or the number of buckets its range intersects with. Cells with the same span are grouped together while within each group sub-groups are formed according to their starting cell. Span numbers greater than a threshold can be grouped together to save storage. Given a query, all the span groups are traversed and depends on the span that group represents, a number of more buckets will be traversed accordingly.

Span-Space (Min-Max Graph) Shen et al. [LSJ96] proposed a very different approach to represent each cell by a point in a $2D$ plane by using the y-axis to represent the max value of a cell and the x-axis the min value of a cell. Each result to a iso-value query corresponds to a square region with its lower right corner touching the line defined by the equation of $x = y$. A *kd-tree* is used to build a hierarchy based on those points. At run time to answer a query the kd-tree is traversed to find the right and lower boundary and those points falling within this region are reported. Shen et al. [SHLJ96] later improved the searching time complexity further by using a uniform partition in the area of the $2D$ plane defined by $x < y$, the only area where all the cells of a data set can fall into.

Interval Tree Cignoni et al. [CMM+97] demonstrated how to use the concept of *interval tree* to answer an iso-surface query, which is essentially a *stabbing query* in the field of *computational geometry*, to achieve the optimal time complexity for searching. Chiang

et al. [CS97] applied this idea but changed the branching factor to make the size of each node in the interval tree no bigger than a disk block to significantly reduce the number of disk I/Os and the memory required for query processing. Later Chiang et al. [YCS98] extended their work to do *out-of-core* iso-surface extraction as well by partitioning the original grids into *meta-cells* and used a kd-tree decomposition to make sure each meta-cell can be fit into the memory.

Surface-Based: (Seed-Growing) In this approach, a *seed-set*, is first identified in the preprocessing time. Seed-set has the property that every iso-surface must intersect with at least one cell in the seed-set. At run time, given a query, the intersecting cells from the seed-set are located and used to propagate to find all other intersecting cells by exploring the adjacent cells.

Extrema Graph Itoh et al. [IK95] proposed a method to find the seed-set by building an *extrema graph* and two sorted lists from the boundary cells. The extrema graph is composed of extrema points and arcs. An extrema point is a cell whose neighbors are all smaller or all bigger in terms of their scalar values, and closer extrema points are connected by arcs. For closed iso-surfaces, which means iso-surfaces that will not intersect any boundary cells, it must intersect with the extrema graph. However, an open iso-surface will intersect with the boundary cells. For this reason, boundary cells are sorted into two lists, by their minimal and maximal values. For each iso-value, first each arc is tested for an intersection, and the intersecting cell for each potentially intersecting arc is found and served as a seed for propagation. Boundary cell lists are also tested to find out the boundary seed cells for propagation.

Volume Thinning Itoh et al. [IYK96] later improved their previous work by *volume thinning*, which is borrowed from the image processing world. Essentially all the cells that their removal will not affect the local connectivity will be eliminated, with the exception that all the extrema points should be preserved. Layers surrounding *voids*, which means the zero-density areas, will be "pricked" to further reduce the seed-set size. This method can get rid of the boundary cells from the seed-set thus significantly reduces the seed-set size.

Interval Tree on Seed Set Bajaj et al [BPS96] combined the idea of interval tree and seed growing by building the interval tree structure only on the seed-set thus further reducing the search domain. Once the intersecting cells from the seed-set are quickly located, they can be used to propagated to find the desired iso-surfaces.

3 Compression Domain Rendering of Regular Volume Data

Volume compression has become a main technique to reduce the storage requirement facing today's huge volume data sets. On the other hand, visualizing these data sets is why these data sets were generated/obtained in the first place. In addition to the naive approach of *decompression before rendering*, there are also several other strategies to bridge the gap between compression and visualization, such as *on-the-fly rendering during decompression*, *on-the-fly decompression during rendering*, and *rendering directly from the compression domain*. We will detail the last one in this section and leave *on-the-fly rendering during decompression* to the next section. Except for the

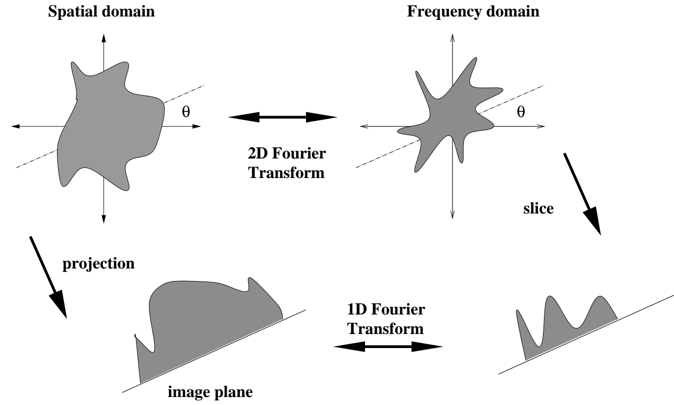

Figure 2: *A 2D example for Fourier Projection-Slice Theorem*

naive approach, all others are meant to reduce the rendering latency, data loading time, or memory footprint size, with the last strategy being possibly the most efficient. It is provably faster, in terms of time complexity, but it suffers from a high quality loss. We have proposed a method to improve the image quality generated through this method in some cases. The compression algorithm used here is lossy and the involved data sets are regular volume grids.

3.1 Previous Work

There have been some previous works on performing renderings directly from the compression domain.

3.1.1 Fourier Rendering

Dunne et al. [DNR90] and Malzbender [Mal93] independently developed a way to do volume rendering directly in the Fourier domain. The basic idea is to use the FPST (Fourier Projection-Slice Theorem). First a 3D DFT is performed to transform a data set from the spatial domain into the Fourier domain. Then according to the desired viewing direction and the image plane, the projection of the volume onto this image plane can be obtained by performing an inverse 2D Fourier Transform on the slice extracted from the Fourier domain, with the slice passing through the center in the Fourier domain and parallel to the image plane in the spatial domain, as shown in Figure 2 for a 2D example. To improve the image quality, Levoy [Lev92] proposed a way to include depth cuing and directional shading into the rendering but four spectra are needed instead of one. Totsuka et al. [TL93] observed that depth cuing can be directly implemented in the frequency domain as a differentiation while directional shading is implemented in the frequency domain as a multiplication, and therefore avoided the increase on the number of spectra. However, due to the lack of occlusion, the images generated using the above approach are still not comparable with those ones generated by the traditional methods. Chiueh et al. [CYH+97] proposed a method to address this problem. This will be covered in more details in Section 3.

3.1.2 Vector Quantization

From the previous work in vector quantization, Ning et al. [NH93] observed that since all the sub-blocks are retrieved from the code-book, and the number of sub-blocks is relatively small, so we might as well first render all the sub-blocks in the code-book according to the given viewing direction and then during the real rendering process, instead of retrieving the sub-block for further rendering, the "pre-rendered" sub-image is retrieved and composited into the final image rendered so far. This results in a very fast rendering, but unfortunately the approximation by the sub-blocks in code-book often leads to "blockiness" in the rendered images.

3.1.3 Wavelet

There have been some works on performing rendering directly from the wavelet domain. In Westermann [Wes94], rendering is done by directly traversing the wavelet domain coefficients. These coefficients are carefully arranged and laid out, together with a possible thresholding applied before the rendering. During the ray traversal process, and at each sample point along the ray, the coefficient structure is traversed, from the finest level to the coarsest level, to locate those coefficients which really contribute to the given sample point and their contribution is calculated. The traversal's stepping size is also adjusted according to the finest level found so that the traversal can be sped up when stepping through a homogeneous area. This implementation, although indeed performs the rendering in the wavelet domain, is very slow. Nevertheless it demonstrates the possibility of doing rendering directly within the wavelet domain when equipped with a very limited amount of memory. Gross [Gro96] and Gross et al. [GLS99] also tried to render images from the wavelet domain directly, but resulted in the following two situations. First, they use an approximated rendering formula which can only produce X-Ray like images. Second, to have the occlusion effect, a more accurate rendering formula is used but then it requires very expensive computation for such an approximation, which makes it almost infeasible.

3.2 Volume Data Compression

There have been many past researches on lossy volume compression for regular grids and most of the algorithms involve a transform from the original domain (or so called *spatial domain* or *time domain*) to another domain (such as *frequency domain*), where it is much easier to filter out information that is less important through the method of *quantization*. After quantization is done, the original coefficients are usually traversed in a "zig-zag" way so that more important coefficients, in terms of magnitudes, can be represented earlier. This linearized sequence is then fed into some entropy encoding scheme, such as Huffman coding or arithmetic coding, to losslessly compress the quantized result. The whole process is shown in Figure 3 (a), where a 3D Fourier transform is used. The reverse process, as shown in Figure 3 (b), starts from an entropy decoding process, followed by a possible de-quantization step (depends on how quantization was done) and an inverse 3D Fourier transform. Throughout this section, we will only focus on Fourier transform, because via the use of FPST, compression domain rendering is made possible. Its relative, cosine transform, although in general offers a better energy compaction result, which can potentially lead to a better compression efficiency with operations only on the real domain, fails to provide the possibility to render directly in the compression domain. In practice, we use Hartley transform to replace the Fourier transform, because not only it also operates completely on the real domain, but also it preserves the same property which lends compression domain rendering possible.

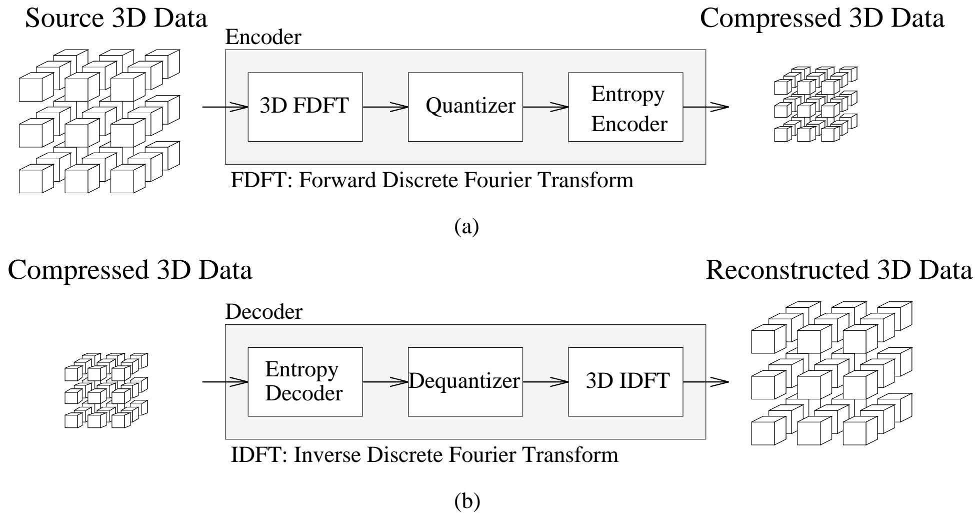

Figure 3: *The data flows of the proposed volume (a) compression and (b) decompression algorithms.*

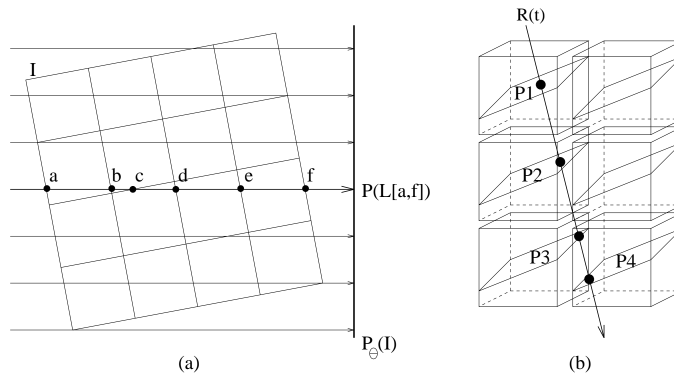

Figure 4: *Summation of projection sums from (a) each block in the 2D case and (b) from each sub-block in the 3D case along the viewing ray.*

3.3 Rendering from Compressed Volume

Since applying FPST to the whole volume fails to provide the *self-occlusion* effect, we divide data sets into equal-sized sub-blocks. It is clear that from Figure 4 (a), a 2D example, where to get of $P(L[a,f])$, the projection value along the line from a to f, is equivalent to getting the sum of $P(L[a,b])$, $P(L[b,c])$, $P(L[c,d])$, $P(L[d,e])$ and $P(L[e,f])$. And the same idea can be generalized to the 3D case, which is shown in Figure 4 (b). Here planes P_1, P_2, P_3 and P_4 represent the projection plane in each sub-block, as explained previously about how FPST is applied. Self-occlusion effect can be simulated by compositing all sub-blocks with each having its own color and opacity.

This sub-blocks approach, although can be used to simulate the self-occlusion effect, suffers from three drawbacks. The first drawback is the undesired *ghost* or *aliasing* effect when applying FPST. And the reason is as follows. A set of discrete samples in the frequency domain (DFT) means a periodic replication of the original signal in the spatial domain. A re-sampling or interpolation, or equivalently a convolution in the frequency domain means a multiplication in the spatial domain.

An ideal choice is to do a convolution with the *sinc* function in the frequency domain, whose inverse transform is a unit box function in the spatial domain, and a multiplication with this box function means a perfect reconstruction. However, the *sinc* function, which can be implemented as a filter, has an infinite support. Either a *windowed* or truncated version of it must be implemented or different filters must be used, and therefore errors are introduced. Let us call the transformed function in the spatial domain the "multiplication kernel", corresponding to its convolution used in the frequency domain. For a convolution filter, if its multiplication kernel is *too big*, the replicated signal may be erroneously included, leading to the *ghost* effects, which happens mostly for some non-orthographic viewing angles. On the other hand, if its multiplication kernel is *too small*, the correct original may get mistakenly suppressed, leading to the aliasing effects. Furthermore, since these artifacts tend to occur along the boundary of each projection image, they become more noticeable when applying the FPST to the each sub-blocks and compositing them, than applying the FPST to the whole volume. To reduce these artifacts, in our implementation, we also *zero-pad* our sub-blocks in spatial domain to provide a "protective" region against undesired artifacts.

