Accelerated Splatting using a 3D Adjacency Data Structure

Jeff Orchard, Torsten Möller

School of Computing Science
Simon Fraser University, Burnaby, British Columbia, Canada
{jjo, torsten}@cs.sfu.ca

Abstract

We introduce a new acceleration to the standard splatting volume rendering algorithm. Our method achieves full colour (32-bit), depth-sorted and shaded volume rendering significantly faster than standard splatting. The speedup is due to a 3-dimensional adjacency data structure that efficiently skips transparent parts of the data and stores only the voxels that are potentially visible. Our algorithm is robust and flexible, allowing for depth sorting of the data, including correct back-to-front ordering for perspective projections. This makes interactive splatting possible for applications such as medical visualizations that rely on structure and depth information.

Key words: volume rendering, splatting, data structure, software acceleration

1. Introduction

The simplification of three-dimensional (3D) volumetric data sets is a critical step in the exploration of the underlying data. The display of key features of the volume (e.g. iso-regions for scalar data sets [5, 16], stream-lines, stream-surfaces and stream-volumes for vector data sets [2]) allows for better comprehension of a 2D projection of the volume. Rendering only those key features allows faster drawing speeds, and also helps prevent important information from being obscured by less important clutter. This enables interaction with the volume, enhancing the 3D understanding of the data.

Iso-surface extraction, as introduced by Lorensen et. al. [16] and other researchers [10, 31] is still one of the most popular visualization techniques for the display of volumetric data sets. It displays a 3D object by rendering a surface that represents a constant intensity. This might include one or two transparent or translucent surfaces. However, iso-surface extraction cannot display certain qualities such as light attenuation due to opacity and depth. These can only be accomplished through direct volume rendering algorithms.

How do we efficiently skip the unnecessary part of the data while still maintaining the full context of the part we want to see? This paper answers that question insofar as it pertains to splatting algorithms.

2. Previous Work

A variety of different volume rendering algorithms have been proposed. There are five different concepts for direct volume rendering: ray-casting [6, 14, 29], the shear-warp algorithm [12], splatting [34], fourier domain volume rendering [17, 33], and texture slicing [3]. A comprehensive comparison of most of these algorithms was conducted by Meißner et. al. [20].

One of the most common image-based algorithms for direct volume rendering is ray-casting, in which the colour and intensity of a pixel on the viewplane is determined by following a ray of light through the volume. Ray-casting constitutes a flexible and robust numerical integration of the rendering integral [19] with a user-defined step size. Hence it is easy to achieve very accurate, high-quality images. However, ray-casting is also relatively slow. Many suggestions have been made to speed up this algorithm. Early ray termination [14] avoids the traversal of regions of the volume that are obscured by other data. The use of coherency as suggested for traditional ray-tracing algorithms [7] was successfully implemented for volumetric ray-casting. This resulted in space-leaping [35], template based ray-casting [36] and the implementation of bounding boxes [30]. All of these techniques thrive on the idea of quickly disregarding irrelevant or "empty" voxels. Despite these advances, ray-casting is regarded as one of the slowest volume rendering algorithms.

Mitsubishi's VolumePro is a special purpose hardware implementation of the ray-casting algorithm on a single circuit board [26]. It uses 4 parallel processing pipelines to perform all the necessary operations to achieve 30 frames per second for a 512x512x512 (12-bit) volume. While its speed is astounding, it is hindered by its lack of flexibility. Currently, it is only able to perform maximum and minimum intensity projections. It is also limited to parallel projections.

The UltraVis system by Knittel [11] is an assembler implementation of ray-casting which makes efficient

use of the MMX and SSE instruction sets of the Intel Pentium III. It achieves amazing frame rates for a ray-casting application, but is unfortunately limited to a very specific type of CPU. Knittel notes that one of the bottlenecks of volume rendering is the limited bandwidth between the CPU, the graphics card, and main memory. He designs a data structure that is wasteful in its overall memory requirement but optimal in its cache coherency. He demonstrates that his data structure produces a cache hit rate of over 99% and claims this is a major factor for the high performance of his system.

General purpose graphics hardware, including hardware texturing implementations, has become popular and powerful in the last five years. Hence the use of texturing hardware for volume rendering has become a key aspect for interactive applications. Cabral et. al. [3] suggested loading the entire volume into specialized video memory where the graphics processor can operate on it efficiently. The volume is resliced so that its slice planes are parallel to the view plane, and its scan lines are aligned with those of the view plane. This requires 3D texture memory which is currently uncommon. Using multi-texturing, Rezk-Salama et. al. [27] proposed a solution to the aliasing problems that occur for 2D texture memory that cannot support view-aligned re-slicing of the data. Although data slicing in hardware texture memory allows for interactive frame rates, the limited precision hardware operations yield poor quality [20].

The shear-warp algorithm of Lacroute and Levoy [12] is a software implementation of the ray-casting method. It achieves fast rendering times by employing a run-length encoding of the data and by decompositing the transformation matrix into 2D shears. The algorithm maintains an opacity value for each screen pixel, terminating the line integral calculation once the opacity exceeds a given threshold. The data structure then excludes these pixels from subsequent scan line processing. Since the shear-warp algorithm gains speed by reducing the interpolation operations to an efficient 2D interpolation, the accuracy of the algorithm is sacrificed. Hence the quality of the images produced by the shear-warp algorithm has been found inadequate in a recent comparative study [20].

