On Enhancing the Speed of Splatting with Indexing

Insung Ihm Rae Kyoung Lee

Department of Computer Science
Sogang University
Seoul, Korea

Abstract

Splatting is an object-space direct volume rendering algorithm that produces images of high quality, but is computationally expensive like many other volume rendering algorithms. This paper presents a new technique that enhances the speed of splatting without trading off image quality. This new method reduces rendering time by employing a simple indexing mechanism which allows to visit and splat only the voxels of interest. It is shown that this algorithm is suitable for the dynamic situation in which viewing parameters and opacity transfer functions change interactively. We report experimental results on several test data sets of useful size and complexity, and discuss the cost/benefit trade-off of our method.

1 Introduction

Scientific visualization is a fast growing area which is concerned with various techniques that help scientists and engineers to extract meaningful and visual information from the results of simulations and experimentations [2]. One of the most actively researched subfields of scientific visualization is volume rendering that deals with scalar and vector data defined on three (or higher) dimensional grids. In this relatively new field several rendering techniques have emerged to analyze, understand, and render objects that are contained in volume data.

Some of the most commonly used direct volume rendering algorithms are ray casting [5, 8, 10], splatting [11], and volume shearing [1, 3]. Splatting is an object-order traversal algorithm where voxels are *properly splatted* into an image plane. In this algorithm, the voxels are sorted slice by slice in the front-to-back or back-to-front order. Each voxel, traversed in the order, is classified and shaded by given opacity and color transfer functions. Then, the voxel is projected into an image plane, and its contribution is accumulated to an image buffer using a projected reconstruction kernel called *footprint*. In this way, successive slices are composited to produce the final image. Splatting is, like many other direct volume rendering algorithms, computationally expensive due to the large size of volume data although it is known to be more efficient than ray casting or volume shearing.

Since the original splatting algorithm traverses all the voxels in some proper order, the computational cost is linearly proportional to the size of the volumetric data set, regardless of its contents. Often, only small portions of volumetric data sets contain objects to be rendered. For example, a large portion of typical CT or MRI data contains air which is seldom rendered. Hence, it is quite likely that computing time can be saved by splatting only those voxels that correspond to objects which are being rendered. It has been reported that volume data with 70-90% of uninteresting voxel points are not uncommon [6, 13]. Spatial data structures like octrees and pyramids were used in encoding volume data so that unnecessary computation in transparent regions can be avoided [6, 9, 13, 1, 4, 7].

Unlike ray tracing or other rendering algorithms, little has been done to reduce rendering time of the splatting method on a uniprocessor while a few algorithms are known for parallel processors [12]. In [4], a pyramidal volume representation was used to improve the speed of splatting. Given transfer functions and view-independent shading functions, an octree is constructed in which each node contains the average RGBA value of all its children and a value indicating the average error associated with the average. Then, the octree is traversed in a viewing order to splat the voxels, depending on the given allowable error that determines the refinement of rendered images. This algorithm reduces rendering costs, but trades off im-

age quality for speed.

This paper presents a new technique that enhances the speed of splatting without sacrificing image quality. In this algorithm, a simple indexing mechanism is employed so that only the voxels of interest are examined and splatted. The data structure used in our algorithm is simpler than octrees, and the traversal order of voxels is basically the same as the traditional splatting algorithm, hence any improvement techniques, such as early termination of composition process, can be easily applied.

Observe that the order in which voxels are stored in an array for *raw* volume data is determined by their positions which, in general, have nothing to do with their density values, that is, the materials which exist at the corresponding grid points. On the other hand, most rendering algorithms use opacity transfer functions for classification, which decide the range of densities of interesting voxels. Hence, the volume data must be stored or at least be accessible efficiently depending on the density values rather than being stored in terms of voxel's positions. In our algorithm, we preprocess the volume data to extract indexing information which enables efficient access to voxels having arbitrary density values. The extracted indexing information is both view-independent and transfer-function-independent. This means that it is built only once in the preprocessing step, and splatting is done efficiently with dynamically changing viewing parameters and transfer functions.