The second drawback comes from the color and opacity approximation for each sub-block. From FPST, only a projection image of each sub-block can be derived. Ideally what we need is a front-to-back (or back-to-front) composited result of the color and opacity values of each sub-block formed by raycasting through it. It can be shown that if a ray take n sample points within a sub-block, the aggregated color and opacity values can be derived from equations 1 and 2 as:

$$C_{sub-block} = \sum_{i=1}^{n} [C_i * O_i * \prod_{j=1}^{i-1}(1 - O_j)] \tag{3}$$

$$O_{sub-block} = 1 - \prod_{i=1}^{n} (1 - O_i) \tag{4}$$

where C_i and O_i represent the color and opacity values at the point i respectively. If the opacity transfer function assumes such a negative exponential form:

$$O_i = 1 - e^{-K * D_i} \tag{5}$$

where D_i is the density value at the point i, then it can be shown that $O_{sub-block}$ can be computed exactly from $\sum_{i=1}^{n} D_i$ as follows:

$$O_{sub-block} = 1 - e^{-K * \sum_{i=1}^{n} D_i} \tag{6}$$

In fact, this form is not as weird as it seems because $1 - e^{-x}$ can be approximated well by x if the value of x is small enough, thus essentially making this transfer function approximately linear. We have done experiments on pure raycasting code to confirm the close relationship between these two transfer functions. To support arbitrary opacity transfer function without any other information available, the best we can do is to assume the scalar vales do not vary very much within a sub-block, thus we can use the average of the projection sum to approximate the scalar value at each sample point, and this leads to:

$$O_{sub-block} = 1 - [1 - O(D_{avg})]^n \tag{7}$$

where $D_{avg} = \frac{\sum_{i=1}^{n} D_i}{n}$. Similarly for color values, we can again make use of the average density:

$$C_{sub-block} = C(D_{avg}) * O(D_{avg}) * \frac{1 - [1 - O(D_{avg})]^n}{O(D_{avg})} \tag{8}$$

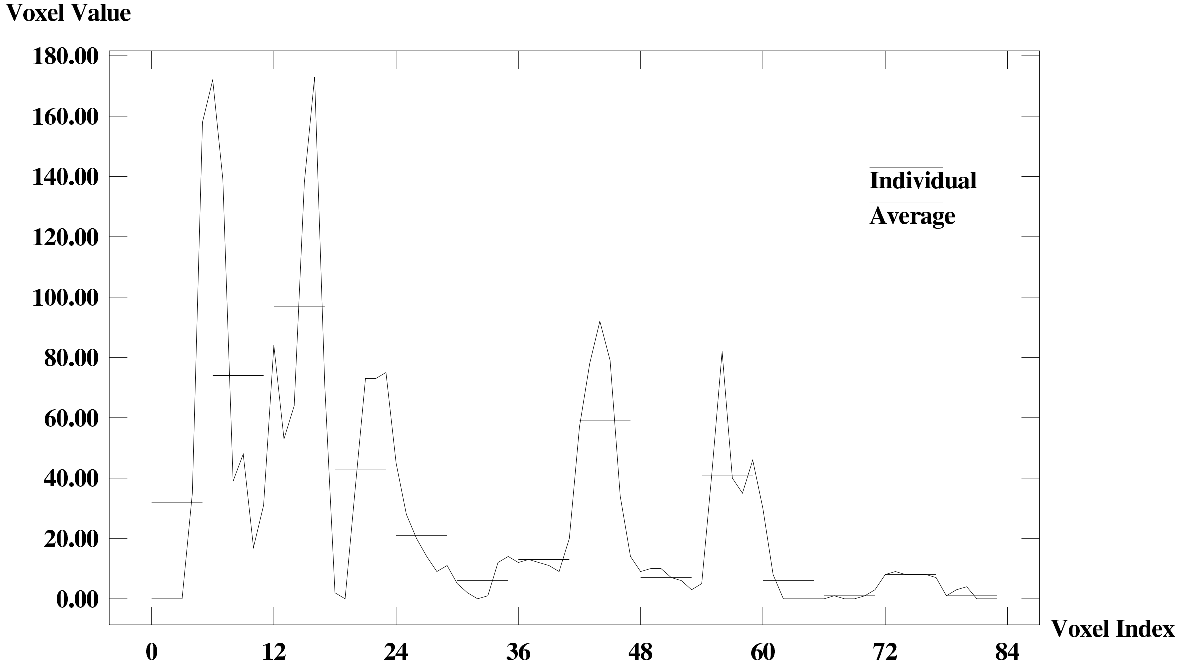

Figure 5: *The sequence of voxel values encountered by a sample projection ray, as shown as the solid line, and the average approximation voxel value used for each sub-block, as shown as the dashed line.*

Notice that a sub-block cannot be too big, otherwise using its average density may not reflect the real distribution very well, as shown in Figure 5, where the scalar value distribution corresponding to a particular ray on a CT data set is recorded. It is apparent that the fluctuation of the data values can be high, which suggests a smaller sub-block size for a better approximation. One more problem is to compute the value of n for calculating the average density. For non-orthographic views, the value of n may vary significantly. One approximation to find this is to build an artificial volume where each voxel has a unit density, and its size the same as a sub-block. By going through the same slice extraction and inverse 2D Fourier transform processes, we can obtain a special image, and the value of n can be determined by sampling on the corresponding pixel from this image.

Given the aggregate color and opacity approximations for each sub-block from its projection sums, we then apply sub-block-by-sub-block spatial domain compositing using the following equations:

$$C_{out} = (1 - O_{in})C_{sub-block} + C_{in} \quad (9)$$

$$O_{out} = (1 - O_{in})O_{sub-block} + O_{in} \quad (10)$$

where C_{in}, C_{out}, O_{in} and O_{out} are with respect to a sub-block, not a voxel. Note that Equation 9 is different from Equation 1 because the occlusion effect due to each voxel has been captured in the aggregate color calculation, and should not be repeated in sub-block-level compositing.

The third drawback comes from the gaps between sub-blocks. Currently we partition sub-blocks in a disjointed manner. Since ideally a sample point can fall within the area where no sub-blocks can cover, it potentially leads to a "blocky" effect in our generated images. For orthographic views, this is not a problem, because sample points can be aligned with the original grid points. One

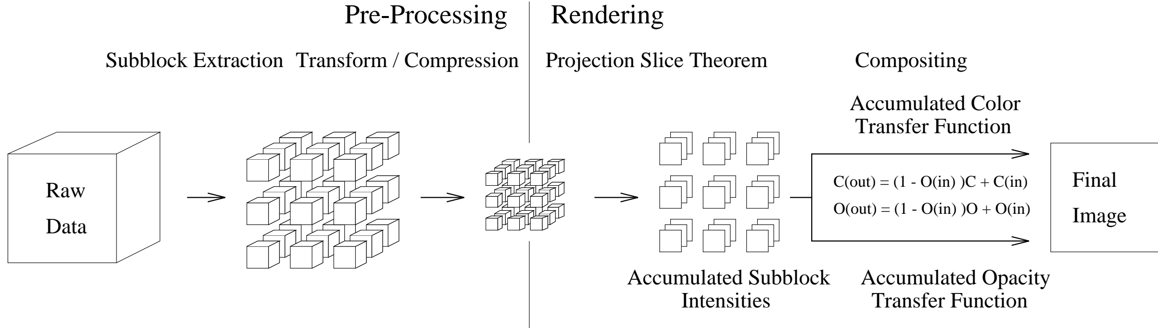

Figure 6: *Integrated volume data compression and rendering. Each sub-block is first independently Fourier-transformed. Then the projection image of each sub-block is computed through the Fourier Projection Theorem. The final rendered image results from compositing the sub-blocks' projection images according to the view angle and opacity/color transfer functions.*

may choose to use an "overlapped" partition scheme but then how to account for the sampling duplication from adjacent sub-blocks in the overlapped regions for arbitrary viewing direction may not be trivial. A possible solution is to let the overlapped areas participate the whole process and subtract their contribution, thus getting back the real contribution of each sub-block. However, this makes the whole scheme more complicated, and smaller overlapped areas tend to include more ghost effects or aliasing errors.

Figure 6 summarizes the whole rendering process through this sub-block-based algorithm.

3.4 Performance Results

We focus mainly on the quality comparison since this is the main goal here, while other factors such as running time speedup and compression efficiency can either be proved theoretically or implementation dependent (for example different quantization schemes may be involved).

Table 1 compares the quality of each sub-block, generated either from a spatial domain projection (summation), or by the FPST, under different sub-block sizes. Color values are normalized to the range between 0 and 255. Each viewing angle is represented by a triple (x, y, z), which means the three rotation angles by each axis. As can be seen in this table, orthographic views tend to generate much better quality because they involved much less frequency domain interpolations, thus introducing less error. Furthermore, bigger sub-block size bears less error. This is because artifacts tend to occur along the boundary whereas MSE is taking an average of errors over all the sub-block, thus reducing the impact of errors as the size increases.

Table 2 demonstrates the MSE between a naive voxel-by-voxel raycasting and our sub-block-by-sub-block, integrated scheme. Again, orthographic views tend to generate better quality due to less frequency domain interpolation error. However, bigger sub-block size now tends to worsen the image quality. The errors mainly come from the *average density* approximation, where bigger sub-block size in general represents poorer approximations.

Sub-block Size	Orthographic		Non-orthographic	
	$< 0 \;\; 0 \;\; 0 >$	$< 0.5 \;\; 0 \;\; 1 >$	$< 0.25 \;\; 0.25 \;\; 0.25 >$	$< 0.48 \;\; 0.1 \;\; 0.95 >$
4 x 4 x 4	0	0.153	25.543	35.980
8 x 8 x 8	0.529	1.536	15.990	23.536
16 x 16 x 16	0.433	1.026	9.980	18.336
32 x 32 x 32	0.301	1.006	4.670	10.312
64 x 64 x 64	0.149	0.217	3.252	3.513

Table 1: *The average mean square errors between the projections derived from the Fourier Projection Theorem and those from spatial domain summing, for various viewing angles. The color values are normalized to the range between 0 and 255.*

Sub-block Size	Exponential		Linear	
	$< 0.5 \;\; 0 \;\; 1 >$	$< 0.48 \;\; 0.1 \;\; 0.95 >$	$< 0.5 \;\; 0 \;\; 1 >$	$< 0.48 \;\; 0.1 \;\; 0.95 >$
2 x 2 x 2	3.115	5.339	4.935	8.377
4 x 4 x 4	7.0699	8.001	9.129	10.560
8 x 8 x 8	10.870	12.698	12.699	14.958
16 x 16 x 16	16.775	17.314	18.397	19.402
32 x 32 x 32	25.781	26.089	27.778	28.321
64 x 64 x 64	32.750	37.336	34.579	39.601

Table 2: *The mean square errors between the rendered images using voxel-by-voxel ray casting and those from sub-block-by-sub-block compositing based on aggregate color and opacity approximation values, for different opacity transfer functions and viewing angles. The color values are normalized to the range between 0 and 255.*

4 On-the-Fly Rendering of Compressed Irregular Volume Data

As being surveyed in Yang [Yan89], the "decompression before rendering" strategy is naive but the most flexible. It can accommodate almost every possible rendering method as well as every possible volume compression method. However, it fails to reduce the decompression latency, data loading time, and memory footprint size. The "rendering in compression domain" strategy, discussed in Section 3, on the other hand, is elegant, efficient, but so far still cannot provide satisfactory rendering quality. There are at least two other possible strategies for integrating compression and rendering. One strategy is called "on-the-fly decompression during rendering", and a good example of this is from Yeo et al. [YL95], where data sets are partitioned into sub-blocks and each sub-block is compressed by a 3D DCT. At run time, depending on which sub-blocks touched by the raycaster, those sub-blocks are fetched and decompressed. In this approach it is the renderer that dictates the order of decompression. The other strategy is called "on-the-fly rendering during decompression". Yang et al. [YMC00] showed an example of such an integration, which is also the core of our proposal, and therefore it will be explained in more depth. Basically it consists of two parts: a lossless tetrahedral grid compression, and an image-order irregular grid rendering.

4.1 Tetrahedral Mesh Compression

To facilitate rendering, first we discover the boundary triangles (faces) of a data set by using its adjacency information. These boundary faces are then compressed by applying the method proposed in Mitra et al. [MC98a], a method that is similar to [GS98]. From these boundary faces, we enumerate the tetrahedra inwardly by forming a new tetrahedron at a time with each of the boundary faces. Some new boundary faces are then formed, and new tetrahedra are again attached, until all the tetrahedra are visited. To enumerate each such new tetrahedron in this process, one can choose to represent it by explicitly using its *fourth* vertex. However, this requires an index into the vertex table for each tetrahedron, and therefore is too space consuming. Yang et al. [YMC00] proposed a scheme which represents this fourth vertex implicitly. Assume *current face* denotes the face from which we grow the next tetrahedron. Empirical result shows that most of the time the pairing fourth vertex is from the faces adjacent to the current face. Each of these faces is adjacent to the current face by an edge, and the non-adjoining vertex is usually the candidate for the fourth vertex, as shown in Figure 7.

If for a given tetrahedron, the fourth vertex cannot be found from the previous edge-adjacent faces, then there are two possible cases. First, this vertex has never appeared before, so it is explicitly represented by its vertex data (coordinates, density, etc.). Second, the vertex appeared before. In this case, we first search from vertices that are incident to the vertices of the current face, but not on any face that is edge-adjacent to the current face (see Figure 8). If so, we can represent it using implicit indices; otherwise, an explicit index pointing to the corresponding vertex table entry is used. In the case of implicit indices, we need to name/order the adjacent faces or vertices in a consistent way so that the compressor/decompressor can uniquely identify the enumeration order. Usually vertices are ordered by their order of appearance, and then faces are ordered accordingly by their vertices. Huffman coding can be applied on top of this to further compress the representation.