Fourier domain volume rendering, introduced by Malzbender [17], and extended by Totsuka and Levoy [33], is a direct volume rendering algorithm that is significantly different from all the others mentioned above. It reduces the algorithmic complexity of rendering by traversing the volume in the frequency domain. By the Fourier projection-slice theorem [1], one can avoid scanning the whole volume by simply taking a single slice from the frequency domain representation of the volume. The slice needs to be inverse-Fourier transformed to arrive at an X-ray type projection of the volume. The resulting algorithmic complexity becomes $O(N^2 \log N)$, whereas visiting every voxel just once has algorithmic complexity $O(N^3)$. By premultiplying the frequency data, one can also achieve depth shading and diffuse lighting effects. The idea of Fourier volume rendering was combined with the wavelet transform by Lippert and Gross [15] as a multi-resolution acceleration to the algorithm. Their method also benefits from the fact that the Fourier transforms of the wavelets and scaling functions can be done analytically. However, all of the Fourier volume rendering algorithms are limited to parallel projections and X-ray type rendering. Since the accuracy of the slicing operation has tremendous effects on the image quality, hardware accelerated slicing may cause unwanted artefacts.

Splatting, an object-based direct volume rendering method developed by Westover [34], reverses the interpolation and compositing steps of the volume rendering pipeline efficiently. The principle of his algorithm is to place interpolation kernels at the center of each voxel which is then "splatted" onto the screen. Although inaccuracies result that become visible as "popping" artifacts [22, 23], Mueller et. al. [20, 21] have shown that through a sheet-buffer implementation of splatting, the accuracy of this algorithm can be competitive to ray-casting. Mueller [24] also demonstrates an efficient perspective implementation of splatting.

Crawfis et. al. [4] proposed a fast implementation of splatting using rendering hardware. Later, Crawfis introduced an implementation using a special data structure, assuring that only splats in a certain iso-range are rendered [5]. Though fast, the technique suffers from the fact that no depth sorting is done, and sorting "on-the-fly" would slow down the algorithm making interactive manipulation impossible. While his technique proves effective for constant-colour cloud-like volumes, it cannot be used for visualizations where depth information and shape preserving rendering is necessary.

Laur and Hanrahan [13] suggested the use of octrees to store the data in a hierarchical form. This data structure creates a more efficient way to traverse and splat the voxels. Their technique recursively divides the data into smaller blocks, and thus forms a tree structure. If an entire subtree of voxels have the same intensity, they can be lumped together and splatted more efficiently. However, non-visible voxels are not always organized

in such a spatially coherent way. Furthermore, this algorithm produces artifacts at the "seams" of the octree-nodes. The data structure proposed in this paper does not suffer from these drawbacks, and outperforms the octree implementation for practical scenarios (see Section 4).

Besides ray-casting, splatting is the only rendering technique that is not restricted to regular grids and orthogonal projections [32]. Extensions to curvilinear and even irregular grids have been introduced in the past [18]. Just recently, splatting-like ideas have also been implemented to render polygonal objects with high polygon count [25, 28]. Unlike ray-casting, splatting can be accelerated using commonly available texture mapping hardware. Hence, we believe splatting is among the most powerful algorithms for rendering, and research into faster and better splatting algorithms will have a substantial impact.

The method proposed in this paper takes ideas from a number of techniques, including the hardware accelerated splatting of Crawfis, the skipping of empty cells like space-leaping, and a 3D data structure like that of Frisken-Gibson [8]. Our goal is to integrate these techniques to produce high quality 3D direct volume renderings with interactive frame rates.

3. Adjacency Data Structure

Frisken-Gibson [8] used a three-dimensional linked list data structure, which she referred to as a linked volume. Her goal was to model the behaviour of elastic materials and to mimic tissue properties for surgical simulation. Volume elements have links to adjacent elements, signifying physical proximity and an elastic force interaction between them. The volume can be cut, breaking these connections, while new volume elements are dynamically added to smooth the cut edges. From each volume element, one can determine its neighbours by following the adjacency pointers. The volume elements need not be aligned on a grid.

Our rendering algorithm uses a similar linked list data structure to that of Frisken-Gibson. However, our links represent empty space, and do not necessarily link elements that are close together. The motivation of our data structure is rendering speed and encoding compression, while hers is efficient and flexible physical volume modelling.

We assume that the scalar volume data is organized on a regular rectangular grid. An opacity value is assigned to every voxel based on the transfer function of Levoy [14]. Many voxels will have an opacity of zero. Splatting these voxels offers no advantage since they have no effect on the final image. Hence, we endeavour to efficiently render only those voxels that are visible, and skip those with zero opacity, while maintaining the spatial context of each voxel.

The identification of visible voxels and invisible voxels is carried out in a pre-processing step. For the purpose of this paper, we will define a voxel as "visible" if its opacity is non-zero. Thus far, the algorithm resembles that of Crawfis [5]. However, we extend this method by inserting the visible voxels into a single 3D data structure that holds all the information needed to render the voxels in back-to-front scan-plane order from any angle. Voxels point along scan lines to the next visible voxel, thereby allowing the rendering algorithm to skip the invisible voxels. The data structure is similar to the 3D linked volume used by Frisken-Gibson [8] since it involves links between voxels. The data structure also resembles the run-length encoding used in the shear-warp algorithm proposed by Lacroute and Levoy [12].

Rendering an image of the volume simply involves traversal of the data structure in scan line order from back to front, visiting only those voxels that we wish to splat. The data structure allows for efficient depth sorting with little overhead, thus maintaining the ability to shade the image without sacrificing speed.