In Section 2, we propose a new way of splatting that enhances the speed of rendering. In Section 3, we show how viewing transformations are performed efficiently in our algorithm. Experimental results are reported in Section 4, and we discuss some of the implication of our algorithm in Section 5.

2 The Indexed Splatting Algorithm

2.1 Data Structure for Fast Voxel Indexing

Fundamental to our speed-enhancing algorithm is the ability to access easily the voxels of interest with arbitrary density values. In this section, we describe the data structure employed for fast voxel indexing. Consider a $n_i \times n_j \times n_k$ volume data set. First, we construct, as many volume rendering algorithms do for the purpose of enhancing the speed, an *augmented* volume from the *raw* volume data, where each *augmented* voxel is made of four bytes : one for density value, one for gradient magnitude, and two for encoded normal.

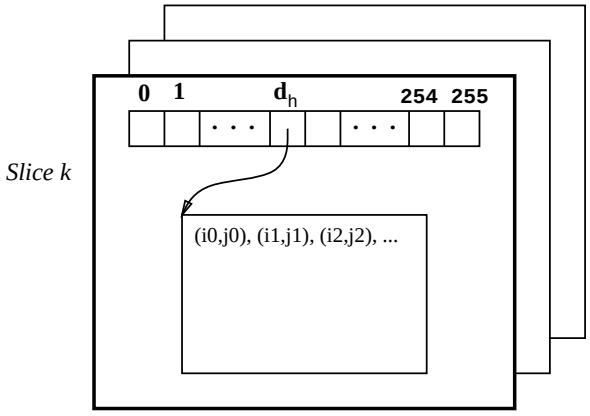


Figure 1: Data Structure for Indexing

Then, an additional data structure, that performs the function of indexing, are precomputed as follows : for each principal axis of the volume data (without loss of generality, consider the k axis.), imagine the series of slices, perpendicular to the axis. For each slice, a sequence of pointers corresponding to the density values d_h that appear in the slice, is enumerated in the increasing order of density values (See Figure 1.). The pointer associated with density d_h is to a data block that keeps the list of the (i, j) indices to the voxels in the slice which have the density value d_h. (In Section 2.3, we describe how these indices are encoded in our implementation.)

Building this data structure for indexing can be viewed as reorganizing the given volume data based on the density values, not the positions of voxels. In most volume rendering algorithms, objects of interest to be rendered are classified in terms of opacity transfer functions. Our new data structure enables us to easily access only those voxels that fall in the ranges of interest which are specified in terms of the opacity transfer functions.

To make the splatting process view-independent, three sets of such a data structure are built in the preprocessing step, one for each principal axis. Given arbitrary viewing parameters (that is, the view reference point, the view-up vector, and the view plane normal), the proper viewing direction is selected so voxels are visited slice by slice in the proper front-to-back order.

2.2 Algorithm

Now, our new splatting algorithm works as follows : assume that we want to render some objects whose density values range from d_{low} to d_{up}. First, the principal axis for the viewing direction is determined among the i, $-i$, j, $-j$, k, and $-k$ axes. (Again, without loss of generality, let's assume the viewing direction is the k axis.) Once the viewing direction is decided, each slice perpendicular to the viewing direction is traversed in the front-to-back order as the original splatting algorithm does. For each slice, we first find the actual lower and the upper bounds quickly, using the binary search, of density values that are between the interval $[d_{low}, d_{up}]$, then, follow the pointers in between to visit only the voxels with values of interest. The remaining steps such as shading, projection, and color blending are the same as the traditional splatting algorithm.

Note that the traversal order of voxels in splatting is basically the same as the traditional splatting algorithm, hence any enhancement techniques, such as early termination of composition process, can be easily applied. The resulting algorithm inherits the merits of the original splatting algorithm as well as it has the following additional characteristics :

- Splatting time is saved by accessing only the necessary voxels.
- The opacity transfer functions can change interactively without building the indexing data structure again.
- The viewing and shading parameters can be modified dynamically without building the indexing data structure again.