Decompression is also straightforward here. Depending on the operation encountered, there are four cases to consider: *explicit*, *index*, *face* and *vertex*. The first case means the fourth vertex just appears for the first time, therefore the vertex table is appended with the following vertex's coordinate information. The second case means the fourth vertex is represented as an index into

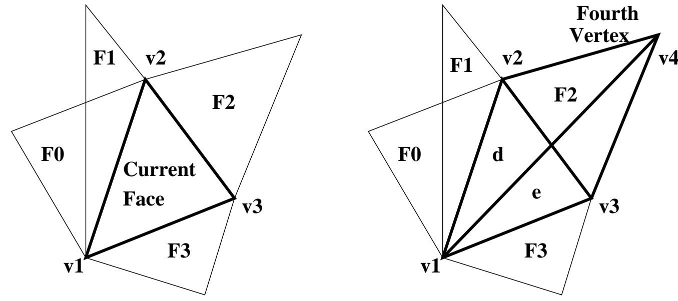

Figure 7: *The fourth vertex of a tetrahedron belongs to a face that is adjacent to the current face. In the left figure, F0, F1, F2 and F3 are the edge adjacent faces of the current face. In the right figure, after forming the new tetrahedron with the fourth vertex, two new boundary faces* $d(v_1,v_2,v_4)$ *and* $e(v_1,v_3,v_4)$ *are enumerated.*

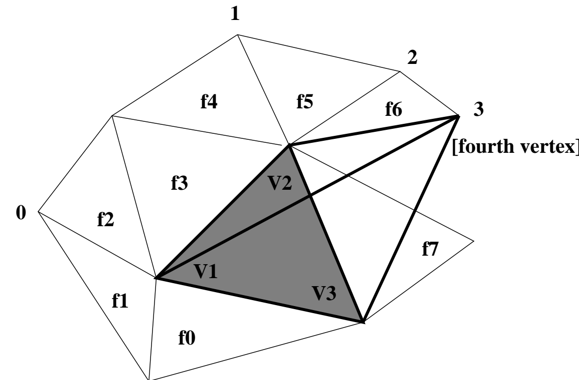

Figure 8: *The fourth vertex belongs to a face adjacent to a vertex (not edge) of the current face. The shaded triangle is the current face and* $f0 \cdots f7$ *are the adjacent faces in the current triangle mesh. Only vertex* 0, 1 ,2 *and* 3 *belong to vertex adjacent faces and vertex* 3 *is the fourth vertex in this case.*

the vertex table, and so the following index value can be directly used to construct the adjacency information of the current tetrahedron. In the third and fourth cases, where the fourth vertex is represented implicitly by some edge-adjacent or vertex-adjacent faces, the following value is used to uniquely identified the corresponding face, thus the pairing fourth vertex, and in turn the index to the vertex table can be discovered. Therefore the necessary adjacency information for the given tetrahedron can be fully reconstructed.

Notice the above compression scheme does not require the compression process to go inward or outward. One can also start from a "core" tetrahedron and grow tetrahedra outwardly. The only difference for growing outward is when a boundary tetrahedron is encountered, and it should be marked differently so that it does not participate in the subsequent tetrahedron growing process. To take care of boundary faces, one more case is added to the encoding scheme to indicate whether the current face is a boundary face or not, but then there is no need to pay the extra overhead of representing the boundary mesh explicitly, as is needed for the "growing inward" case.

4.2 On-the-Fly Rendering of Compressed Grids

4.2.1 Modifications to the Baseline Algorithm

As explained in the related work section, the baseline algorithm is from Bunyk et al. [BKS97] but with some modifications. First, instead of sampling on the faces of each intersecting tetrahedron, we do unit-distance samplings along each ray. This allows the use of arbitrary transfer functions and also increases the accuracy, since the original composition of color and opacity is just an approximation. The enabling technique for unit-distance samplings is being able to do interpolation within a tetrahedron. As is shown in the appendix, we only need totally four divisions, plus other cheaper operations, such as additions and multiplications, for all the interpolations with a tetrahedron. Furthermore, these divisions will only be performed when a tetrahedron really contribute to some pixels and some sample points. All other optimizations from the original paper are still preserved, such as how to quickly test if a point falls within a triangle.

Second, to further reduce memory footprint, we enforce a garbage collection mechanism. When a tetrahedron is done rendering, and if it is guaranteed not to be used again, then its associated resource should be released. To quickly identify if a tetrahedron should be released or not, we perform a *readiness check*. A tetrahedron passing such a test can be rendered and then its associated memory can be immediately released. The necessity of such a readiness check can be best explained by a 2D example in Figure 9. In this figure, assume rays are shooting from the image plane in a downwards direction, and ignore those upwards rays for now. We further assume that the tetrahedron decompression order is A, B, C, D and E. Now at the time when tetrahedron A just got decompressed and sent to the renderer, the renderer can render/advance the rays arriving at face a. Because the rays that can arrive at face e have not yet arrived, tetrahedron A can not be thrown away immediately. It needs to wait until the rendering of tetrahedron C is done to resume the rendering of rays arriving on face e and then releases its associated memory. An efficient way to do such a check is to first perform a *classification* of each tetrahedron according to its projection shape, once the viewing direction is known. See Figure 10 for all six possible classes of shapes. It can be shown that using at most four 2D cross products and mapping these results to a radix-3 table look-up, the classifications can be done very fast. Once the classification of a tetrahedron is known, the readiness of a tetrahedron can be easily tested and is done based on the following principles. If the projection of all the *processed* faces of this tetrahedron can cover the projection

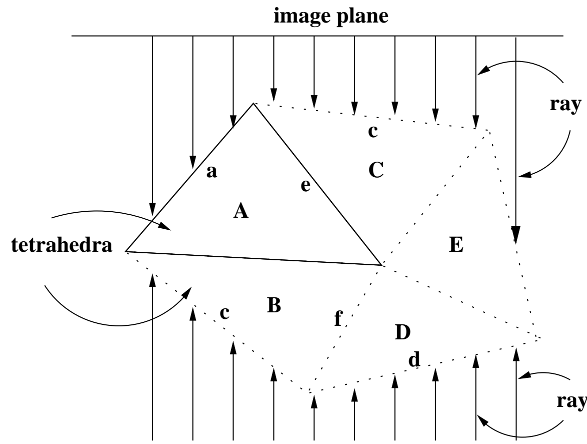

Figure 9: *This figure shows a 2D case where a tetrahedra traversal order may lead to the late arrival of some rays.*

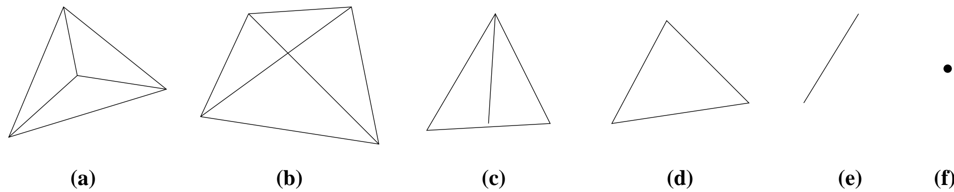

Figure 10: *The possible geometrical shapes of a tetrahedron's projection.*

of all its faces, this tetrahedron is *ready*. A face becomes *processed* once all the rays (maybe there is none) that can arrive on it have already arrived (attached). For example, in Figure 10, there are two ways for this tetrahedron to become *ready*: either the face corresponding to the biggest triangle is processed or the other three smaller faces are processed. Note that if a tetrahedron has three processed faces, then it must be ready. There are two situations where a face can become *processed*. First, all the boundary faces will be attached rays at the beginning before any tetrahedra get decompressed, therefore all the boundary faces will become *processed* by definition. Second, after each tetrahedron being rendered, all of its faces become *processed*.

Third, to further reduce the memory footprint, we may need to perform rendering from both direction. See Figure 9 again, now if tetrahedra D and E are to be decompressed first, they will not be rendered until tetrahedra A, C and E are done. If we allow the renderer to work from both directions, as also shown in this figure, then tetrahedra D and E can be rendered and their memory can be released earlier. The modification is shown in Figure 11.

4.2.2 Integration of Compression and Rendering

Given a tetrahedral mesh, the system first does a preprocessing to find out the boundary tetrahedra and thus the boundary faces by using its adjacency information. Boundary faces are compressed separately using a surface compression algorithm. Starting from the boundary faces, tetrahedra are enumerated or compressed inward. At run time, once the viewing direction is given, first the boundary faces are decompressed and all the boundary faces are back projected to find out all the rays that can intersect with them. For the rays that intersect with multiple faces, the intersection

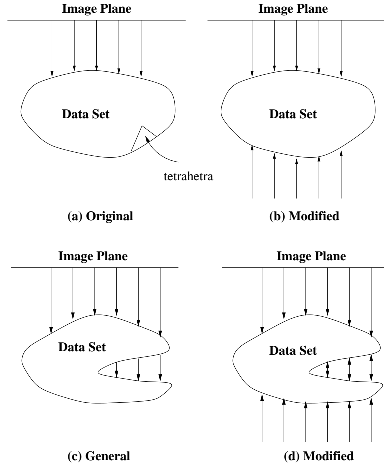

Figure 11: *A 2D example of the original rays versus the modified rays. In (a), all the rays are shooting in one direction, while in (b), for the same data set, rays are duplicated in the opposite direction; (c) represents a more general case where a data set can have multiple segments for some rays, while (d) shows our modified version of rays for this data set.*

points are sorted and chained according to their distance to the image plane. Whenever a ray intersects with a data set, it creates a *segment*, which is a continuous portion within the data set. Each segment is further decomposed into two sub-segments. As shown in Figure 12. We use two separate water-marks per segment to represent the the current progress for each ray direction. Once the two water-marks meet, it means the rendering for this segment is done. Once all the segments for a ray are done, this ray is done. Once all the rays that can intersect with a data set are done, the rendering is complete. Notice for color and opacity compositing between segments and sub-segments, different front-to-back formulas should be used:

$$C_{total} = C_{front} + (1 - O_{front})C_{back} \quad (11)$$

$$O_{total} = O_{front} + (1 - O_{front})O_{back} \quad (12)$$

Here C_{front} and C_{back} are the color values of the front and back (sub-)segments and O_{front} and O_{back} are the opacity values of the front and back (sub-)segments. The aggregated color and opacity values are denoted by C_{total} and O_{total}. They are different from the original front-to-back sample-by-sample compositing formulas as shown in equations 1 and 2, but similar to equations 9 and 10, because the underlying reasoning or deriving process is the same.

Before processing all the tetrahedra, all the rays that can hit some boundary faces are "attached" to their corresponding faces. This attachment simply means rays are put into the queues of their associated faces. Since we perform unit-distance samplings, each attachment is also associated with a "next expected sample point", thus defining the corresponding water-mark as well. Notice for each segment, the high water-mark lowers itself in the upper sub-segment while the low water mark raises itself in the lower sub-segment, until these two meet with each other. As each tetrahedron is decompressed, we check for its "readiness" from its four composing faces. If it is not ready, it is put into all the queues of its faces which are not yet *processed* so that it can wait for its readiness in the future; if it is ready, then we append this tetrahedron to a tetrahedron ready queue and start the rendering precessing from this ready queue. The reason we do not process a

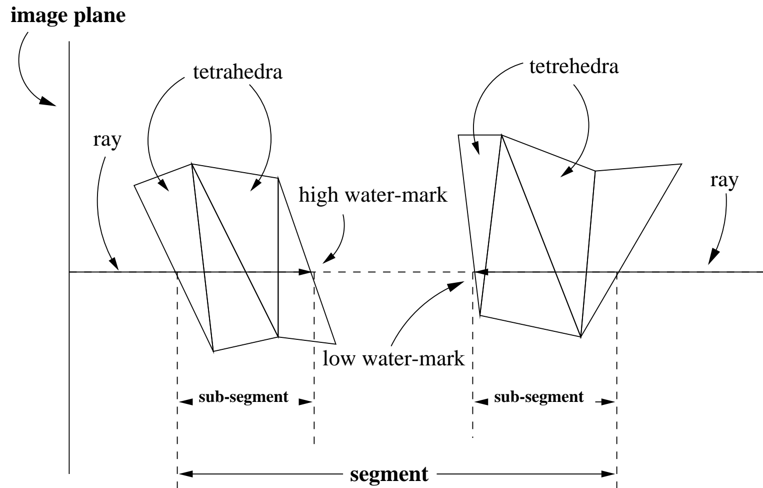

Figure 12: *A 2D example of the segment and subsegments for a given ray.*

ready tetrahedron right away but put it in the ready queue is because once a tetrahedron is done processing, all its four faces become *processed* thus it can potentially make more tetrahedra waiting on these faces become ready, or equivalently, it can trigger a "chain-reaction". Therefore each time whenever a tetrahedron is decompressed, it is tested for their readiness, and possibly put to the ready queue. The program control is thus a while loop handling the tetrahedra in the ready queue, and the control returns to the decompressor only when the ready queue is empty. Each time after a tetrahedron is rendered, all the tetrahedra in the waiting queues of its faces, which just became *processed*, will update the status of their composing faces, and tested for readiness again. Proper actions will again be taken, by enqueuing these tetrahedra either to some faces' waiting queues, or to the tetrahedron ready queue.

4.3 Performance Evaluation

4.3.1 Testbed and Data Sets

We have implemented the whole system on a Pentium II 300 MHz machine with 320 MB memory. The testing data sets are the following six data sets. The last four were converted from curvilinear grids into tetrahedral grids, while the first two are already in the form of tetrahedral grids. Their characteristics are listed in Table 3. The data set sizes are basically proportional to the number of tetrahedra.

