Differential Volume Rendering: A Fast Volume Visualization Technique for Flow Animation

Han-Wei Shen and Christopher R. Johnson
Department of Computer Science
University of Utah
Salt Lake City, UT 84112.
E-mail: hwshen@cs.utah.edu and crj@cs.utah.edu

Abstract

We present a direct volume rendering algorithm to speed up volume animation for flow visualizations. Data coherency between consecutive simulation time steps is used to avoid casting rays from those pixels retaining color values assigned to the previous image. The algorithm calculates the differential information among a sequence of 3D volumetric simulation data. At each time step the differential information is used to compute the locations of pixels that need updating and a ray-casting method is utilized to produce the updated image. We illustrate the utility and speed of the differential volume rendering algorithm with simulation data from computational bioelectric and fluid dynamics applications. We can achieve considerable disk-space savings and nearly real-time rendering of 3D flows using low-cost, single processor workstations for models which contain hundreds of thousands of data points.*

Introduction

While there is a rich history of numerical techniques for computing the dynamics of wave propagation, only recently have researchers been able to visualize the complex dynamics of large 3D flow simulations [1, 2, 3]. Visualizations of flow dynamics typically involve characterizing relevant features of vector and scalar fields. While visualizing vector fields is particularly important in many applications, it is also important to quantify and visually characterize scalar features of the flow field such as, temperature, voltage, and magnitudes of vector quantities. Direct volume-rendering techniques, effective tools for exploring 3D scalar data, have been proposed as a methodology to visualize scalar features in the flow field [4, 5, 6]. Unlike surface-rendering methods, direct volume-rendering methods can be used to visualize 3D scalar data without converting intermediate geometric primitives. By assigning appropriate colors and opacities to the scalar data, one can render objects semi-transparently to expand the amount of 3D information available at a fixed position. Volume-rendered images can also be superimposed upon surface-oriented icons or textures, thus allowing for simultaneous scalar and vector field composite visualizations.

A wide range of volume rendering techniques have been applied to the problem of flow visualization. Ma and Smith [3] introduced a *virtual smoke* technique to enhance visualization of gaseous fluid flows. This technique allows users to interactively insert a *seed* into an interesting location and only the region immediately surrounding the seed is rendered. Max et al. [5] introduced 3D textures advected by wind flow upon volume rendered climate images to visualize both the scalar and vector fields of the images. Crawfis and Max [7] make use of *textured splats* which utilize volume splatting and 3D texture mapping techniques to reveal scalar and vector information simultaneously. Additionally, Max et al. [8] developed the concept of *flow volumes*, volumetric equivalents of stream lines, to represent additional information about the vector field.

We are motivated to develop a more efficient way to visualize scalar fields by our attempts to visualize simulation data from a model of electrical wave propagation within the complex geometry of the heart and in large scale models of unsteady compressible fluid flow [9, 10]. Because of the regular structures used to characterize the simulation data, we can use direct volume-rendering techniques to visualize the states at each time step. We characterize states within the model by assigning different colors and opacities. By animating the volume-rendered images at each time

*Such as the SGI Indy or comparable workstations step, we can effectively investigate the propagation of waves throughout the volume.

Direct volume-rendering methods employing ray casting algorithms have become the standard methods to visualize 3D scalar data. To characterize the dynamic behavior of the flow field, one generates a series of volume rendered images at different time steps and then records, stores, and animates the sequence. Because direct volume rendering is very time consuming to realize such animations for models of any significant size (i.e. realistic problems in science and engineering), standard volume-rendering techniques are prohibitive for animating hundreds of time steps interactively. Moreover, the disk space required for storing hundreds or thousands of sets of volumetric simulation data could be overwhelming.

The main contribution of this paper is the development of an algorithm which significantly reduces the time to create volume-rendered flow animations of scalar fields. Furthermore, our algorithm reduces the amount of disk space needed for storing volume data. We achieve these reductions by implementing a differential volume-rendering method. The method utilizes data coherency between consecutive time steps of simulation data to accelerate the volume animation and to compress the volume data. The method is independent of specific volume-rendering techniques and can be adapted into a variety of ray casting paradigms which can be used to further accelerate the visualization process [11, 12, 13].