2.3 Details on the Indexing Data Structures

The indexed splatting algorithm saves the rendering time by traversing only the voxel points of interest. However, extra memory is necessary to store the information on indexing. It could lead to an enormous memory requirement if inefficient data structures were used to store indexing information. On the other hand, any space-saving encoding method to store the information efficiently should not be complicated enough to waste the time that is saved by indexing. In our current implementation that permits volume data of size up to $256 \times 256 \times 256$, the following simple encoding method for indexing is used. For the k-th slice, and for the density value d, the memory block, that contains the indices (i, j) of (i, j, k) whose density is d, is organized as follows : first, each voxel (i, j) is associated with an index value ind_{ij} that is defined as $ind_{ij} = i + j * n_x$. Then consider the enumeration of voxel points in the scan-line order. Rather than store the i and j pair for each voxel (i, j) consuming two bytes, we store the offset value off that is the displacement from the predecessor. One byte is used for the offset, which means that distance at most 255 is possible. When the offset is greater than 255, one byte containing zero (null value) is put, and followed by two bytes for the i and j index of the next voxel. Hence, one byte is necessary when the next voxel with density d is within 255 voxel off, or three bytes are required. We observe that due to the property of spatial coherence of volume data this encoding technique results in saving the total number of bytes for the index information.

3 Efficient Viewing Transformations

Viewing transformation is the process in which a grid point (x, y, z) in the voxel-space is mapped to a point (u, v) in the image-space. Efficient computation of the transformation is essential since a huge number of voxels are usually projected into an image plane. To probe objects in volume data interactively, we often change viewing parameters dynamically. Hence, the viewing transformation for arbitrary viewing parameters must be calculated efficiently as well as each voxel is projected quickly.

In our indexed splatting algorithm, the voxel points are not visited in a regular manner as in the original splatting method (for example, either i, j, or k direction fastest, one of the remaining two directions second fastest, and the last remaining direction slowest.). Note that we visit only the voxels with densities between ranges of interest. Since the traversing order is rather arbitrary, the viewing transformation can not be done incrementally as proposed in [12].

In this section, we are concerned with efficient computations of two tasks : one is to compute the viewing transformation given arbitrary viewing parameters, and the other is to actually project each voxel into the image plane once the viewing transformation is set. Remind that computing the transformation and projecting a voxel correspond to building a 4×4 transformation matrix, and multiplying it by a 4-vector, respectively.

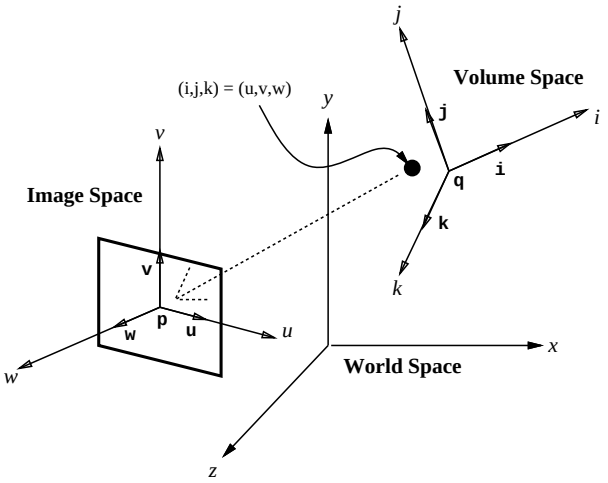


Figure 2: Orthographic Transformations

3.1 Orthographic Projections

In Figure 2, it is illustrated how orthographic projections of an arbitrary voxel point (i, j, k) into the image plane are carried out. In our viewing transformation scheme, we assume that the volume data can exist anywhere with arbitrary orientations, and the image plane can be defined by arbitrary viewing parameters. This assumption is well-suited to the dynamical situation in which both camera and objects move freely in the world coordinates (x, y, z). First, the image-space (u, v, w) is specified by a point $\mathbf{p} = (p_x, p_y, p_z)^t$, and three orthonormal vectors $\mathbf{u} = (u_x, u_y, u_z)^t$, $\mathbf{v} = (v_x, v_y, v_z)^t$, and $\mathbf{w} = (w_x, w_y, w_z)^t$. Notice that $\mathbf{p}$ is the view reference point, and $\mathbf{v}$ and $\mathbf{w}$ are the view-up vector and view plane normal, respectively. Secondly, the voxel points are expressed with respect to the volume space (i, j, k) defined by a point $\mathbf{q} = (q_x, q_y, q_z)^t$ and three orthonormal vectors $\mathbf{i} = (i_x, i_y, i_z)^t$, $\mathbf{j} = (j_x, j_y, j_z)^t$, and $\mathbf{k} = (k_x, k_y, k_z)^t$.