If the set of visible voxels changes, the data structure can be updated dynamically to add or remove voxels with minimal overhead. For example, the viewer may wish to change the lower opacity threshold (to remove the most transparent of the visible voxels). The data structure can be incrementally updated accordingly.

3.1 Data Structure Definition

From this point forward, we will define two voxels as "adjacent" if they are in the same scan line and have no visible voxels between them. A structure containing 6 indices is associated with each voxel. It keeps the index of the next visible voxel in each direction along each principal axis. We will call this structure an "adjacency structure".

An adjacency structure has two indices for each axis. These indices direct us along the scan line to an adjacent voxel (the next visible voxel). In order to detect when we have reached the end of a scan line, we cap both ends of the scan line with a "virtual voxel". These virtual voxels do not hold volume data, but have an adjacency structure that points to the first visible voxel in its scan line, as well as to adjacent virtual voxels corresponding to other scan lines. By this mechanism, the data volume is encapsulated in a box of virtual voxels.

These virtual voxels play a key role in traversing the volume efficiently.

There are three levels of virtual voxels: box face, box edge, and box corner. Each one acts as a scan-line cap for the previous level. Figure 1 illustrates the location and hierarchy of these voxels.

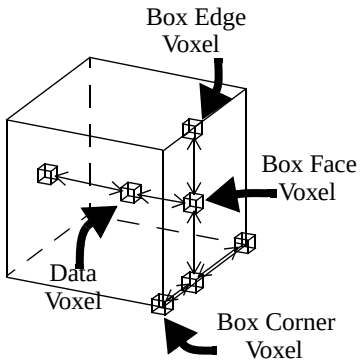

Figure 1. Voxel nomenclature. Every voxel resides in a scan line. Every scan line is capped on both ends by virtual voxels, which come in three varieties: box face, box edge, and box corner.

With this architecture, not only can we skip irrelevant voxels in a scan line by following the indices from one end to the other, but we can skip entire scan lines that have no visible voxels. We do this by also keeping track of the adjacencies between the box face voxels. If a box face voxel is not pointed to, it will be skipped over. Continuing this philosophy, we exclude box edge voxels that delimit empty planes.

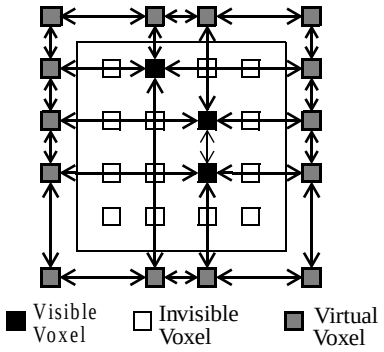

Figure 2. Two-dimensional adjacency structure. This 2D example demonstrates how invisible data voxels are not included in the data structure. The arrows indicate voxel adjacencies.

Figure 2 shows a 2D analog of our data structure. The black squares represent visible data voxels, while white squares are invisible data voxels. The gray squares are the virtual voxels that form a box around the data volume. Both corner voxels and edge voxels are present (box face voxels need a third dimension to exist). The arrows represent voxel adjacencies, indicating that two voxels are pointing to each other. As one follows the adjacencies through the data structure, none of the invisible voxels are visited.

It is interesting to note that the box face voxels represent a binary parallel projection of the visible voxels. The same is true for box edge and box corner voxels for their corresponding projections.

3.2 Building the Data Structure

The structure is initialized by adding the 8 corner voxels of the volume block, each having 3 indices to point to their adjacent corners. From that point on, the structure grows as we add one data voxel at a time. With the addition of a data voxel to the structure, a cascade of virtual voxels is updated. For example, a data voxel resides in three different scan lines, one for each axis. There are six box face voxels that cap these three scan lines, and they must also be part of the data structure. If these box face voxels are not already in the data structure, they are added. The addition of a box face voxel requires the inclusion of the appropriate box edge voxels. This process continues up the hierarchy until it reaches the box corner voxels, which are already included in the data structure. Voxels can also be removed from the data structure by a similar process.

3.3 Data Structure Traversal

The beauty of the adjacency data structure is that it facilitates fast and efficient back-to-front ordering without having to visit any invisible data voxels. The structure is traversed in scan line order. The order in which the axes are traversed is established by evaluating the dot product of the view vector with the unit vectors along each of the three axes. The axis that yields the largest dot product magnitude is closest to colinear with the view vector, and hence defines the back-to-front trajectory. For the purpose of this paper, we will denote that axis as the “slow” axis. Similarly, we will refer to the “medium” axis and the “fast” axis as those that correspond to the middle and smallest dot product magnitudes, respectively. The computer implementation of this ordered traversal translates to nested loops in which the slow axis corresponds to the outermost loop and the fast axis corresponds to the innermost loop.

The algorithm starts at the corner voxel farthest from the eye position. In figure 3, the starting voxel is labelled with the letter A. Recall that the corner voxel

has adjacency indices for each of the three axes. Following the adjacency along the slow axis, we arrive at a box edge voxel, labelled B in the diagram. Notice that voxel B need not be a neighbour of voxel A. Voxel B is included in the data structure because it subtends the first plane that contains at least one visible data voxel. From voxel B, we follow the index that corresponds to the medium axis to arrive at a box face voxel, labelled C. Now we follow voxel C's index along the fast axis to our first data voxel, D.

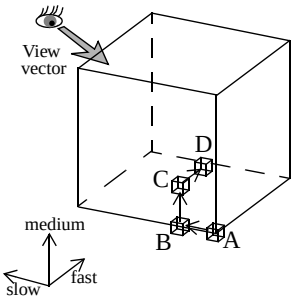

Figure 3. Adjacency volume data structure traversal. Corner voxel A is the farthest from the eye point (the eye point is on the far side of the volume, and to the left).