4.3.2 Performance and Memory Requirement

As mentioned before, our system can reduce the memory footprint significantly, especially for large data sets. This reduction can further improve the system performance noticeably, especially for out-of-core rendering. In some cases the generic "decompression before rendering" approach simply cannot render at all due to its much bigger memory footprint requirement. To understand how the memory usage drops in our system, Figure 13 compares the peak memory requirement for the generic approach versus Gatun. As can be seen from this figure, as data set sizes increase, the peak memory required by Gatun is usually less than half of the size of the generic approach. Note that

Data set	Points	Tetrahedra	Faces	Boundary Faces
Spx	2896	12936	27252	2760
Fighter	13832	70125	143881	7262
Blunt	40960	187395	381548	13516
Combustion	47025	215040	437888	15616
Post	109744	513375	1040588	27676
Delta	211680	1005765	2032084	41468

Table 3: *Description of our testing data sets.*

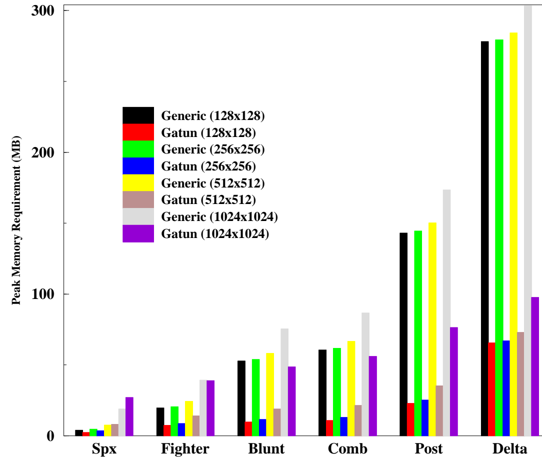

Figure 13: *Peak memory requirement for different data sets.*

because no garbage collection scheme is implemented, the generic approach will consume memory at its peak size throughout, but with the help of garbage collection, Gatun in general will use much less memory than its peak size.

To understand how Gatun performs better than the generic approach, Figure 14 compares the execution time, which includes the decompression time, the I/O time, and the the rendering time altogether, for all six data sets, four image resolutions, and three memory configurations: 320MB, 160MB, and 80MB. Especially for the last two memory capacities, some of the rendering will go out of core, therefore their rendering times lengthen noticeably.

To highlight how Gatun out-performs the generic rendering system when memory is stressed, Table 4 lists the total execution time for four data sets at some resolution when only 160 MB or 80 MB memory capacity is available. As can be seen, paging starts to emerge (marked as "*") and slows down the rendering dramatically. As a result, Gatun can perform at least one or two orders of magnitudes faster than the generic system.

4.4 Further Optimizations

There are two problems in Gatun that should be addressed. The first problem is related to the overheads associated with classification and readiness check. Classification needs to be done at

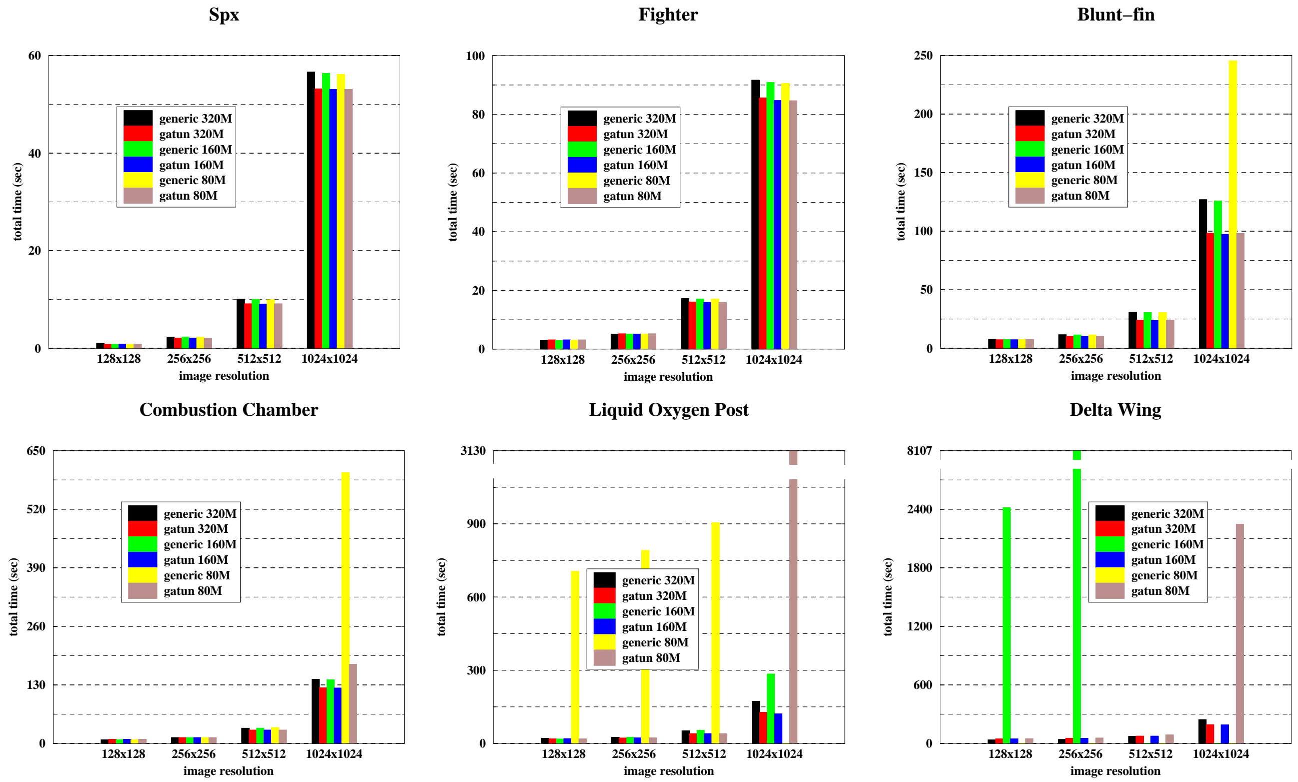

Figure 14: *Execution time for all memory configurations.*

Data Set	Image Resolution	160 MB		80 MB	
		Generic	Gatun	Generic	Gatun
Blunt-fin	1024×1024	125.88	97.36	245.54*	97.96
Combustion Chamber	1024×1024	141.28	123.12	601.17*	176.15*
Liquid Oxygen Post	128×128	18.51	19.17	705.67*	19.26
	256×256	24.80	23.40	791.35*	23.45
	512×512	54.06	40.69	964.70*	40.73
	1024×1024	284.95*	122.16	N/A	3129.55*
Delta Wing	128×128	2414.41*	47.77	N/A	50.26

Table 4: *Execution time for four data sets at some resolutions for 160 MB and 80 MB memory capacity. "*" means* paging *happened.*

most once for each tetrahedron, while multiple readiness checks may be required for each tetrahedron. In practice, a significant percentage of tetrahedra do not contribute to any pixel at all thus all the associated classification and readiness checks are wasted. Worse yet, this cost is *resolution independent*, and therefore its relative cost increases for images with smaller resolution. Fortunately this can be easily fixed by performing a "pre-filtering" step, which projects the four vertices of a tetrahedron onto the image plane and finds their bounding box. If the bounding box is contained within or matches with a cell formed by four pixels on the image plane, then obviously this tetrahedron will not contribute to any pixels on the image. Otherwise, a normal classification is performed. Experiments show that for all the six data sets, only less than 10% of the total number of tetrahedra will pass the pre-filtering test and still contribute nothing. Notice once a tetrahedron is pre-filtered, no more readiness check or rendering is necessary but its four faces nevertheless change their status to *processed* so that other waiting tetrahedra can be potentially made ready.

The second problem is about the support of early ray termination. Recall in the rendering algorithm the reason we went for a bi-directional rendering is to deal with each decompressed tetrahedron as soon as possible. However, due to the occlusion effect, the rendering performed from the opposite direction may end up being wasted; on the other hand, using one-directional rendering may consume a bigger memory, because of the hanging-around of decompressed tetrahedra; however, it may give a better performance if most of the rays terminate very early. We will compare this tradeoff more thoroughly in the future.

5 On-the-Fly Simplification and Rendering of Compressed Irregular Volume Data

Taking on step further beyond the algorithm discussed in Section 4, we have also found that volume simplification can be easily integrated into the previous framework [YC01a]. Since volume simplification can be viewed as a form of lossy compression, the work described in this section provides a powerful mechanism to dynamically create versions of a tetrahedral mesh at multiple resolution levels directly from its losslessly compressed representation. Furthermore, this integration offers the potential to dynamically adjust the amount of rendering computation to match a given hardware/software platform while maintaining adequate interactivity.

5.1 On-the-Fly Tetrahedral Mesh Simplification

The on-the-fly tetrahedral mesh simplification algorithm is composed of two parts: the static simplification algorithm and the dynamic (run-time) simplification algorithm. The static simplification algorithm computes a *view-independent* simplification hierarchy, while the dynamic simplification algorithm simplifies the mesh at run time based on the given framework simplification ratio and the static hierarchy. In principle, the integration can be applied to any volume simplification algorithms that use the *vertex merge* primitive, or different primitives (such as *edge collapse, triangle collapse, or tetrahedron collapse*) that can be converted into this *vertex merge* primitive. The fact that *edge collapse* is used excessively in today's (surface and volume) simplification algorithms makes this virtually a non-constraint. Furthermore, because this primitive is more general than *edge collapse*, it can also be applied to simplification algorithms that are based on the *vertex clustering* primitive. Moreover, because many view-dependent simplification just builds the simplification hierarchy in a different way than the view-independent case but is still based on the *edge-collapse* as the primitive,

our integration framework can also accommodate view-dependent simplification. In this implementation, we chose a generic volume simplification algorithm proposed by Gelder et al. [GVW99] to build the static simplification hierarchy, and its implementation is briefly discussed next.

5.1.1 Static Simplification Algorithm

Gelder et al.'s algorithm [GVW99] consists of two parts: the internal vertex algorithm and the external (boundary) vertex algorithm. Basically for each vertex, whether it's internal or external, the goal is to find its best *destination vertex* that it can be merged into, and the criterion is to minimize the amount of change due to this *edge collapse*. Most simplification algorithms, for surface or volume, share the same flavor, with just the difference on how the change is quantified, or equivalently, how the error metric is defined. For this paper, there are two kinds of metrics: density-based and mass-based. The density of a tetrahedron is defined as the average scalar values on its four vertices, while the mass of a tetrahedron is its density value times its volume size, which is already well defined geometrically. Now for each vertex, each of its edge collapse towards one of its neighbors will affect the mass or density of some tetrahedra adjacent to this vertex, the sum of these changes defines the error (or cost) for this edge collapse operation. Notice these changes are "signed" because some tetrahedra may be affected positively or negatively, in terms of mass or density. Furthermore, some edge may not be *collapsible* because doing so will cause some of the tetrahedra to become "negative volumed". The cost for such edge collapse operations is marked as infinity. The error associated with each vertex is determined by choosing an edge with one of its neighbors that minimizes this cost, thus also deciding its destination vertex. The paper suggests that the mass-based error metric is superior to the density-based error metric. These two metrics are called *data-based* metric because it has something to do with the data values on each vertex. Another metric, which is called *geometry-based* metric and applies only to external vertices, is only based on the geometric property. First, to avoid apparent shape or silhouette changes, a boundary vertex can only be merged into another boundary vertex. Second, each edge collapse for a boundary vertex will affect not only its adjacent tetrahedra, but also its adjacent boundary faces, changing from a group of old faces to a group of new faces. The geometry-based error is defined by summing all the *geometric difference* between all possible (old, new) pairs of faces, with the constraint that each such pair must share at least one old edge. The *geometric difference* between two faces is defined by the norm of the difference vector from the two normals on these faces, multiplied by the area of the old face, and divided by the total area of all the old faces. The final error metric associated with an edge collapse of this boundary vertex is a weighted average of its data-based error and geometry-based error. In addition to the original constraint on an edge collapse for an internal vertex, each edge collapse of a boundary vertex also cannot cause any of the affected faces to have "negative area", otherwise the cost of this edge collapse is marked as infinity. Now each boundary vertex will pick a neighboring boundary vertex as its destination vertex such that the cost for this edge collapse is minimal.

Once the error metric of each vertex is calculated, a simplification hierarchy can be built using a priority queue. All these vertices are first inserted into this priority queue according to their associated error metric value. Each time the vertex with the smallest error value is chosen, and its corresponding edge collapse is performed, all its affected neighboring vertices are deleted from the priority queue and their associated errors are re-calculated. These vertices are then re-inserted into the priority queue. This process proceeds until a user-specified error threshold is met, the desired number (or percentage) of remaining vertices is satisfied, or none of the vertices can be

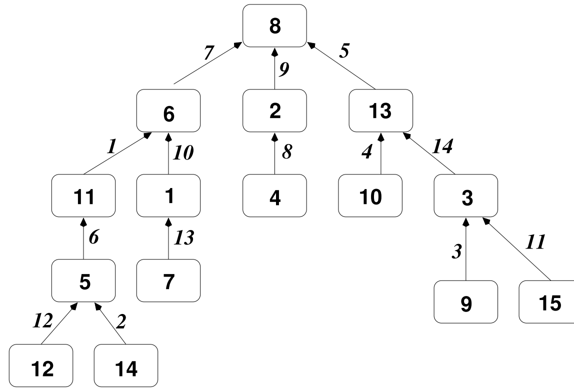

Figure 15: *An example merge-tree that shows all the vertex merge operations, as represented as edges, that are being considered, and their relative priorities, as indicated by the weights on the edges.*

further simplified due to the pre-imposed constraints, i.e., no tetrahedron has negative volume, no boundary face has negative area, and a boundary vertex can only be merged into another boundary vertex.