Differential Volume Rendering

From preliminary studies of our wave propagation simulations, we noticed that the only elements which changed values between consecutive time steps, when the time steps were small, were the activated cells and their neighbors. We hypothesized that only a fraction of elements in the volume change from any given time step to the next in simulations of physical flow phenomena. In addition, when a sequence of propagating images is animated, the viewing parameters usually don't change. In our ray casting method, we are able to cast rays along a path corresponding only to changed data elements. Therefore, the pixels in the new image keep the same colors that they had previously unless they correspond to changed data elements. Retaining the color values from the non-changing pixels results in a significant time savings. The differential volume-rendering algorithm thus exploits the temporal coherence between sets of volume data from different time steps in order to speed up volume animation of the 3D flow.

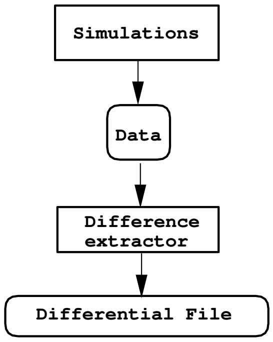

Figure 1: Visualization Pipeline: Static Phase

The differential volume rendering method separates the data generation and data visualization processes. Scientists perform simulations to obtain sequences of data at different time steps. The differential volume-rendering algorithm extracts the differential information which contains the differences between the data files at each consecutive time step. According to the specified viewing direction, the pixel positions where new rays need to be cast can be computed from the differential information and then the ray casting process is invoked to produce the updated image. Because the variation between consecutive time steps is small, the differential information file, which replaces the whole sequence of volume data, can yield tremendous savings in terms of disk space.

Visualization Pipeline

The visualization pipeline of the differential volume-rendering method can be divided into two phases, static and dynamic.

First, data is generated from a simulation which might typically consist of hundreds or thousands of time steps worth of information. Second, the difference extractor is invoked to compare the simulation data of consecutive time steps to obtain the positions of changed data elements between simulation steps. The positions of those changed elements and their corresponding time step values are output into a single *differential file*. Because the differential file contains the state histories of all the data elements through the whole course of the simulation, the only information needed for the rendering process, the volume data at each time step can be then discarded, yielding a considerable savings in terms of disk space. These operations are classified as the *static phase* because they need to be performed only once for a simulation. Figure 1 illustrates the operations in the static phase.

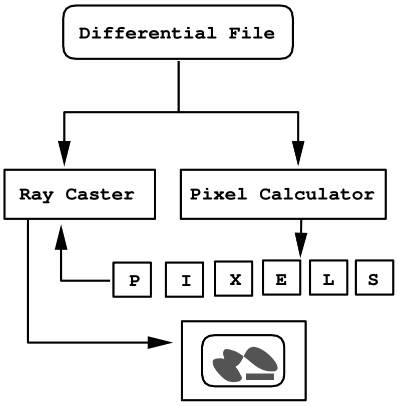

Figure 2: Visualization Pipeline: Dynamic Phase

The differential information obtained over the duration of the simulations contains the 3D positions and values of changed data elements. Those positions are independent of the viewing direction of the rendering. By pre-processing the simulation data and producing the differential file in advance, we can avoid the delay of calculating the differential information while performing ray casting.

At each time step, the positions of changed elements are extracted from the differential file and the pixels where new rays need to be cast can be computed according to the viewing direction and the sampling method. The resultant pixel positions are placed into a ray casting list to which the ray casting process refers before firing new rays to produce the updated image. We classified these operations as the *dynamic phase* because the pixels corresponding to those changed elements are dependent on the viewing direction. The pixel positions remain undetermined until the user specifies the viewing parameters. The operations in the dynamic phase of the pipeline are illustrated in Figure 2 and are outlined algorithmically below.

```
for (each time step t)
{
  for (each changed element[x,y,z])
   {
    calculate the corresponding pixel [u,v];
    update volume[x,y,z];
    store [u,v] into the ray casting list;
   }
  for (each pixel [u,v] in the ray casting list)
   {
     cast a ray from [u,v] into the volume;
     update the image value;
   }
  display the image;
}
```

Pixel Positions Calculation

There are many interpolation and ray sampling methods which can be used in the ray casting algorithm. For interactive rendering rates, but coarse image quality, we can use a discrete ray which avoids the use of interpolation. To obtain higher quality images, one can increase the sampling rate along the ray and use, for example, a trilinear interpolation scheme.

Continuous Rays and Trilinear Interpolation: When using continuous ray sampling along with a trilinear interpolation scheme, we compute the value of any point in the volume by interpolating the values of the data element's encompassing eight vertices. If any of these eight data elements changes its value, the interpolated value of the point inside that cube needs to be re-computed. We define the *interpolation space* of a data element as the space where the values of all the points located inside are influenced by that data element while the interpolation is performed. In the case of a 3D regularly structured grid, a vertex is shared by its eight adjacent cubes. Therefore, for any data element, its interpolation space is the volume of its eight adjacent cubes.