From voxel D, the scan line is traversed until the box face voxel on the far side is encountered, indicating the end of the scan line. At that point, the traversal returns to voxel C and follows the medium pointer and traverses the next non-empty scan line. Once we have reached the far end of the medium axis (shown as the vertical axis in figure 3), we return to box edge voxel B and simply jump to the next box edge voxel, and repeat the whole process. Eventually, the process will encounter the corner voxel opposite the starting point, signalling the end of the traversal.

4. Results

The adjacency algorithm was implemented in ANSI C++ using the OpenGL and GLUT graphics packages. Wherever possible, the transformations were implemented using the OpenGL library since they are hardware accelerated on many video cards. The splat was made from a 2D Gaussian kernel and was saved as a texture map, allowing us to take advantage of more hardware acceleration [4]. Timing benchmarks were run on a Microsoft Windows 900MHz Pentium III workstation with 512Mb of RAM and an NVidia GeForce2 Ultra graphics card with 64Mb of DDR RAM. The program rendered to a 300x300 pixel window.

The adjacency volume splatting algorithm was compared to three other splatting algorithms, all similarly implemented: traditional splatting, the unordered splatting of Crawfis [5], and octree splatting of Laur and Hanrahan [13]. In a rendering pass of the traditional splatting algorithm, every voxel in the volume is visited, splatting only those voxels that are deemed visible by the chosen transfer function.

In the unordered splatting algorithm, a list of visible voxels is established in a preprocessing step, and that list is traversed in a spatially non-specific order, splatting every voxel. It should be noted that our implementation of the unordered splatting algorithm is not designed to test the speed of Crawfis' method, but rather to use as a speed comparison to gauge the amount of overhead introduced by the adjacency algorithm's depth-sorted traversal. In our unordered splatting implementation, we store many values that are not necessary for Crawfis' splatting (normal vectors, adjacency indices, etc.). We chose to calculate and store these values because they are used in the adjacency splatting method, making for a more direct and meaningful comparison. The unordered splatting algorithm and the adjacency algorithm are not comparable in terms of their output, since the adjacency algorithm creates depth-sorted renderings with shading, while the unordered algorithm is used to render cloud-like volumes.

The splatting methods were benchmarked on five different data sets. The "Jeff" data set is a 256x256x129 (8-bit) T1 weighted MRI scan of the author's head. The "Frog" data set is a 500x470x136 (8-bit) segmented MRI volume of a frog. The "CT Abdomen" data set is a 128x128x142 (8-bit) angiogram CT scan of a human abdomen. The "CT Head" data set is a 128x128x93 (8-bit) CT scan of a human head. Finally, the "HIPIPH" data set is a 64x64x64 (8-bit) volume that shows the ψ function for a high potential iron protein molecule. Figure 4 shows adjacency algorithm renderings of each of the data sets.All the benchmark timings were done with parallel projection rendering. However, in object-based rendering, the occlusion in a perspective projection can be different than that in a parallel projection. The issue of the correct back-to-front (or equivalently, front-to-back) ordering for perspective splatting was published by Swan [32]. They split the back-to-front rendering along the scan plane containing the viewpoint, reversing the order for each side of the plane. This enhancement was easy to implement into the adjacency splatting algorithm to give fast and accurate perspective projection