Now what we would like to do is to orthographically map a voxel point (i, j, k) into the corresponding point (u, v) on the image plane. The voxel point is, in fact, $\mathbf{q} + i \cdot \mathbf{i} + j \cdot \mathbf{j} + k \cdot \mathbf{k}$ in the world space, which is also expressed as $\mathbf{p} + u \cdot \mathbf{u} + v \cdot \mathbf{v} + w \cdot \mathbf{w}$ for the unknowns u, v and w. Equating these two expressions, we are led to

$$\begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} \end{pmatrix} \begin{pmatrix} u \\ v \\ w \end{pmatrix} = (\mathbf{q} - \mathbf{p}) + \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

Since $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are orthonormal, (u, v, w) is expressed as

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} \mathbf{u}^t \\ \mathbf{v}^t \\ \mathbf{w}^t \end{pmatrix} (\mathbf{q} - \mathbf{p}) + \begin{pmatrix} \mathbf{u}^t \\ \mathbf{v}^t \\ \mathbf{w}^t \end{pmatrix} \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

From this, we can get the (u, v) point on the image plane after the orthographic projection as follows :

$$\begin{aligned} u &= \mathbf{u} \circ (\mathbf{q} - \mathbf{p}) + i \cdot (\mathbf{u} \circ \mathbf{i}) + j \cdot (\mathbf{u} \circ \mathbf{j}) + k \cdot (\mathbf{u} \circ \mathbf{k}) \\ v &= \mathbf{v} \circ (\mathbf{q} - \mathbf{p}) + i \cdot (\mathbf{v} \circ \mathbf{i}) + j \cdot (\mathbf{v} \circ \mathbf{j}) + k \cdot (\mathbf{v} \circ \mathbf{k}) \end{aligned}$$

where $\circ$ is the inner product of two vectors.

Now imagine the rendering process : when a user set the viewing parameters before starting splatting, $(\mathbf{q}-\mathbf{p})\circ\mathbf{u}$, $(\mathbf{q}-\mathbf{p})\circ\mathbf{v}$, $\mathbf{u}\circ\mathbf{i}$, $\mathbf{u}\circ\mathbf{j}$, $\mathbf{u}\circ\mathbf{k}$, $\mathbf{v}\circ\mathbf{i}$, $\mathbf{v}\circ\mathbf{j}$, and $\mathbf{v}\circ\mathbf{k}$ are computed only once, and $k \cdot (\mathbf{u} \circ \mathbf{k})$ and $k \cdot (\mathbf{v} \circ \mathbf{k})$ are added to u and v for the initial k (k could be zero or $n_k - 1$.). Under the assumption that the viewing direction is the k axis, the slices perpendicular to this axis is examined one by one in the front-to-back order. For each slice k_s, k is fixed, so $k_s \cdot (\mathbf{u} \circ \mathbf{k})$ and $k_s \cdot (\mathbf{v} \circ \mathbf{k})$ are calculated once per slice. Note that since k is uniformly increased or decreased, these values can be computed incrementally in two additions. Then, for each varying (i, j) of (i, j, k_s), (u, v) is computed using 4 additions and 4 multiplications. Thus we see that the total cost of the viewing transformation for each rendering is $6+2n_k+4n_{voxel}$ additions and $24+4n_{voxel}$ multiplications, where n_{voxel} is the number of voxels that are actually projected.