To make use of this static hierarchy later at run time, we chose to build the simplification hierarchy all the way to the end. That is, we will simplify a given tetrahedral mesh as much as possible, without giving any pre-specified error threshold or desired number of remaining vertices. Therefore, the only stopping criterion is the pre-imposed constraints. In addition, we record the order in which each vertex is selected, because this order can be used to guide how the run-time simplification algorithm should be performed. To speed up the run-time simplification process, a data structure called the *merge-tree* is built, and an example of merge-tree is shown in Figure 15. Here each square node represents a vertex while on each edge it records its edge collapse order, or a *rank*, and merging direction is implicit. A vertex that is never merged with any others is called an *independent vertex* and will correspond to the root in a merge-tree representation. In general a forest of such merge-trees may be possible if more than one independent vertices exist.

5.1.2 Run-Time Simplification Algorithm

At run time, given a user-specified parameter, the goal is to quickly identify the same simplified mesh from an original mesh as if the static simplification algorithm is carried out on the original mesh. For simplicity, assume for now that users only specify the desired number of remaining vertices, and let's call this number N. Later we will show how other input parameters can also be easily converted into this special input pattern. Since each edge collapse or vertex merge operation will remove exactly one vertex, the maximum number of such operations is thus bounded.

From Figure 15, a vertex that others are merged into, can again be merged into another vertex, thus creating a "transitive" relation. At run time, however, only part of all the merging operations will be performed; therefore the central problem is to find out the final "aggregated" vertex, called *ancestor vertex*, that each vertex should be merged into, according to the input N. A one-pass top-down traversal through all the merge-trees will do the job more specifically. Each node initially set its ancestor to itself and as the traversal starts, we will examine each node to see if it has a parent. If so, we check if the associated upper link has a rank less than N, and set the ancestor

to be its parent's ancestor if that is the case. Each node thus is touched just once, but after the traversal is done, each node will know its ancestor, according to the input N.

Input in the form of "percentage of vertices simplified" or "percentage of vertices left" can be easily converted to the "number of remaining vertices" form. However, there is another typical input form that needs special processing, that is, the error threshold. This means a user wants to simplify the mesh so that the no edge collapse operation can introduce an error that is larger than the error threshold. To support this metric, there are at least two methods. The first choice uses an approach similar to the previous ranking method, and assumes each edge collapse has recorded the associated error. Again a simple traversal can find the ancestor for each node, by replacing the original checking with the checking to see if an upper link, if exists, has an error less than the threshold, and set the ancestor field to its parent's ancestor field if so. This approach, although seems reasonable, may not reflect the results when the static simplification algorithm is applied step by step. This is because during the construction of the static hierarchy, sometimes the current minimal error value extracted from the priority queue may be smaller than some previously extracted error values. A different way, although requires some nominal processing, can faithfully reflect what will happen if a static scheme is used. This requires a linear array whose size is as big as the number of vertices, and the $n-th$ entry records the largest error, initially zero, that it has seen so far after performing the $n-th$ vertex merge operations. The values from this array will be sorted in a non-decreasing order. Given a desired error threshold, a binary search can identify exactly after how many vertex merge operations, the resulting error will exceed the error threshold for the first time. This is exactly the point the static algorithm should have stopped. We record this number of operations as N, and fall back to the algorithm described in the beginning of this section. Notice except for the last input type, where an additional binary search is involved, only one pass of walking through all the nodes is required. Because the merge-trees structure need not be modified it can be used repeatedly.

5.1.3 Integration with Decompression and Rendering

Once the run-time simplification step has determined the ancestor for every node, simplification can be readily integrated into the decompression/rendering pipeline. The basic idea behind this integration is as follows. Given the input input N, if a vertex is different from its ancestor, then it must be merged to some other vertex. If a tetrahedron has fewer than four distinct ancestors, this tetrahedron will be eventually simplified away, and therefore rendering it is a waste of effort; otherwise, this tetrahedron is *valid*. However, we may not be able to render it because some of the ancestor vertices are not available, or decompressed yet. Therefore, each such tetrahedron will set its *waiting count* field to be the number of ancestor vertices that it is still waiting for, and put itself into the tetrahedron waiting queues of all the ancestor vertices that it waits for. Whenever a new vertex is decompressed, it first checks if it has any tetrahedra waiting for it, and if so, dequeues these tetrahedra and decreases their waiting count by one. Any tetrahedron whose waiting count reaches zero is sent to the renderer, because all its vertices are present. Notice the renderer may buffer a tetrahedron again sometimes because of garbage collection, but that has nothing to do with simplification.

In Gatun, decompression starts from the boundary surfaces, and the boundary surface compression applies a different algorithm. Since boundary surface is also susceptible to simplification, and it is also used for setting up segments for each ray before any tetrahedra being decompressed, the boundary face simplification could become a problem. Fortunately one can test if a boundary

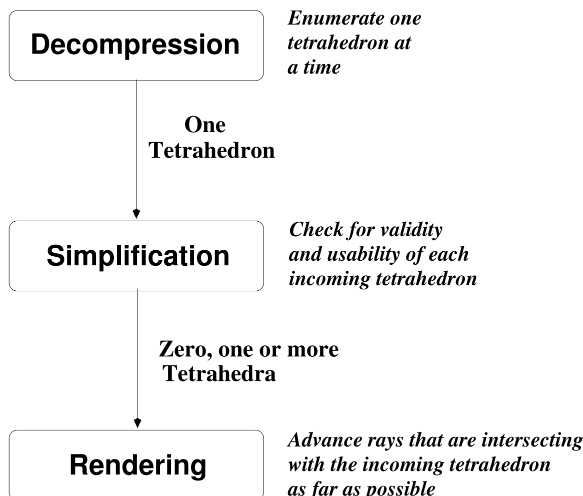

Figure 16: *The interface between and the operations performed by the stages in the integrated pipeline.*

face contains three distinct ancestor vertices or not, and if so we put it in its ancestor vertices' face waiting queue and initialize its waiting count accordingly. Those boundary faces have fewer than three ancestor vertices are discarded. Each "non-simplified" boundary face, once all its waiting ancestor vertices arrive, can then go through the normal process in Gatun, that is, back-projection, finding intersecting rays, and so on. Notice because a boundary vertex can only be merged into another boundary vertex, one boundary surface can only be simplified into another boundary surface, thus clearly separating the decompression for the boundary surface and that for the internal tetrahedra. Figure 16 depicts the relationships between the different stages in the Gatun pipeline. It is clear that the modification to Gatun to incorporate the simplification functionality is minimal: the checking for the number of distinctive ancestor vertices, the queuing of tetrahedra, and their waiting count management.

5.2 Performance Evaluation

To evaluate the performance, we have used the same six data sets. The compression and rendering algorithms we used are exactly the same as in Section 4 with only some modification to bridge the simplification in between. We have also implemented Gelder et al.'s algorithm [GVW99] as our testing algorithm. We conducted the experiment on a new test-bed, which is a PIII 700 MHz machine running RedHat Linux, with 384 MB memory.

We first measured the run-time simplification overhead, which mainly comes from the top-down traversal of the merge-tree to adjust each vertex's ancestor field, and the enqueuing and checking for pending tetrahedra or faces that are waiting for their vertices to be decompressed. Results show that the run-time simplification overhead is always less than 5% of the total end-to-end rendering time for all the data sets, thus demonstrating the feasibility of this run-time simplification scheme.

Second, we compare the performance of this system with the baseline system. To have a fair comparison, the baseline system reads a data set from its compressed form, decompresses it within the memory, simplifies the uncompressed mesh, and renders the resulting simplified mesh. In practices, without modifying the interfaces of decompressor, simplifier and renderer, some intermediate

file readings/writings is required. This tends to increase the relative performance benefits of our system. Figure 17 shows such a comparison for generating 256×256 images from all six data sets, under different simplification ratios. As can be seen from this figure, our system wins in all the cases.

To show how simplification helps to reduce the end-to-end time, we vary the simplification ratio and see how it impacts the performance. Simplification ratio is defined as the number of simplified vertices versus the number of all vertices, and therefore simplification ratio 0 means no simplification at all while simplification ratio 0.99 means 99% of the vertices are simplified. We have also made a different renderer in our system, which is called *area-based* renderer, in addition to Gatun's *volume-based* renderer. The area-based renderer assumes all the sample points are only on the faces of tetrahedra, just like the original Bunyk's algorithm [BKS97]. Volume-based renderer, as in the original Gatun, uses sample points within tetrahedra, as a normal raycasting algorithm will do on regular grids. Although a volume-based renderer provides better image quality because it uses more accurate transfer functions and compositing formulas, the rendering performance gain from simplification is expected to be small, since most volume simplification algorithms tend to minimize the overall silhouette change, and as a result limit the extent of volume change thus the reduction in volume-based rendering time.

Figure 18 shows the rendering time comparison between these two renderers under different simplification ratios. Surprisingly we found that the volume-based approach is not lagging behind the area-based approach by much. The reason is that volume-based renderers can also benefit from simplification because some intermediate results, involved in the computation of the weights for interpolation at each sample point, can be reused more often when tetrahedra become larger, which is usually the result of simplification. As for how interpolation reuses the intermediate results for computing the weights is explained in the appendix. This also explains why the performance of volume-based renderers also improves with the increase in simplification ratio.

To understand how much volume simplification degrades the final rendered image quality, we calculated the RMSE (root mean square error) between the resulting images generated from simplified data sets and the original data set under different simplification ratio. The RMSE computation excludes background pixels where there is no ray contribution. Figure 19 shows the quality degradation is minimal and noticeable deviations only arise for very aggressive simplification ratio.

5.3 Remark: Integration of Simplification and Rendering

This is in fact a special case or a "simplified" version of the integration mentioned in this section, by just removing the compression part from the whole pipeline. The relationship between the simplifier and renderer remains the same.

6 Summary and Proposed Research

6.1 Summary of Integration Framework

The main contribution of our previous work is the development of techniques to integrate compression, rendering and even simplification together into a unified pipeline. Except for the work on rendering in the compression domain, most of the techniques are general, because they can be applied to different simplification algorithms, different lossless compression algorithms, and even

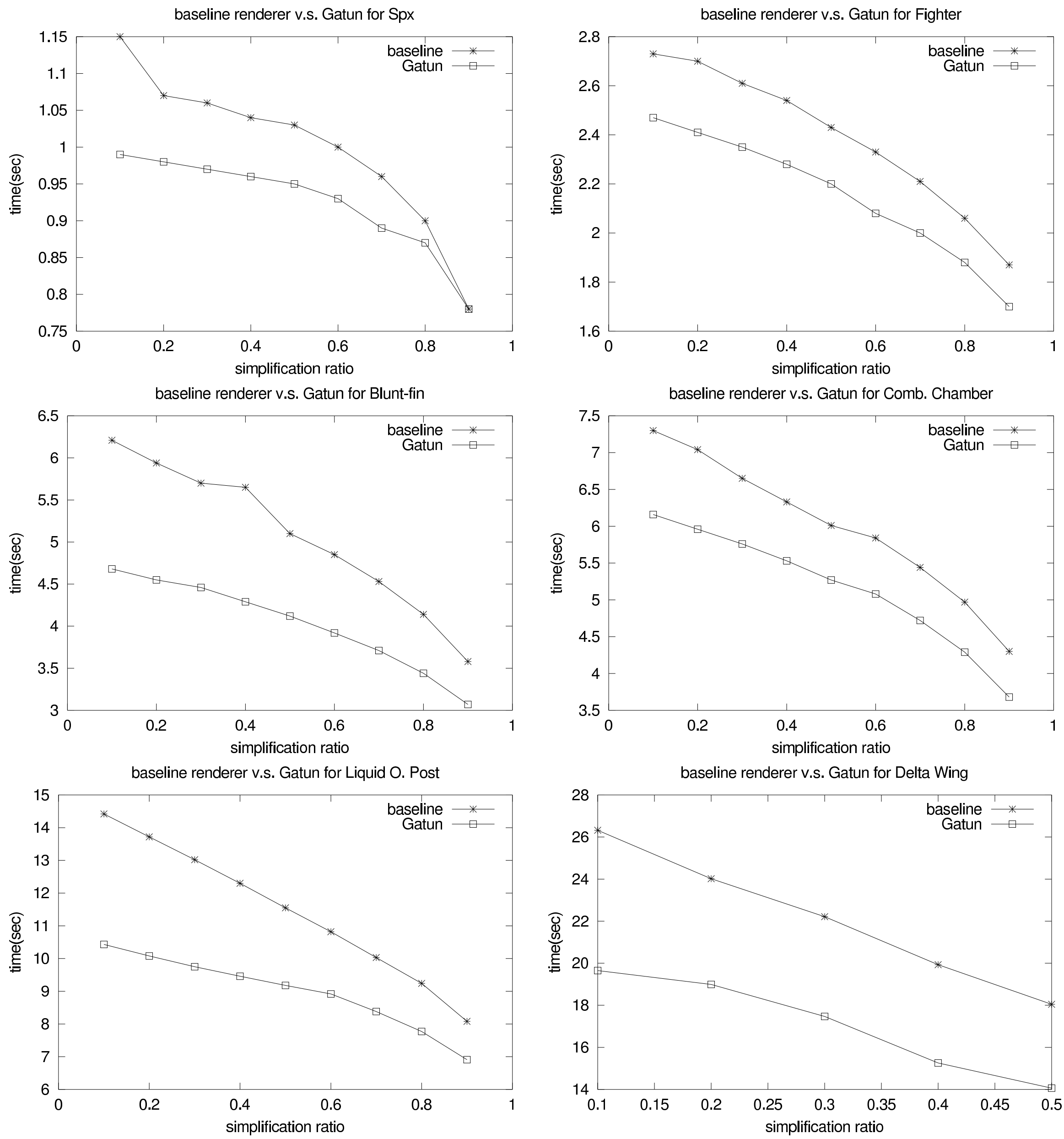

Figure 17: *Comparisons between the baseline renderer and Gatun for different simplification ratio. Image resolution is set at* 256×256.