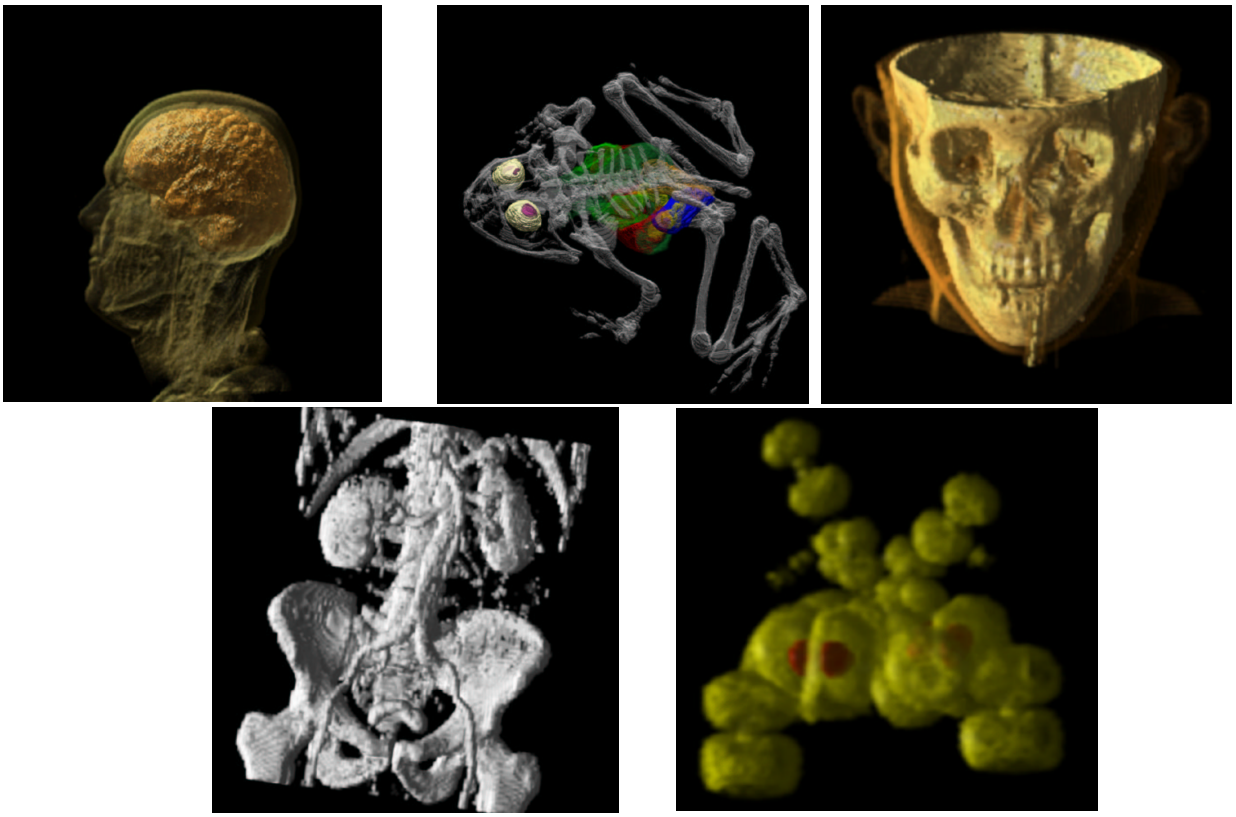

Figure 4. Images produced by the adjacency algorithm. Starting from the top-left and proceeding clockwise, the volumes are "Jeff", "Frog", "CT Head", "HIPIPH" and "CT Abdomen". See table 3 for more information on each of the renderings.

Table 1

	Frame Rate (frames per second)				
Algorithm	Jeff 256x256x129 513,799 splats 6.1% of voxels	Frog 500x470x136 354,894 splats 1.1% of voxels	CT Head 128x128x93 197,828 splats 13.0% of voxels	CT Abdomen 128x128x142 79,914 splats 3.9% of voxels	HIPIP 64x64x64 19,253 splats 7.3% of voxels
Adjacency	1.96	2.85	2.64	8.09	13.95
Unordered	2.99	3.89	2.58	8.07	13.95
Traditional	0.93	0.24	2.48	4.39	12.74
Octree	1.52	-	2.54	7.50	12.42

renderings. The timing difference between the parallel and perspective projection versions of the adjacency algorithm is negligible.

4.1 Rendering Speed

Table 1 gives the benchmark results for the four algorithms on the five data sets. The frame rates are given in units of frames per second, and were established by averaging the frame drawing time over 36 frames (rotating the volume through 360^o in increments of 10^o). The octree method could not be run on the "Frog" data set because the algorithm ran out of memory during preprocessing. The adjacency, unordered and octree algorithms all exhibit similar frame rates for the

three smallest volumes. This phenomenon is likely a result of the rendering speed being limited by the pixel fill rate, since the splat sizes are relatively large. All three accelerated splatting algorithms are faster than the traditional method, particularly for large data sets.

Figure 4 plots the number of selected splats versus rendering time (in seconds per frame) for the adjacency and unordered splatting methods. The graph distinguishes between a displaying run, in which an image is displayed on a computer monitor, and a non-displaying run, which is identical except that the final graphics command that actually draws the splat is not invoked. The non-displaying benchmarks were created to test the traversal time independently of the actual drawing.

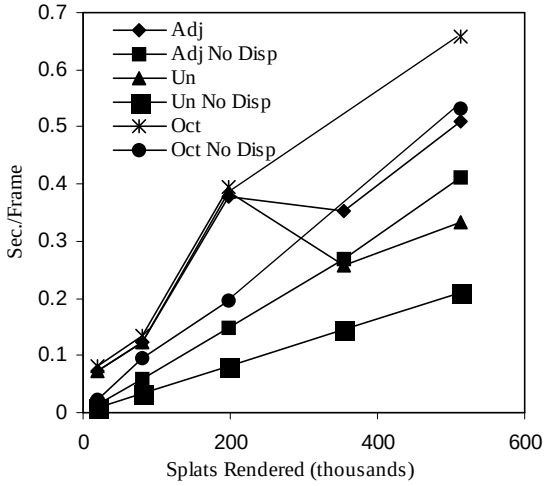

Figure 5. Number of splatted voxels versus frame rendering time in seconds per frame. Both displaying and non-displaying timings are graphed. The non-displaying timings involved all the same processing except for the actual splat drawing.

The frame rendering times for the non-displaying benchmarks are a linear function of the number of splats rendered. Furthermore, if the gradient and colour graphics commands are also removed from the program, the three algorithms perform at roughly the same speed. The separation of the lines in figure 5 is likely caused by a combination of the traversal efficiency and cache coherence.

We did not carry out a formal study of the preprocessing times for the different algorithms, but some basic observations may shed some light on their general performance. The preprocessing step of building the data structure in the adjacency algorithm ranged from 0.61 seconds (for the "HIPIPH" data set) to 9.5 seconds (for the "Jeff" data set), and roughly corresponded to the total number of voxels in the data structure. While the adjacency algorithm preprocessing has to build the data structure, it saves time by only calculating a fraction of the number of gradient vectors and voxel colours. The traditional algorithm does not have to build the data structure, but computes gradient and colour vectors for voxels that will ultimately not be splatted. The preprocessing times required by the two algorithms was comparable. The octree splatting algorithm required somewhat longer preprocessing times.