3.2 Perspective Projections

While most volume rendering techniques assume orthographic projections due to their simplicity, perspective projections are also useful particularly because they provide depth-cue information. In this subsection, we show that computations for perspective transformations are slightly more expensive than orthographic transformations, but can be done in a similar way.

For perspective projections, another viewing parameter $\mathbf{e} = (e_x, e_y, e_z)^t$ is specified corresponding to the projection reference point (See Figure 3.). The

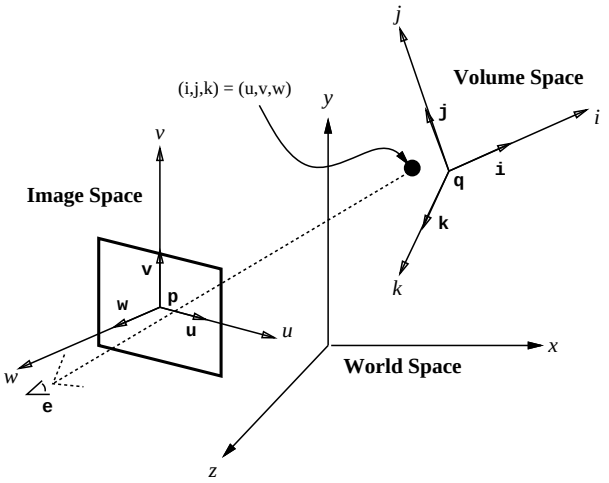


Figure 3: Perspective Transformations

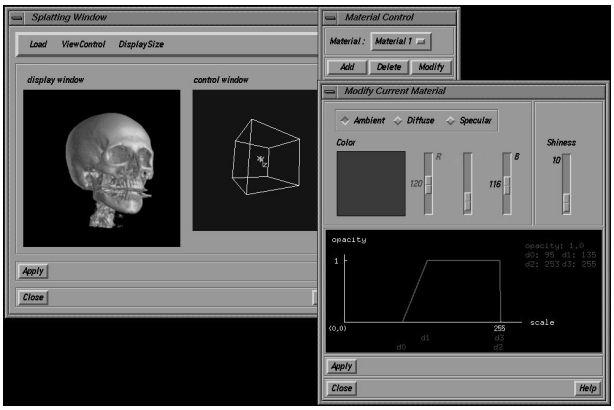


Figure 4: An Example SGVR Session

projector connecting $\mathbf{e}$ and a voxel (i, j, k) can be expressed parametrically as $\mathbf{L}(t) = \mathbf{e}\cdot(1-t)+(\mathbf{q}+i\cdot\mathbf{i}+j\cdot\mathbf{j}+k\cdot\mathbf{k})\cdot t = (\mathbf{q}-\mathbf{e}+i\cdot\mathbf{i}+j\cdot\mathbf{j}+k\cdot\mathbf{k})\cdot t+\mathbf{e}$. Then, for the parameter value t_0 corresponding to the intersection of the projector with the image plane, $(\mathbf{L}(t_0)-\mathbf{p})\circ\mathbf{w} = 0$, and solving it for t_0 results in

$$t_0 = \frac{\mathbf{w}\circ(\mathbf{p}-\mathbf{e})}{\mathbf{w}\circ(\mathbf{q}-\mathbf{e}) + i\cdot(\mathbf{w}\circ\mathbf{i}) + j\cdot(\mathbf{w}\circ\mathbf{j}) + k\cdot(\mathbf{w}\circ\mathbf{k})}$$

As in orthographic projections, the intersection point can be expressed for the unknowns u, v, and w as

$$\begin{aligned}\mathbf{p} + u\cdot\mathbf{u} + v\cdot\mathbf{v} + w\cdot\mathbf{w} &= \{(\mathbf{q}-\mathbf{e}) + i\cdot\mathbf{i} + j\cdot\mathbf{j} \\ &\quad + k\cdot\mathbf{k}\}\cdot t_0 + \mathbf{e}\end{aligned}$$