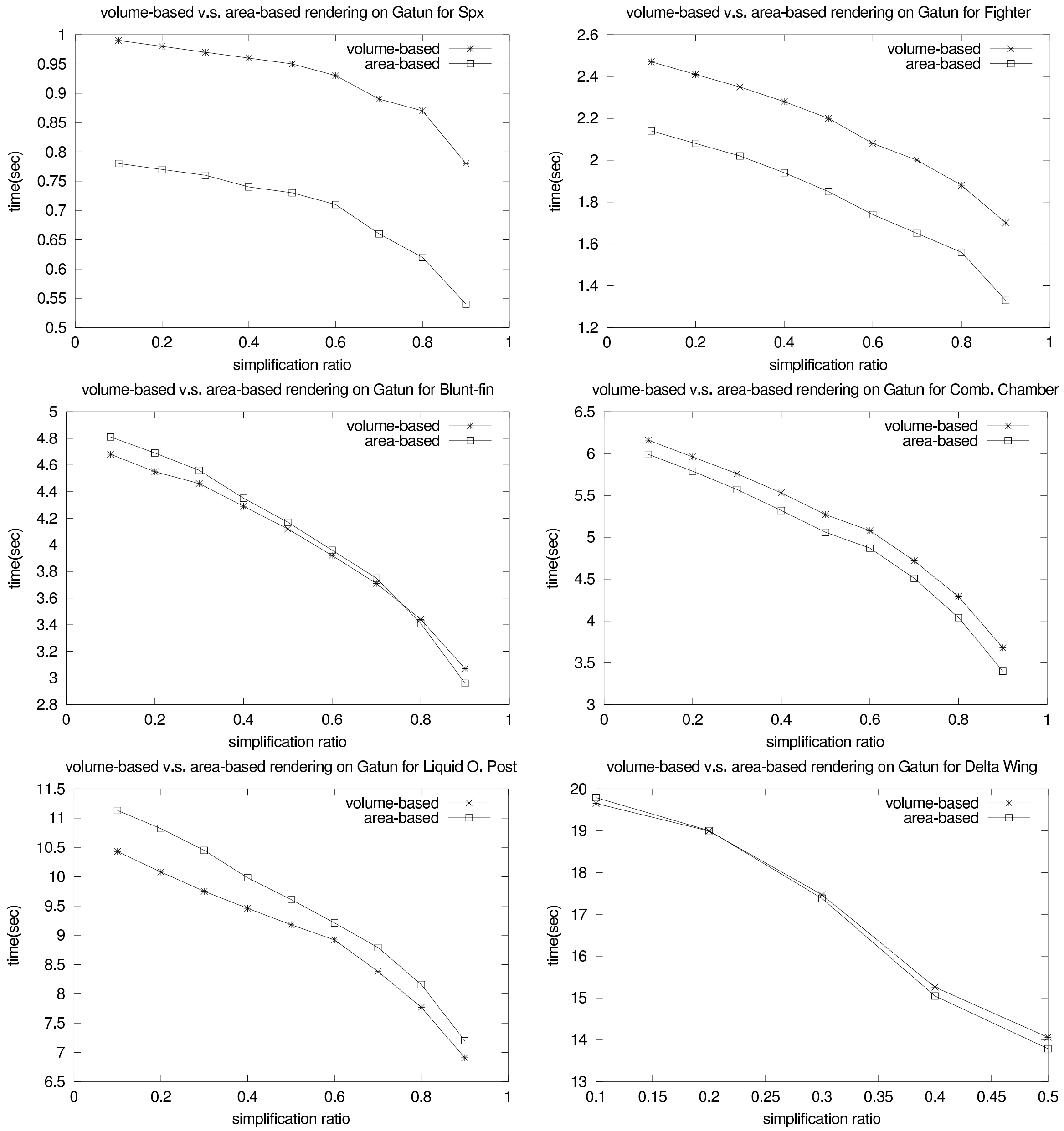

Figure 18: *Comparisons between volume-based and area-based version of Gatun for different simplification ratio. Image resolution is set at* 256×256.

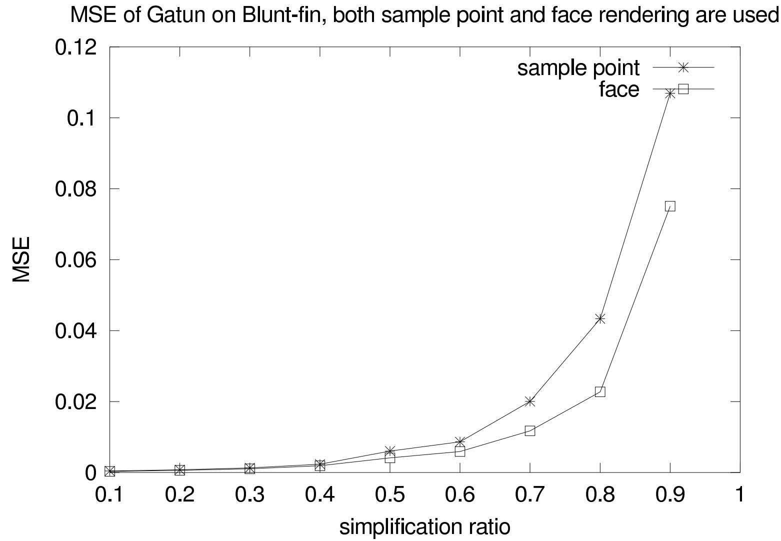

Figure 19: *The RMSE of the images generated from Blunt-fin data set for different simplification ratio. Image resolution is set at* 256×256

different rendering algorithms. Let us examine those techniques one by one. The main focus is on tetrahedral meshes as well as on triangular meshes.

As mentioned before, the only restriction to the applicable simplification algorithms is that the involved primitive can be converted to the operation of a *vertex merge*. As pointed out earlier also, *edge collapse* and *vertex clustering* are two of the most popular primitives and each of them can be treated as a vertex merge operation. Notice this not only applies to volume simplification, but also applies to surface simplification as well. The only other important primitive that is left out is the *vertex removal* operation, where some *re-triangulation* or *re-tetrahedralization* is required. However, it is less likely that *vertex removal* will be used frequently in volume simplification since it is not always the case that the resulting region can be *tetrahedralized*. Second, if vertex removal is applied only on surface meshes, or on the boundary faces of a volumetric grid, it can be shown that a vertex removal operation followed by a "particular" type of re-triangulation can be converted into vertex merge operations. Figure 20 gives one such example. This type of re-triangulation has the property that some vertex (such as the vertex A in Figure 20) must participate in all the new triangles. And it is evident that by collapsing the vertex pair formed by this vertex with the vertex to be removed, the same re-triangulation can be achieved. The same is true for the vertex removal followed by a re-tetrahedralization. To summarize, our technique can be used to combine simplification with rendering, with or without the presence of compression, and for both surface mesh and volumetric mesh, or at least for the boundary surface of a volumetric mesh. A very promising extension is to support view-dependent simplifications in this framework as well because it just involves different ways to build the simplification hierarchy, which should be orthogonal to the compression and rendering algorithms in use.

It is more interesting to examine the feasibility of using other lossless compression algorithms. Most decompression algorithms are able to decompress one tetrahedron at a time, and more importantly, a newly decompressed tetrahedron should be somehow "adjacent" to part of the mesh that has already been decompressed. The first property almost always holds for a mesh decompression

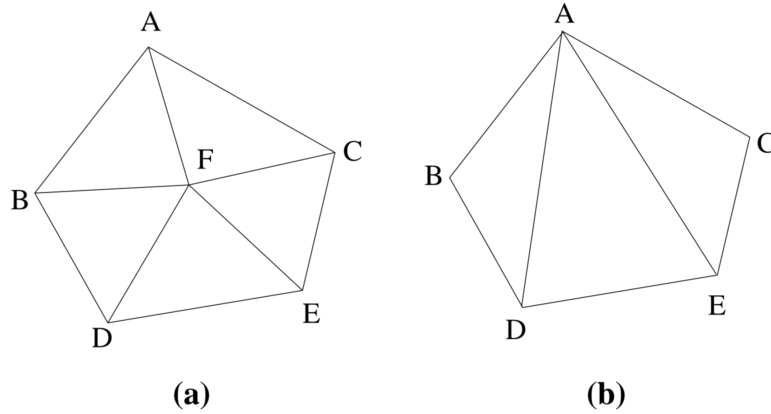

Figure 20: *(a) The original mesh. After vertex F is removed and the resultant mesh can be re-triangulated into (b). It is clear that these operations can be replaced by a single edge collapse (or vertex merge) operation of* $\overline{AF}$ *(from F to A).*

algorithm while the second property is almost necessary to achieve high compression efficiency, because it allows exploring all possible "reuse" of neighboring vertices, edges, or faces. It seems that a "layering" enumeration approach is by far the most effective scheme [GGS99, YMC00] for compressing tetrahedral grids. The only additional complication is just to turn some of the compression algorithms to work from "outward" to "inward" and compress the boundary surface separately. Although this fixed decompression order of tetrahedra can not favor arbitrary view directions, therefore some buffering from the renderer is necessary, it is probably the best that a renderer can hope for from a decompressor, in terms of dealing with the correct ordering, without sacrificing much compression efficiency. The layering structure is really not fundamental to the integration of rendering and simplification, but is crucial to saving the memory footprint size for necessary buffering. For surface mesh, the integration of compression and rendering is trivial, as the ordering problem is already taken care of by the traditional *Z-Buffer* algorithm, therefore no buffering in the renderer is required. For integration of compression and simplification, our technique can still be applied, given the fact that checking for duplicate ancestor vertices is not the bottleneck of the pipeline.

Finally, it is not necessary that we only stick to Bunyk et al.'s algorithm [BKS97]. Some of the projection-based method or even plane-sweep algorithms are also applicable. Probably the only change is that the concept of "rays" does not exist any more, and each face becomes the operation unit. When combined with the readiness check, not only an efficient garbage collection mechanism can be employed to reduce the memory footprint size, but also the correct rendering order can be enforced. Notice filtering of small tetrahedra whose footprints do not cover any pixels, can still be applied to avoid their associated overheads for readiness checks.

6.2 Future Directions

We list some possible future directions here, although we do not have time to pursue each of them. Our proposed work is described in the next sub-section.

6.2.1 Integration of Compression, Simplification, and Rendering for Surface Meshes

The work we presented in the previous section can be easily extended to deal with surface meshes. Not only there are more surface compression algorithms and simplification algorithms available but also rendering is much easier as the ordering is not important. In some cases, even decompression can be done in hardware, as shown by Mitra [Mit00]. One remaining problem is how to perform simplification fast enough so that it will not become the bottleneck. For static scenes, the simplification overheads are probably negligible, as being demonstrated in the previous section for volumetric meshes. For dynamic scenes, where the rendering direction keeps on changing, the simplification ratios may need to be adjusted to maintain reasonable rendering speed. Because decompression takes a negligible amount of time, simplification may become one bottleneck. To solve this, an incremental simplification algorithm needs to be devised, where only those parts of the surface mesh that get affected by a different simplification ratio are modified. This can be done by maintaining a data structure from which, given a rank, the vertex merging pair can be retrieved. A sorted array of the original vertex merging or edge collapsing chain and a simple binary search or hashing can easily provide the required functionality. To support smooth transition between different simplification ratios, first the affected vertex merging operations are found; second the associated change of ancestors are located; finally the resulting change can be calculated and only the affected triangles are re-sent to the graphics pipeline. Several simplification operations may be batched together to further improve the performance, but more data structures may be needed. To support dynamic scenes, at each stage the current simplified mesh as well as the ancestor information for each vertex, may need to be stored for efficient transition from one simplified mesh to another.

6.2.2 Integration of Compression, View-Dependent Simplification and Rendering for Surface Meshes

For perspective rendering, view-dependent simplification algorithms should be used to provide selective and refined control on which parts of a mesh to be simplified. As mentioned before, the only difference between view-independent simplification and view-dependent simplification lies in the way to build the simplification hierarchy. For surface mesh, assuming the viewing direction is fixed, then because rendering order is not important, all we need to do is to re-send all affected faces to the rendering pipeline to account for the transition between adjacent meshes with different simplification ratios.

6.2.3 Integration of Compression, View-Dependent Simplification and Rendering for Volumetric Meshes

Although here we discuss mainly the integration with view-dependent simplification algorithms, most of the reasoning can be well applied to the integration with view-independent simplification, our proposed work in the next sub-section. The integration of compression, simplification and rendering for volumetric meshes is more involved than for the surface meshes for the following reasons. First, the hardware support for irregular grids compression/decompression and rendering is virtually non-existent, except for some hardware-assisted rendering based on the projection or plane-sweep approach, which at best can provide only approximations. This makes the whole process much slower if higher accuracy is desired. Second, view-dependent simplifications are mostly used for browsing a data set. To support higher rendering speed, usually more aggressive

(view-dependent) simplifications should be applied. However, if the decompression time is non-negligible, it will incur extra rendering latency. Third, to support a new simplification ratio, either we keep the whole mesh around after its first decompression/simplification and perform another simplification from it, or we can perform decompression/simplification each time as if a new image is to be generated. It seems the former is favored because not only the initial decompression effort is reused, but it may also save some computation, if efficient incremental simplification algorithm is available. Fourth, to render a mesh with different simplification ratios, it seems not trivial to store the first decompressed/simplified mesh and perform incremental rendering from it and get the equivalent rendering effect. Therefore a new decompression/simplification may be needed each time. This seems to contradict with our previous choice of favoring an incremental algorithm, thus making this problem very challenging as a whole.

6.3 Proposed Work

In this sub-section, we propose two extensions to our current work. The first one is to combine all our previous techniques, together with some modifications, to perform *just-in-time rendering*. The second one is to combine compression with iso-surface extraction by decomposing a data set into layers based on the scalar values stored with its vertices.

6.3.1 Motivations and Challenges

The goal of the first extension is to trade quality for speed by dynamically adjusting the simplification ratio to satisfy the rendering time constraint. For example, a user can rotate a data set during browsing by dragging the mouse, and the velocity of manipulating the mouse could determine the desired rotation speed. Obviously the faster the mouse moves, the more aggressive simplification one should apply. However, the initial decompression time will incur a start-up rendering latency. In addition, how to evolve the image quality between meshes with different simplification ratios with minimal effort while delivering correct rendering effect is challenging.