4.2 Memory Analysis

Our implementation of each algorithm pre-calculates and saves relevant voxel properties. In the traditional splatting algorithm, every voxel requires 17 bytes of memory: 4 bytes for a colour vector (red, greed, blue, alpha), 12 bytes for a 3-float gradient vector, and 1 byte for the voxel intensity value. Thus, if $w \times h \times d$ are the dimensions of the data volume, then the memory required by the traditional method is $m_T = 17whd$.

For the adjacency rendering algorithm, there are two levels of allocation, a small allocation for every voxel in the volume, and a larger allocation for only those voxels that will be in the data structure. First, every voxel in the volume requires 5 bytes of memory: 1 byte for its intensity value, and a 4-byte pointer to the larger allocation that contains additional information. This additional structure is only allocated for those voxels that are included in the adjacency data structure, and requires 28 bytes: 4 bytes for colour, 12 bytes for a gradient vector, and 12 additional bytes in the form of 6 two-byte integers used as indices to adjacent voxels. Recall that the adjacency algorithm requires the volume be encapsulated by virtual voxels. Thus, if $w \times h \times d$ are the dimensions of the data volume, then $(w+2) \times (h+2) \times (d+2)$ are the dimensions of the total voxel allocation, which includes the box of virtual voxels. So, if s is the number of data and virtual voxels included in the data structure, then the memory, m_A, used by this algorithm is,

$$m_A = 5(w+2)(h+2)(d+2) + 28s \ .$$

It is important to note that the adjacency data structure includes virtual voxels, as well as visible data voxels. For example, the data structure that was used to create the rendering of "Jeff" in figure 4 has 513,799 data voxels (as noted in the figure caption), as well as 115,688 virtual voxels. Hence, the total number of voxels in the data structure is 629,487.

This may seem like an increased memory requirement compared to the traditional method. However, the number of voxels that have a non-zero opacity is typi-

cally only a fraction (less than 15% for the data sets examined here) of the total number of voxels in the data set.

There is a more streamlined version of the adjacency data structure that we did not implement. In our memory comparison, we will refer to this new version as implementation B, while calling the original implementation A. Method B discards the original volume data once the data structure is built, so that only voxels in the data structure have memory allocated to them. This strategy does not allow for fast updating of the data structure because the original volume data is no longer in memory. In implementation B, each voxel in the data structure takes 47 bytes of memory: 1 byte for voxel intensity, 4 bytes for colour, 24 bytes in the form of six 4-byte pointers (to adjacent voxels), 3 two-byte integers for voxel position, and a 12-byte gradient vector. The data structure built on this format can take less memory than method A, but affords less flexibility for dynamic updating. We will label the amount of memory used by this method m_B.

Table 2

Algorithm	Memory Required (bytes)		
	Required for every voxel	Required for each of s voxels in the adjacency data structure	Total Memory Requirement
Traditional	$1+k$	0	$(1+k)N^3$
Adjacency A	5	$12+k$	$5(N+2)^3+(12+k)s$
Adjacency B	0	$43+k$	$(31+k)s$

There are many ways to compare the memory requirements for the adjacency algorithms A and B, and the traditional splatting method. Before we do any comparisons, we will generalize the methods by classifying pieces of information as "critical" or "additional". For example, voxel intensity and pointers to other voxels are critical to the operation of both adjacency implementations. These entities are categorized as critical information. Except in adjacency implementation B, neither colour nor the gradient vector needs to be kept in memory since each can be derived from voxel intensity. They are classified as additional information.

Table 2 outlines the memory requirements of the 3 algorithms by distinguishing between critical and additional information, as well as information needed for every voxel, or just for those voxels that are included in the adjacency data structure. In the table, k represents the number of bytes allocated for additional information, as outlined above. For simplicity, we will assume that the volumes have dimensions $N \times N \times N$ (for adjacency algorithms, the effective volume size is $(N+2)^3$ due to the box of virtual voxels surrounding the volume).

Define $\hat{s}$ to be the fraction of the total number of voxels that are included in the adjacency data structure (which includes both data voxels and virtual voxels). Thus, we have

$$\hat{s} = \frac{s}{(N+2)^3} \text{ which implies that } s = \hat{s}(N+2)^3 .$$

Now, we can look at the ratio of the memory required by the two adjacency algorithm implementations.

$$\frac{m_A}{m_B} = \frac{(5+(12+k)\hat{s})(N+2)^3}{(43+k)\hat{s}(N+2)^3} = \frac{5+(12+k)\hat{s}}{(43+k)\hat{s}}$$

Setting this ratio equal to 1, we see that the two algorithms use the same amount of memory when 16.1% of the total number of voxels are included in the adjacency data structure (i.e. when $\hat{s} = 0.161$). For fewer voxels, method B uses less memory than method A.

The size of k also factors into the memory equation. For every one-byte increase in the size of k, the traditional algorithm allocates N^3 bytes of memory, whereas the adjacency algorithms A and B allocate s bytes each. Recall that s is likely much smaller than $(N+2)^3$. Hence, the bigger k is, the more of a memory advantage the adjacency algorithms offer.

Table 3 shows the amount of memory required by the traditional algorithm and each adjacency implementation for each of the 5 data sets. The table gives the number of visible voxels, as well as s, the number of voxels in the adjacency data structure (both visible voxels and virtual voxels), and $\hat{s}$, the fraction of voxels used in the data structure. The memory required by the traditional rendering method does not depend on the number of voxels splatted, so it is used as a baseline for comparison. Memory requirements for the unordered splatting algorithm are omitted here because there is no need for the gradient vector when rendering a non-depth-sorted data set, so memory comparison is irrelevant.