From this, we can show that the mapped point (u, v) (w must be zero.) is computed as follows :

$$\begin{aligned}u &= \{\mathbf{u}\circ(\mathbf{q}-\mathbf{e}) + i\cdot(\mathbf{u}\circ\mathbf{i}) + j\cdot(\mathbf{u}\circ\mathbf{j}) \\ &\quad + k\cdot(\mathbf{u}\circ\mathbf{k})\}\cdot t_0 - \mathbf{u}\circ(\mathbf{p}-\mathbf{e}) \\ v &= \{\mathbf{v}\circ(\mathbf{q}-\mathbf{e}) + i\cdot(\mathbf{v}\circ\mathbf{i}) + j\cdot(\mathbf{v}\circ\mathbf{j}) \\ &\quad + k\cdot(\mathbf{v}\circ\mathbf{k})\}\cdot t_0 - \mathbf{v}\circ(\mathbf{p}-\mathbf{e})\end{aligned}$$

Again, when a user set the viewing parameters before starting rendering, $\mathbf{u}\circ(\mathbf{q}-\mathbf{e})$, $\mathbf{u}\circ\mathbf{i}$, $\mathbf{u}\circ\mathbf{j}$, $\mathbf{u}\circ\mathbf{k}$, $\mathbf{u}\circ(\mathbf{p}-\mathbf{e})$, $\mathbf{v}\circ(\mathbf{q}-\mathbf{e})$, $\mathbf{v}\circ\mathbf{i}$, $\mathbf{v}\circ\mathbf{j}$, $\mathbf{v}\circ\mathbf{k}$, $\mathbf{v}\circ(\mathbf{p}-\mathbf{e})$, $\mathbf{w}\circ(\mathbf{q}-\mathbf{e})$, $\mathbf{w}\circ\mathbf{i}$, $\mathbf{w}\circ\mathbf{j}$, $\mathbf{w}\circ\mathbf{k}$, and $\mathbf{w}\circ(\mathbf{p}-\mathbf{e})$ are computed once. For each slice k_s, three incremental additions are necessary for the denominator of t_0, u, and v. For each varying (i, j) of (i, j, k_s), t_0 is computed using two additions, two multiplications, and one division, and then u and v are computed in two additions, one subtraction, and three multiplications, respectively.

In case of perspective projections, the depth information of the voxel point being projected is often important. In the splatting algorithm, this information could be used to select the proper size of footprint tables adaptively. When the projection reference point $\mathbf{e}$ is on the w axis, the value $w = \mathbf{w}\circ(\mathbf{q}-\mathbf{p}) + i\cdot(\mathbf{w}\circ\mathbf{i}) + j\cdot(\mathbf{w}\circ\mathbf{j}) + k\cdot(\mathbf{w}\circ\mathbf{k})$ from orthographic projections will be a good choice.

4 Experimental Results

Both original and indexed splatting algorithms have been implemented, and added to our SGVR (SoGang Volume Rendering) scientific visualization system (Figure 4). The performance results for the two algorithms are summarized in Figure 5. We have experimented with five data sets to generate RGB images of size 256×256 using orthographic projections. The test cases H1 (head) and H2 (head with a cutting plane) are from the "UNC head" data set that is $256 \times 256 \times 225$ CT scan of a human head (Figure 6 (a) and (b)), and the case B1 (brain) is from the "UNC brain" data set which is $256 \times 256 \times 167$ MRI scan of a human head (Figure 7(b)). HD (decimated head) and BD (decimated brain) are tested using the decimated versions of the head and the brain data sets with the resolutions $128\times128\times113$ and $128\times128\times84$, respectively. Two more cases E1 (engine block) and E2 (engine part) were tested with the $256 \times 256 \times 110$ "engine block" data set (Figure 8 (a) and (b)).