The second work is to address the iso-surface extraction for very large data sets. Given a data set, most *out-of-core* iso-surface extraction algorithms decompose a data set into parts based on the spatial coordinates of the vertices. We propose to decompose the data set into *layers*, based on the data values of the vertices, and each layer serves as an independent unit for compression/decompression. The challenge here is how to service each iso-surface query as fast as possible while minimizing the amount of data that need to be retrieved from disk.

6.3.2 On-the-fly Simplification for Just-in-Time Rendering

To apply on-the-fly simplification to just-in-time rendering, or particularly data set browsing, decompression time must be reduced. It is obvious that decompression time is insensitive to the image resolution, therefore no matter how low the image resolution is, the decompression time remains fixed. In fact, empirical measurements showed that decompression time can account for more than 50% of the end-to-end rendering time for low resolution images. If memory capacity is sufficient, we can eliminate this initial startup latency for decompression/simplification by keeping the current tetrahedra mesh in uncompressed form for later processing. This approach is simple, but suffers from two drawbacks. First, the allocated memory for a tetrahedral mesh cannot be released even after rendering, thus making it less appealing for out-of-core rendering. Second, although one can minimally evolve between two meshes of different simplification ratios by modifying only

those tetrahedra affected, it is difficult to account for the rendering effect during the transition, if we do not want to render everything from scratch. If memory capacity is not sufficient, and/or we want to avoid the previous drawbacks, we can resort to a more straightforward approach: decompress/simplify/render each time as if a new image is generated. This solves the incremental rendering problem but now each frame suffers from the decompression delay. A compromise is to keep the first decompressed/simplified mesh around, and after modifying the affected portion due to simplification, just render the whole mesh again. Notice this approach has the problem of hogging memory, possibly redundant rendering work, and the decompression delay.

We propose a method which can strike a better balance, at the cost of decreasing the compression efficiency by half. First, we perform the decompression/simplification at the following fixed simplification ratios: 0.5, 0.75, 0.875, until the most simplified mesh can be manipulated/rendered in real time by the underlying hardware. Each such mesh will be compressed using the lossless compression algorithm and written to the disk. Given the mouse movement speed, assume there is a way to decide the maximal simplification ratio to support the corresponding rendering speed. Then the biggest simplification configuration whose simplification ratio is less than the desired simplification ratio, is located, and the corresponding compressed mesh is decompressed/simplified/rendered according to the chosen simplification ratio. The current simplified mesh will be kept around for transition into other simplified meshes whose simplification ratio is still within range. That is if the target simplification ratio that falls out of the current range, then a different base mesh configuration should be used as the starting point for decompression/simplification/rendering. This approach is similar in spirit to mip-mapped texture mapping.

To achieve correct rendering effect when evolving from one mesh to another mesh with a different simplification ratio, we first identify affected faces/tetrahedra, and re-render the affected regions accordingly. For simplicity, assume the transition is always from a mesh with a lower simplification ratio to a mesh with a higher ratio. Since transition can be viewed as a sequence of vertex merge operations, incremental rendering can be done by the following two approaches: either by a sequence of incremental rendering steps, or by aggregating the changes and performing the incremental rendering in one shot.

The first approach is similar to the one used in the surface mesh case. For each entry in the vertex merge sequence, it locates the next entry from the simplification order recorded earlier during preprocessing, to determine the next vertex merge operation to be performed. as well as the affected tetrahedra. These affected tetrahedra should be removed, but since it is difficult to "undo" the rendering effect contributed by these tetrahedra, one solution may be to find rays intersecting these tetrahedra, and "redo" the rendering of these rays. A better solution is to redo the rendering on even smaller rendering units, i.e.,*sub-segments*, within these rays. Notice this is possible because we can keep the current mesh around, together with all its old rendering state, which includes the accumulated color and opacity values for each sub-segment.

However, since each sub-segment may still contain a lot of unrelated tetrahedra, their corresponding rendering effort will be wasted. Therefore, instead of carrying out the change step by step, the second approach is to try to aggregate all the vertex merge operations together and hopefully to reduce the number of unrelated tetrahedra in those sub-segments where re-rendering is required. According to the new number of vertex merge operations to be performed, some vertices may have different ancestor vertices. After identifying the locations of these new vertex operations in the merge-tree data structure, all the changes can be determined by re-traversing this data structure in a top-down fashion. All the affected vertices, or the vertices whose ancestors are changed, can

be identified, as well as all the affected tetrahedra, and the rest of the modifying work is the same as the previous approach. To save traversal time, *sub-merge-trees* may be built, which enumerate only the vertices participating in the new vertex merge operations.

To further improve the rendering performance, one may apply early ray termination optimization by first calculating the accumulated opacity preceding an affected sub-segment. If it is already saturated, then the rendering steps after that can be skipped. However, those (sub-)segments whose rendering are skipped should be marked explicitly so that in case some preceding saturated (sub-)segments are removed due to a different simplification ratio, their rendering should be repeated if necessary.

To transit between meshes from higher-simplification-ratio configurations to lower-simplification-ratio ones is more complex than it looks. We can resort to a simpler solution by always performing the changes from a fixed simplification configuration whose simplification ratio is just smaller than the desired one, although it may require more processing. Otherwise the vertex table for each simplification configuration may need to be kept accessible all the time, so that a vertex split operation can be generated, and again the affected tetrahedra can be found for re-rendering.

During the browsing of a data set, if users stop moving the mouse, then the un-simplified mesh or a pre-specified finest simplification configuration will be rendered. If the underlying hardware is sufficiently powerful, we may afford to fork another process or thread to pre-decompress/simply/render a mesh corresponding to the mesh at the next finer level mesh in the background, thus masking the overhead of decompression.

6.3.3 Layered Representation for Out-of-Core Iso-Surface Extraction

To accommodate out-of-core iso-surface extraction, a data set should be partitioned into parts, disjointed or slightly overlapped, so that each part can be rendered separately, as done by Chiang et al. [YCS98]. In their work, a data set is partitioned into a *kd-tree* according to the vertices' coordinates, and within each part, an interval tree is used to answer dynamic queries. However, since iso-surface extraction is about finding the corresponding surfaces for a particular *iso-value*, we argue that a *value-based* partitioning scheme may result in a more efficient search.

First, we compute the range of all scalar values. Second, this range is partitioned into sub-ranges. There are many ways to partition a range into sub-ranges. A naive way is to assume uniform distribution of query iso-values, therefore each sub-range is of the same size. We may also assume a normal distribution therefore forming bigger sub-ranges around the mean and smaller ones towards boundary values. Another different idea is to make use of the histogram and ensure that each sub-range catches more or less the same number of tetrahedra. The last method seems reasonable because the number of sub-ranges, say N, can more of less guarantee that only $1/N$ of the total tetrahedra will get touched, assuming the number of tetrahedra that span multiple sub-ranges is relatively small. After deciding each sub-range, the third phase is the distribution phase. Each tetrahedron is distributed to the sub-range with which its data value range overlaps. As mentioned, a tetrahedron may intersect with multiple sub-ranges, but in general such cases should mostly happen at the boundary of each sub-range. Notice the resulting sub-mesh, corresponding to each sub-range, need not be connected, even though the original mesh is. Fourth, we build an interval tree from all sub-ranges as if we treat each sub-range as a self-contained tetrahedron. Given an iso-value, this interval tree can be used to quickly locate which sub-mesh we should go to. Finally each sub-mesh is compressed separately to further reduce storage requirement. In the compression algorithm, the traversal order may need to be modified to favor the tetrahedra with

closer scalar values, rather than the original *breadth-first* traversal strategy.

This scheme combines compression with interval tree and could proportionally restricts the search scope of dynamic search to a fixed portion of the original mesh. Furthermore, when decompression and iso-surface extraction proceed together, the target iso-surfaces can be generated in the form of triangle strips because decompression algorithm itself always enumerates triangles in the sub-mesh in a "spiral" and adjacent way.

If triangle reduction is a goal, one can also incorporate simplification into the above scheme. Some related work tackled this issue in a different way. Recall that the *adaptive marching cubes* algorithm, proposed by Shu et al., performs the marching cubes algorithm on the fly according to the desired threshold, which is equivalent to performing volume simplification implicitly. On the other hand, the *discretized marching cubes* algorithm, proposed by Montani et al. [MSS94a] modified the marching cubes algorithm slightly to make it more "user-friendly" for the ensuing simplification algorithm. Instead of performing surface mesh simplification, we may first apply a volume simplification on a target volumetric mesh, and then an iso-surface extraction algorithm to extract surfaces from this simplified volumetric mesh. Presumably a much smaller number of triangles will be generated. Therefore the technique of integrating compression and simplification in the previous section can also be applied here.

A Appendix

This appendix is to show how an interpolation operation within a tetrahedron could be done efficiently. First it can be shown that given a tetrahedron $ABCD$, where each alphabet represents the coordinates of a vertex of this tetrahedron, then for a particular x within it, its coordinate can be determined by

$$\frac{Vol(BCDx)A + Vol(ACDx)B + Vol(ABDx)C + Vol(ABCx)D}{Vol(ABCD)} \tag{13}$$

where $Vol(abcd)$ means the volume of the tetrahedron formed by the vertices a, b, c and d. However, this calculation can be further simplified without calculating any tetrahedron volume size. Observe that $Vol(BCDx)A/Vol(ABCD)$ is equal to computing the ratio of the "distance from x to the plane formed by BCD" versus the "distance from A to the same plane", because these two tetrahedra share the same bottom face but with different height. Recall the formula from a point (x_0, y_0, z_0) to a plane whose plane equation is $ax + by + cz + d = 0$, is

$$\frac{|ax_0 + by_0 + cz_0 + d|}{\sqrt{a^2 + b^2 + c^2}} \tag{14}$$

therefore unless $a^2 + b^2 + c^2$ is zero, the ratio of such can be simplified to just the ratio of the two corresponding numerators, thus eliminating the costly square root operations. Furthermore, since the "distance from A to the plane formed by BCD" part is fixed, and similarly the other three numerators, these numerators can be computed and re-used throughout all the interpolations within the same tetrahedron. This is also why by using only four divisions, all the interpolations within the same tetrahedron can be carried out. For the degenerate case where a tetrahedron becomes a triangle, a line, or even a point, we just remove the number of coefficients necessary for computation and lower the dimension of computation from volume to area or length.

For scalar value interpolation on x from the four scalar values on A, B, C and D, because usually we assume a linear variation within a tetrahedron, so the weights calculated for interpolating the coordinates can be used for interpolating the scalar values (or other associated attributes) as well.

References

[BBHV86] R. N. Bracewell, O. Buneman, H. Hao, and J. Villasenor. Fast Two-Dimensional Hartley Transform. *Proceedings of the IEEE*, 74(9):1282–1282, September 1986.

[BDM+88] M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H. Peitgen, D. Saupe, and R. F. Voss. *The Science of Fractal Images*. Springer-Verlag, 1988.

[BKS97] P. Bunyk, A. E. Kaufman, and C. T. Silva. Simple, Fast and Robust Ray Casting of Irregular Grids. Technical report, Center for Visual Computing, State University of New York at Stony Brook, 1997.

[BPS96] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast Isocontouring for Improved Interactivity. In *Proceedings on 1996 Symposium on Volume Visualization*, pages 39–46, October 1996.

[Bra84] R. N. Bracewell. The Fast Hartley Transform. *Proceedings of IEEE*, 72:1010–1018, 1984.

[CCM91] K. K. Chan, K. S. Chuang, and C. A. Morioka. Visualization and Volumetric Compression. *SPIE Proceedings*, 1444:250–255, 1991.

[CCM+00] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of Tetrahedral Meshes with Accurate Error Evaluation. In *IEEE Visualization '2000*, October 2000.

[Che01] B. Chen. Image-Based Rendering of Surfaces from Volume Data. In *IEEE Workshop on Volume Graphics, 2001*, June 2001.

[CHF96] W. O. Cochran, J. C. Hart, and P. J. Flynn. Fractal Volume Compression. *IEEE Transactions on Visualization and Computer Graphics*, 2(4), 1996.

[Chu92a] C. K. Chui. *An Introduction to Wavelets.* Academic Press, 1992.

[Chu92b] C. K. Chui. *An Introduction to Wavelets. Wavelet Analysis and its Applications - Volume 1.* Academic Press, 1992.

[CKM+99] J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum (Eurographics '99)*, 18:367–376, 1999.

[CLL+88] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two Algorithms for the Three-Dimensional Construction of Tomograms. *Medical Physics*, 15:320–327, 1988.

[CMM+97] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, April 1997.

[CP98] S. L. Chan and E. O. Purisima. A New Tetrahedral Tessellation Scheme for Isosurface Generation. *Computers & graphics*, 22(1):83–90, 1998.

[CS97] Y. Chiang and C.T. Silva. I/O Optimal Isosurface Extraction. In *IEEE Visualization '97*, pages 293–300, October 1997.

[CSK96] B. P. Carneiro, C. T. Silva, and A. E. Kaufman. Tetra-Cubes: An Algorithm to Generate 3d Isosurfaces Based Upon Tetrahedra. In *Proceedings of the IX SIBGRAPI International Conference*, pages 205–210, 1996.

[CVM+96] J. Cohen, A. Varshney, D. Manocha, G. Turk, and H. Webber. Simplification Envelopes. In *SIGGRAPH '96*, pages 119–128, August 1996.

[CYH+97] T. Chiueh, C. Yang, T. He, H. Pfister, and A. Kaufman. Integrated Volume Compression and Visualization. In *Proceedings of Visualization '97*, pages 329–336, October 1997.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets.* SIAM, Philadelphia, Pennsylvania, 1992.

[DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. *Computer Graphics*, 22(4):65–74, August 1988.

[DH92] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Rendering. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 91–106, October 1992.