It should be noted that as s increases, the total amount of memory required by the adjacency methods will eventually overtake that of the traditional method. The exact value of s at which the adjacency implementation A overtakes the traditional method can be derived by setting $m_T = m_A$ and solving for s. For each of the 5 data sets, an $\hat{s}$-value of approximately 40%

Table 3

Algorithm	Memory Required (Megabytes) and Percentage of Traditional (in parentheses)				
	Jeff 256x256x129 513,799 splats s = 629,487 $\hat{s}$ = 7.2% of voxels	Frog 500x470x136 354,894 splats s = 600,979 $\hat{s}$ = 1.8% of voxels	CT Head 128x128x93 197,828 splats s = 257,365 $\hat{s}$ = 16.0% of voxels	CT Abdomen 128x128x142 79,914 splats s = 128,255 $\hat{s}$ = 5.3% of voxels	HIPIP 64x64x64 19,253 splats s = 31,372 $\hat{s}$ = 10.9%
Traditional	137.06	518.15	24.70	37.72	4.25
Adjacency A	58.39 (42.6%)	171.97 (33.2%)	14.53 (58.8%)	15.03 (39.8%)	2.21 (52.2%)
Adjacency B	28.22 (20.6%)	26.94 (5.2%)	11.54 (46.7%)	5.75 (15.2%)	1.41 (33.1%)

would make the two methods use the same amount of memory. This figure is much higher than the typical range of 1% to 20% observed in our study, as shown by table 3.

5. Conclusions and Future Extensions

The adjacency data structure rendering algorithm is capable of creating the same images as the traditional splatting algorithm, but with considerably less processing time. The benchmarks clearly demonstrate the advantage of using the adjacency data structure over the standard splatting algorithm, especially for large data sets. The frame rates of the unordered algorithm and the adjacency algorithm are comparable, particularly for medium and small data sets. This finding represents a substantial improvement over the unordered splatting algorithm, since the adjacency algorithm affords depth ordering and directional shading with little overhead. Furthermore, the adjacency data structure is so robust and easy to navigate, that it allows for the proper back-to-front ordering for perspective splatting with a negligible processing cost. The adjacency algorithm even outperforms the octree splatting algorithm.

The specific performance depends somewhat on the distribution of visible voxels throughout the volume. For example, contrast the situation in which 10,000 visible voxels form a tight clump, against the situation in which 10,000 visible voxels are uniformly distributed throughout the volume. The latter case has its pertinent data spread over many scan lines, forcing the algorithm to traverse a greater number of virtual voxels. However, for all the data sets we tested, the speed increases afforded by skipping invisible voxels outweighs the cost of traversing the adjacency data structure.

It was noted above that the three accelerated splatting algorithms all perform about the same for small data sets. The most likely cause of this irregularity is pixel fill rate. However, the non-displaying benchmarks suggest that volume traversal and cache coherence both contribute to rendering slowdown. In both the adjacency and the octree algorithms, voxels are not necessarily processed in a memory-sequential order, and likely cause frequent cache misses. Our implementation of the unordered method does not suffer from this handicap, since the voxels are stored in a sequential array. The adjacency and octree implementations are not optimized for cache coherency. Research into cache coherent optimizations of these splatting algorithms may yield considerable performance gains.

If gradient vectors and voxel colours are pre-computed and stored in memory, the adjacency algorithm takes far less memory than the traditional splatting method for typical data sets. The savings come from the fact that not all the gradient vectors and voxel colours are required to render a scene, and the adjacency method takes advantage of that. Depending on the distribution of visible voxels, the adjacency algorithm required between 30% and 60% of the memory used by the traditional method. However, further memory savings can be realized. Currently, all virtual voxels that are included in the data structure have the full allotment of indices, even though not all of them are used. For example, a box corner voxel only needs 3 adjacency pointers. This approach was taken simply for programming ease. Furthermore, instead of recording the index of the adjacent voxels, one could equivalently record the distance to the adjacent voxel. These steps will be shorter, and may require only one byte. If there is a need for a step larger than 255, a “stepping voxel” can be added with zero opacity. However, this strategy

introduces a trade-off in traversal speed, and makes building an optimal data structure more complex.

Despite the efficient way in which the voxels are visited, the adjacency data structure still renders many voxels that are not seen. For example, voxels that are surrounded by other opaque voxels (such as an inner structure of the head) need not be splatted since they will not be seen from any angle. Voxels can be added to or removed from the data structure with minimal effort. Hence, if an algorithm arises that can effectively determine which voxels are not seen from any angle, those voxels can easily be removed from the structure.

References

[1] R.N.Bracewell, *Two Dimensional Imaging*, Prentice Hall Inc., Englewoods Cliffs, NJ, 1995.

[2] C.Bajaj, *Data Visualization Techniques*, Wiley & Sons, Ltd. Chichester, West Sussex, England, 1999.

[3] B.Cabral, N.Cam, J.Foran, *Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware*, Proc. of Symp. on Volume Vis., pp. 91-97 (Oct. 1994).

[4] R.A. Crawfis, N. Max, *Texture Splats for 3D Scalar and Vector Field Visualization*, Proc. of IEEE Conference on Vis., pp. 261-266 (Oct. 1993).

[5] R.A. Crawfis, *Real-time Slicing of Data Space*, IEEE Vis., pp. 271-278 (Oct. 1996).

[6] R.A. Drebin, L. Carpenter, P. Hanrahan, *Volume Rendering*, Proc. of SIGGRAPH, 22(4), pp.51-58 (Aug. 1988).