The timings were measured on an SGI Workstation

(Ave. : Sec., Mem. : Mbyte)

Data	Resolution	Ratio	No Indexing		Indexing		Speedup
			Ave.	Mem.	Ave.	Mem.	
H1	$256 \times 256 \times 225$	0.072	28.31	56.3	5.01	62.7	82.3%
H2	$256 \times 256 \times 225$	0.072	25.88	56.3	4.55	62.7	82.4%
HD	$128 \times 128 \times 113$	0.067	4.39	7.1	1.43	8.0	67.4%
B1	$256 \times 256 \times 167$	0.17	24.27	41.7	6.24	50.6	74.3%
BD	$128 \times 128 \times 84$	0.17	3.49	5.25	1.37	6.45	60.7%
E1	$256 \times 256 \times 110$	0.16	20.28	27.5	8.55	33.0	57.8%
E2	$256 \times 256 \times 110$	0.012	13.16	27.5	0.57	33.0	95.7%

Figure 5: Comparisons of Two Algorithms

with a 150MHz R4400 CPU and 96 Mbytes of memory, and the timing results are given for both algorithms along with the memory requirements. Also given are the spatial ratios of the number of voxels that were actually visited to the whole number of voxels in the data sets. These ratios indicate how much space objects of interest, that is, objects we want to render, take with respect to the whole volume space, and is the major factor in the performance of the indexed splatting algorithm. In our implementation, four bytes are used to store one voxel : one for density, two for encoded normal, and one for gradient size. Precomputation of the encoded normal vectors and gradient sizes allows us to perform fast classification and shading. For the $256 \times 256 \times 225$ head data (H1), the number of voxels is about 14.1 millions, and hence about 56.3 Mbytes of memory are necessary for splatting without indexing.

On the other hand, about 62.7 Mbytes of memory is needed, so we see that roughly 6.4 Mbytes of additional memory are consumed for the indexing data structure. We have to mention that the amount of required memory for indexing depends on the materials residing in data sets. In the test with H1, only the bones and muscles were loaded for indexing. When skins were loaded, about 20 Mbytes of memory were used. In the simpler data set E1, only 5.5 Mbytes of memory were required in loading the whole indices. We are now focusing on the development of a better scheme that encodes data blocks for (i, j) indices efficiently without harming the timing performance of indexing too much.

The indexing algorithm produces $57.8 - 95.7\%$ of time reduction, which obviously depends on the test data. In the best (E2) and worst (E1) cases, the indexing method improved rendering speed by factors of about 23.1 and 2.4, respectively. In case of perspective projection, it took roughly 1.2 times longer than orthographic projection, although it also relies on the data sets. Notice the somewhat exaggerated effect of perspective projection for the head data in Figure 7 (a).

The timing results reveal several interesting facts about our enhanced algorithm. First, we see that the spatial ratio, that actually decides how many voxels are visited, is somewhat small, and that the speedup tends to be better as the ratio gets smaller (Compare E1 and E2 to see this.). This observation is clearly understood since the new splatting algorithm was designed to visit the rendered objects only.

Secondly, the spatial ratio is not the only factor that affects the performance. That is, the reduction in rendering time obtained by indexing is highly dependent on the scenes. Consider B1 and E1 that demonstrate the effect of semitransparent objects on the performance of our algorithm. Although they have almost the same spatial ratios, the timings for the two data sets are somewhat different. The main factor that made differences is the opacity transfer functions. Since a large portion of engine block (E1) is semitransparent, almost all voxels are processed. In case of B1, lots of the voxels in the behind are blocked by ones in the front without being splatted, resulting in fast rendering.

Consider how the original splatting algorithm spends rendering time. It first visits all the voxels in a volume data set one by one, doing classification of each voxel. It is shaded, and projected if it turns out to be one of the voxels of interest. Then, resampling is performed using a footprint table. The reduction in splatting time of our algorithm is made possible by not visiting unnecessary voxels using the indexing mechanism. Although just visiting and classifying each voxel takes very little time compared to the shading and resampling precesses, it is found out that accumulation of such a small effect for the whole volume data makes a prominent difference.

5 Closing Remarks

We have proposed a new technique that enhances the speed of splatting without trading off image quality. This new method saves rendering time by employing a simple indexing mechanism so that only voxels of interest are processed in splatting. We observe that the proposed data structure for indexing is suitable for the dynamic situation in which viewing parameters and opacity transfer functions change interactively. Tests with several data sets of useful sizes and complexities showed $57.8 - 95.7\%$ of time reduction at the reasonable cost of additional memory for indexing.