[DNR90] S. Dunne, S. Napel, and B. Rutt. Fast Reprojection of Volume Data. In *Proceeding of the 1st International Conference on Visualization in Biomedical Computing*, pages 11–18, May 1990.

[EDD+95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution Analysis of Arbitrary meshes. In *SIGGRAPH '95*, pages 173–182, August 1995.

[Ede90] H. Edelsbrunner. An Acyclicity Theorem for Cell Complexes in D Dimensions. *Combinatorica*, 10:251–260, 1990.

[ESC00] J. El-Sana and Y. Chiang. External Memory View-Dependent Simplification. *Eurographics '2000*, 19(3), 2000.

[ESV99] J. A. El-Sana and A. Varshney. Generalized View-Dependent Simplification. *Eurographics '99*, 18(3), 1999.

[Fis95] Y. Fisher. *Fractal Image Compression. Theory and Application.* Springer-Verlag, 1995.

[FMSW00] R. Farias, J. Mitchell, C. Silva, and B. Wylie. Time-Critical Rendering of Irregular Grids. In *Proceedings of the XIII SIBGRAPI International Conference*, 2000.

[Fru94] T. Fruhauf. Raycasting of Nonregularly Structured Volume Data. *Computer Graphics Forum (Eurographics '94)*, 13(3):294–303, 1994.

[FY94] J. E. Fowler and R. Yagel. Lossless Compression of Volume Data. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 43–53, October 1994.

[Gal91] R. S. Gallagher. Span Filtering: An Optimization Scheme for Volume Visualization of Large Finite Element Models. In *IEEE Visualization '91*, pages 68–75, October 1991.

[Gar90] M. P. Garrity. Raytracing Irregular Volume Data. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:35–40, November 1990.

[GGS99] S. Gumhold, S. Guthe, and W. Straβer. Tetrahedral Mesh Compression with the Cut-Border Machine. In *Proceedings of the 10th Annual IEEE Visualization Conference*, October 1999.

[Gie92] C. Giertsen. Volume Visualization of Sparse Irregular Meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[GK96] A. V. Gelder and K. Kim. Direct Volume Rendering with Shading via 3D Textures. In *ACM/IEEE Symposium on Volume Visualization*, pages 22–30, October 1996.

[GLS99] M. H. Gross, L. Lippert, and O. G. Staadt. Compression Methods for Visualization. *Future Generation Computer Systems*, 15(1):11–29, 1999.

[Gra84] R. M. Gray. Vector Quantization. *IEEE ASSP Magazine*, 1:4–29, April 1984.

[Gro96] M. H. Gross. Integrated Volume Rendering and Data Analysis in Wavelet Space. Technical Report 250, Computer Science Department, Swiss Federal Institute of Technology, 1996.

[GS98] S. Gumhold and W. Straber. Real Time Compression of Triangle Mesh Connectivity. In *Proceedings of the 25th Annual ACM Conference on Computer Graphics (SIGGRAPH)*, September 1998.

[GVW99] A. V. Gelder, V. Verma, and J. Wilhelms. Volume Decimation of Irregular Tetrahedral Grids. In *Computer Graphics International 1999 Conference Proceedings*, 1999.

[GY95] M. H. Ghavamnia and X. D. Yang. Direct Rendering of Laplacian Pyramid Compressed Volume Data. In *Proceedings of Visualization '95*, pages 192–199, 1995.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh Optimization. In *SIGGRAPH '93*, pages 19–26, July 1993.

[HH92] Paul Hinker and Charles Hansen. Geometric Optimization. In *IEEE Visualization '93*, pages 55–64, 1992.

[HK99] L. Hong and A. Kaufman. Fast Projection-Based Ray-Casting Algorithm for Rendering Curvilinear Volumes. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):322–332, October 1999.

[Hop96] H. Hoppe. Progressive Meshes. In *SIGGRAPH '96*, August 1996.

[Hop97] H. Hoppe. View-Dependent Refinement of Progressive Meshes. In *SIGGRAPH '97*, August 1997.

[IK95] T. Itoh and K. Koyamada. Automatic Isosurface Propagation Using an Extrema Graph And Sorted Boundary Cell Lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, December 1995.

[IYK96] T. Itoh, Y. Yamaguchi, and K. Koyamada. Volume Thinning for Automatic Isosurface Propagation. In *IEEE Visualization '96*, pages 303–310, October 1996.

[Jac89] A. Jacquin. *A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding.* PhD thesis, Georgia Institute of Technology, August 1989.

[Kau91] A. R. Kaufman. *Volume Visualization.* IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1991.

[Kau94] A. R. Kaufman. Research Issues in Volume Visualization. *IEEE Computer Graphics and Applications*, 14(2):63–67, March 1994.

[Kni00] G. Knittel. The UltraVis System. In *IEEE Visualization '2000*, October 2000.

[KT96] Alan D. Kalvin and Russell H. Taylor. Surperfaces: Polygonal Mesh Simplification with Bounded Error. *IEEE Computer Graphics and Applications*, 16(3):64–77, May 1996.

[LC87] W. E. Lorensen and H. E. Cline. Marching Cube: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[Lev88] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, March 1988.

[Lev90a] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[Lev90b] M. Levoy. Volume Rendering by Adaptive Refinement. *The Visual Computer*, 6(1):2–7, February 1990.

[Lev92] M. Levoy. Volume Rendering Using the Fourier Projection-Slice Theorem. In *Proceedings of Graphics Interface '92*, pages 61–69, 1992.

[LH91] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Computer Graphics*, 25(4):285–288, 1991.

[Lin00] P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. In *SIGGRAPH '2000*, pages 259–262, 2000.

[LL94] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation. In *Computer Graphics*, pages 451–458, July 1994.

[LSJ96] Y. Livnat, H. Shen, and C. R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, March 1996.

[LW85] M. Levoy and T. Whitted. The Use of Points as a Display Primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

[Mal89] S. G. Mallat. A Theory for Multiresolution Signal Decomposition: The Wavelet Representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[Mal93] T. Malzbender. Fourier Volume Rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[Mao96] X. Mao. Splatting of Non Rectilinear Volumes Through Stochastic Resampling. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):156–170, June 1996.

[Max95] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[MC98a] T. Mitra and T. Chiueh. A Breadth-First Approach to Efficient Mesh Traversal. In *Proceedings of the 13th ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 31–38, September 1998.

[MC98b] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *Proceedings of Visualization'98*, pages 239–245, October 1998.

[MHC90] N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:27–33, November 1990.

[MHK95] X. Mao, L. Hong, and A. E. Kaufman. Splatting of Curvilinear volumes. In *IEEE Visualization '95*, pages 61–68, 1995.

[Mit00] T. Mitra. *Mesh Compression and Its Hardware/Software Applications*. PhD thesis, State University of New York at Stony Brook, December 2000.

[MMC99] K. Mueller, T. Moeller, and R. Crawfis. Splatting without the Blur. In *IEEE Visualization'99*, pages 363–371, October 1999.

[MSHC99a] K. Mueller, N. Shareef, J. Huang, and R. Crafis. IBR-Assisted Volume Rendering. In *Late Breaking Hot Topics session in Visualization '99*, October 1999.

[MSHC99b] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2), April 1999.

[MSS94a] C. Montani, R. Scateni, and R. Scopigno. Discretized Marching Cubes. In *IEEE Visualization '94*, pages 281–287, October 1994.

[MSS94b] C. Montani, R. Scateni, and R. Scopigno. A Modified Look-Up Table for Implicit Disambiguation of Marching Cubes. *The Visual Computer*, 10:353–355, 1994.

[Mur93] S. Muraki. Volume Data and Wavelet Transforms. *IEEE Computer Graphics and Applications*, 13(4):50–56, 1993.

[MY96] K. Mueller and R. Yagel. Fast Perspective Volume Rendering with Splatting by Utilizing a Ray-driven Approach. In *Proceedings of Visualization '96*, pages 65–72, 1996.

[NH91] G. M. Nielson and B. Hamann. The Asymptotic Decider: Removing the Ambiguity in Marching Cubes. In *IEEE Visualization '91*, pages 83–91, October 1991.

[NH92] P. Ning and L. Hesselink. Vector Quantization for Volume Rendering. In *Proceeding of the 1992 Workshop on Volume Visualization*, pages 69–74, October 1992.

[NH93] P. Ning and L. Hesselink. Fast volume Rendering of Compressed Data. In *Proceedings of Visualization '93*, pages 11–18, May 1993.

[OGS98] M. H. Gross O. G. Staadt. Progressive Tetrahedralizations. In *IEEE Visualization '98*, pages 397–402, October 1998.

[PHK+99] H. Pfister, Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *SIGGRAPH '99*, pages 251–260, 1999.

[PRS99] R. Pajarola, J. Rossignac, and A. Szymczak. Implant Sprays: Compression of Progressive Tetrahedral Mesh Connectivity. In *IEEE Visualization '99*, pages 299–305, October 1999.

[PTVF97] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1997.

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels - surface Elements as Rendering Primitives. In *SIGGRAPH '2000*, pages 335–342, 2000.

[RB93] J. Rossignac and P. Borrel. Multi-Resolution 3d Approximation for Rendering Complex Scenes. *Geometric Modeling in Computer Graphics*, pages 455–465, 1993.

[RL00] S. Rusinkiewicz and M Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *SIGGRAPH '2000*, pages 343–351, 2000.

[RO96] K. J. Renze and J. H. Oliver. Generalized Unstructured Decimation. *IEEE Computer Graphics and Applications*, 16(6):24–32, 1996.

[Sab88] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Computer Graphics*, 22(4):51–58, August 1988.

[SHLJ96] H. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in Span Space with Utmost Efficiency. In *IEEE Visualization '96*, pages 287–294, October 1996.

[SK90] D. Speray and S. Kennon. Volume Probes: Interactive Data Exploration on Arbitrary Grids. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:5–12, November 1990.

[SM97] C. Silva and J. Mitchell. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), June 1997.

[SMW98] C. Silva, J. Mitchell, and P. Williams. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In *ACM/IEEE Volume Visualization Symposium '98*, pages 87–94, 1998.

[SR99] A. Szymczak and J. Rossignac. Grow & Fold: Compression of Tetrahedral Meshes. In *Proceedings of the 5th ACM Symposium on Solid Modeling and Applications*, pages 54–64, June 1999.

[ST90] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:63–70, November 1990.

[SZK95] R. Shu, C. Zhou, and M. S. Kankanhalli. Adaptive Marching Cubes. *The Visual Computer*, 11:202–217, 1995.

[SZL92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. In *SIGGRAPH '92*, pages 65–70, July 1992.

[THJW98] I. J. Trotts, B. Hamann, K. I. Joy, and D. F. Wiley. Simplification of Tetrahedral Meshes. In *IEEE Visualization '98*, pages 287–295, October 1998.

[TL93] T. Totsuka and M. Levoy. Frequency Domain Volume Rendering. In *Computer Graphics*, pages 271–278, August 1993.

[Tur92] Greg Turk. Re-Tiling Polygonal Surfaces. In *IEEE Visualization '92*, pages 55–64, 1992.

[UK88] C. Upson and M. Keeler. V-Buffer: Visible Volume Rendering. *Computer Graphics*, 22(4):367–376, August 1988.

[WCA+90] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct Volume Rendering of Curvilinear Volumes. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:41–47, November 1990.

[Wes89] L. Westover. Interactive Volume Rendering. In *Volume Visualization Workshop (Chapel Hill, NC, May 18-19)*, pages 9–16, 1989.

[Wes90] L. Westover. Footprint Evaluation for Volume Rendering. *Computer Graphics*, 24(4):367–376, 1990.

[Wes94] R. Westermann. A Multiresolution Framework for Volume Visualization. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 51–58, October 1994.

[WG92] J. Wilhelms and A. V. Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[Wil92] P. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2), 1992.

[Wit99] C. M. Wittenbrink. CellFast: Interactive Unstructured Volume Rendering. Technical Report HPL-1999-81R1, Hewlett-Packard Labs, 1999.

[WKB99] M. Wan, A. Kaufman, and S. Bryson. High Performance Presence-Accelerated Ray Casting. In *IEEE Visualization '99*, pages 379–386, 1999.

[WM92] P. L. Williams and N. Max. A Volume Density Optical Model. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 61–68, October 1992.

[WMG98] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-Weighted Color Interpolation for Volume Sampling. Technical Report HPL-97-31R2, Hewlett-Packard Laboratories, Palo Alto, April 1998.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic View-dependent Simplification for Polygonal Models. In *IEEE Visualization '96*, pages 327–334, 1996.

[Yan89] D. Yang. New Fast Algorithm to Compute Two-Dimensional Discrete Hartley Transform. *Electronic Letters*, 25(25):1705–1706, December 1989.

[Yan00] C. Yang. Integration of Volume Compression and Visualization: A Survey. Technical report, ECSL of Computer Science Department, SUNY at Stony Brook, September 2000.

[YC01a] C. Yang and T. Chiueh. An Integrated Pipeline of Decompression, Simplification and Rendering for Irregular Volume Data. Technical report, ECSL of Computer Science Department, SUNY at Stony Brook, March 2001.

[YC01b] C. Yang and T. Chiueh. I/O Conscious Volume Rendering. In *Proceedings of VisSym01*, 2001.

[YCS98] C.T. Silva Y. Chiang and W.J. Schroeder. Interactive Out-of-Core Isosurface Extraction. In *IEEE Visualization '98*, pages 167–174, October 1998.

[YL95] B. Yeo and B. Liu. Volume Rendering of Dct-Based Compressed 3D Scalar Data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43, March 1995.

[YMC00] C. Yang, T. Mitra, and T. Chiueh. On-the-Fly Rendering of Losslessly Compressed Irregular Volume Data. In *Proceedings on Visualization '2000*, October 2000.

[YRL+96] R. Yagel, D. Reed, A. Law, P-W. Shih, and N. Shareef. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. *IEEE-ACM Volume Visualization Symposium*, pages 55–62, November 1996.