[7] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, *Computer Graphics: Principles and Practice, 2nd edition*, Addison-Wesley, 1995.

[8] S.F. Frisken-Gibson, *Using Linked Volumes to Model Object Collisions, Deformation, Cutting, Carving, and Joining*, IEEE Trans. on Vis. and Comp. Graphics, 5 (4), pp. 333-348 (Dec. 1999).

[9] I. Ihm, R. Lee, *On Enhancing the Speed of Splatting with Indexing*, IEEE Vis., pp. 69-76 (Oct. 1995).

[10]T. Itoh, K. Koyamada. *Automatic isosurface propagation using an extrema graph and sorted boundary cell lists*, IEEE Trans. on Vis. and Comp. Graphics, 1 (4), pp. 319-327 (Dec. 1995).

[11]G. Knittel, *The ULTRAVIS System*, 2000 Symp. on Volume Rendering, pp. 71-79 (Oct., 2000).

[12]P. Lacroute, M. Levoy, *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*, Proc. of SIGGRAPH, pp. 451-458 (July 1994).

[13]D.Laur, P.Hanrahan, *Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering*, Proc. of SIGGRAPH, 25(4), pp.285-288, 1994

[14]M.Levoy, *Display of Surfaces From Volume Data*, IEEE Comp. Graph & Appl., 8 (3), pp. 29-37 (May 1988).

[15]L.Lippert, M.H.Gross, *Fast Wavelet Based Volume Rendering by Accumulation of Transparent Texture Maps*, Comp. Graphics Forum, 14(3), Proc. of EUROGRAPHICS 1995, pp. 431-444.

[16]W.E.Lorensen, H.Cline, *Marching Cubes: a High Resolution 3D Surface Reconstruction Algorithm*, Proc. of SIGGRAPH, 21(4), pp. 163-169 (July 1987).

[17]T.Malzbender, *Fourier Volume Rendering*, ACM Trans. on Graphics, 12 (3), pp. 233-250 (July 1993).

[18]X. Mao, *Splatting of Non Rectilinear Volumes Through Stochastic Resampling*, IEEE Trans. on Vis. and Comp. Graphics, 2 (2), pp. 156-170 (June 1996).

[19]N. Max, *Optical Models for Direct Volume Rendering*, IEEE Trans. on Vis. and Comp. Graphics, 1 (2), pp. 97 - 108 (June 1995).

[20]M. Meißner, J. Huang, D. Bartz, K. Mueller, R. Crawfis, *A practical comparison of popular volume rendering algorithms*, 2000 Symp. on Volume Rendering, pp. 81-90 (Oct., 2000).

[21]K. Mueller, T. Möller, and R. Crawfis, *Splatting without the blur*, Proc. Vis. 1999, pp. 363-371, 1999.

[22]K. Mueller, R. Crawfis, *Eliminating Popping Artifacts in Sheet Buffer-Based Splatting*, Proc. Vis. 1998, pp. 239-245, 1998.

[23]K. Mueller, T. Möller, J.E. Swan II, R. Crawfis, N. Shareef, R. Yagel, *Splatting Errors and Antialiasing*, IEEE Trans. on Vis. and Comp. Graphics, 4(2), pp. 178-191 (June 1998).

[24]K. Mueller, R. Yagel, *Fast perspective volume rendering with splatting by using a ray-driven approach*, Proc. Vis. 1996, pp. 65-72, 1996.

[25]H. Pfister, M. Zwicker, J. van Baar, M. Gross. *Surfels: Surface Elements as Rendering Primitives*, Proc. of SIGGRAPH 2000, pp. 335-342 (July 2000).

[26]H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, L. Seiler, *The VolumePro Real-Time Ray-Casting System*, Proc. of SIGGRAPH 1999, pp. 251-260 (August 1999, Los Angeles, California).

[27]C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl, *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization*, in Proc. 2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware.

[28]S. Rusinkiewicz, M. Levoy. *QSplat: a multiresolution point rendering system for large meshes*, Proc. of SIGGRAPH, pp.343-352 (July 2000).

[29]P. Sabella, *A Rendering Algorithm for Visualizing 3D Scalar Fields*, Proc. of SIGGRAPH, 22 (4), pp. 51-58 (August 1988).

[30]L.M. Sobierajski, A.E. Kaufman *Volumetric Ray Tracing*, Symp. on Volume Vis., pp.11-18 (Oct. 1994).

[31]H.W.Shen, C.D.Hansen, Y.Livnat, C.R.Johnson, *Isosurfacing in span space with utmost efficiency (ISSUE)*, IEEE Vis., pp. 287-294 (Oct. 1996).

[32]J.E. Swan II, *Object-Order Rendering of Discrete Objects*. Doctoral Dissertation, Department of Comp. & Information Sci., Ohio State University, May 1997.

[33]T. Totsuka, M. Levoy. *Frequency domain volume renderingy*, Comp. Graphics (Proc. of SIGGRAPH 1993), 27 (4), pp. 271-278 (August 1993).

[34]L. Westover. *Footprint Evaluation for Volume Rendering*, Comp. Graphics (Proc. of SIGGRAPH), 24 (4), pp. 367-376 (August 1990).

[35]R.Yagel, Z.Shi, *Accelerating Volume Animation by Space-Leaping*, Proc. of Vis., pp.62-69 (Oct. 1993).

[36]R. Yagel, A. Kaufman, *Template-Based Volume Viewing*, Comp. Graphics Forum, 11(3), Proc. of EUROGRAPHICS, pp. 153-167 (Sept. 1992).