Currently, we are applying the data structure for indexing to volume ray tracing, and the preliminary results indicate a significant rendering time reduction.

Acknowledgments

We are grateful to Mr. Philippe Lacroute who helped get the test data sets.

References

[1] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of the 1992 Workshop on Volume Visualization*, pages 91–106, Boston, 1992.

[2] A. Kaufman, editor. *Introduction to Volume Visualization*. IEEE Computer Society Press, 1991.

[3] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(4):451–458, 1994.

[4] D. Laur and P. Hanrahan. Hierarchical splattings : A progressive refinement algorithm for volume rendering. *Computer Graphics*, 25(4):285–288, 1991.

[5] M. Levoy. Display of suirface from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[6] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[7] D. Meagher. Efficient synthetic image generation of arbitrary 3-d objects. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, pages 473–478, 1982.

[8] P. Sabella. A rendering algorithm for 3d scalar fields. *Computer Graphics*, 22(4):51–58, 1988.

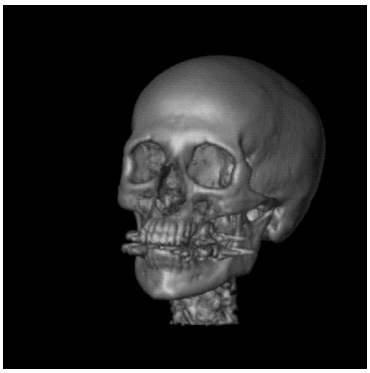
[9] K. Subramaanian and D. Fussell. Applying space subdivision techniques to volume rendering. In *Proceedings of Visualization '90*, pages 150–159, San Francisco, 1990.

[10] C. Upson and M. Keeler. V-buffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, 1988.

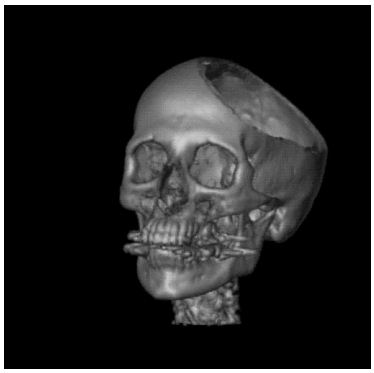
[11] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, 1990.

[12] L. Westover. *Splatting - A parallel, feed-forward volume rendering algorithm*. PhD thesis, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, July 1991.

[13] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

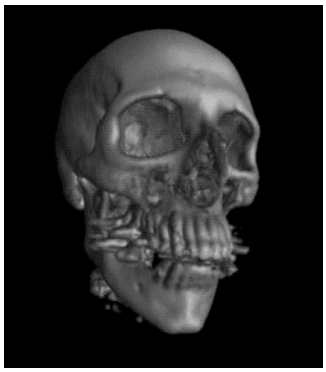


(a) human head



(b) human head cut by a plane

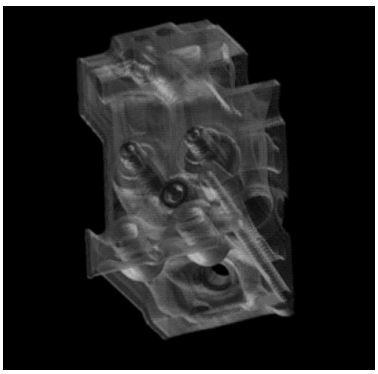
Figure 6: CT Human Head



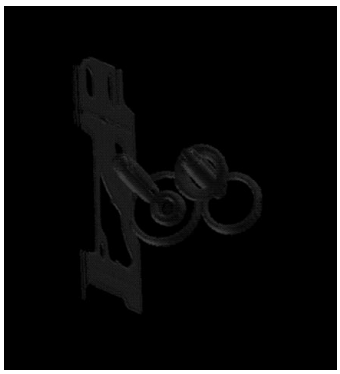
(a) perspective human head

(b) human brain

Figure 7: Perspective CT Head and MRI Human Brain



(a) engine block



(b) part of engine

Figure 8: Engine Block