To locate those pixels which will cast a ray through a data element's interpolation space, we project the eight vertices of that cubic volume back to the image plane according to the viewing parameters (parallel or perspective). The pixels bound by the projected region then need to be updated.

Discrete Rays and Zero-Order Interpolation: Discrete rays can be obtained by extending the digital differential analyzer (DDA) scan converting line algorithm [14] into three dimensions. In a zero-order interpolation scheme, at each forwarding step only the nearest voxel is sampled. No actual interpolation operation is performed. Suppose the 3D line equations are $y = m1 * x + b$ and $z = m2 * x + c$, and both $m1$ and $m2$ are less than 1 and greater than -1. According to the discrete ray algorithm, the ray in position x would be increased by 1 at each forwarding step. After i times of forwarding, the ray's discrete position is $[i, \text{Round}(m1 * i + b), \text{Round}(m2 * i + c)]$, and the voxel located at that position is sampled.

To calculate pixels corresponding to the changed voxel element, one projects the voxel position back to the image plane according to the viewing direction. Most of the time the projected position won't be located exactly at the image plane's grid point. The four surrounding pixels of that projected point thus need to be selected to cast new rays. We cast rays from the four surrounding pixels instead of choosing only the nearest pixel to assure that the changed voxel will be hit and sampled.

Template Based Rays: When parallel projection is used, all rays fired from the image plane have the same slope and thus the same incremental form of forwarding path. Therefore, one can calculate the incremental form once and store it as a template [12, 4]. During the ray casting procedure, instead of computing the ray's new position at each time step, which increases computational complexity, we can use ray templates. Suppose we have the forwarding templates in u, v and w directions which have the forms of ray_template[i].u, ray_template[i].v, and ray_template[i].w. The ray_template[i].u stores, for instance, the distance the ray should march at the next step in u direction after the ray has taken i steps of forwarding. Suppose the current position of a ray is [u, v, w], then the next position [u', v', w'] of the ray is calculated as:

```
u' = u + ray_template[i].u;
v' = v + ray_template[i].v;
w' = w + ray_template[i].w;
```

To guarantee a complete and uniform tessellation of the volume by 26-connected rays' sampling, we cast rays from the *base-plane*, which is parallel to one of the volume faces. After having obtained the projected image on the base-plane, one performs a 2D mapping from the base-plane to the image plane to obtain the final image.

The templates described above give us the information about how far a ray will forward at each step in each axis direction. From those templates, one computes the displacement a ray has forwarded in each direction after a particular time step. Inversely, we can also obtain the number of forwarding steps corresponding to a particular displacement. This information gives us two more templates in each axis direction which can be described as two functions, displacement_to_step() and step_to_displacement(). Given a displacement from a volume point to the base plane, the displacement_to_step() function will return the steps taken by a ray to reach that point. Given a forwarding step number for a ray, the step_to_displacement() will return the displacement from the base plane.

With these extra templates, the pixel position calculation becomes simple. Given a point in volume space, we can find the distance from that point to the base plane. By using the displacement_to_step template, we are able to calculate the steps taken by a ray from the base plane to that point. Thus we can look up the step_to_displacement templates to get the displacement in each direction, and the starting position of the ray can be calculated. By adopting the template-based ray casting method and utilizing precomputed displacement and step templates, we can accurately and efficiently calculate the pixel positions.

Acceleration of Ray Casting

Coordinate Buffer: Yagel and Shi [11] proposed a *coordinate buffer* to store the first- and last-hit voxel positions for every pixel on the image plane allowing one to rapidly skip empty space for subsequent renderings. The differential volume rendering algorithm further utilizes this idea for volume animations to speed up the ray casting process.

If the viewing direction is fixed during the rendering of a sequence of volume data, each ray from the image plane follows exactly the same forwarding path during the whole course of renderings with different sets of volume data. If only a small fraction of data elements change between consecutive simulation steps, then most of the information stored in the coordinate buffer from the previous rendering can be retained.

Initially, the sampling range for each pixel constitutes the full depth of the volume. After the first rendering, the effective sampling range for each pixel can be obtained and stored. At each subsequent time step, when the differential volume rendering algorithm extracts a changed data element from the differential file to compute the corresponding pixels, the 3D position of that data element can be compared with those computed pixels' effective sampling ranges stored in the coordinate buffer to decide if those ranges, i.e., the first- and last-hit voxel positions, need to be changed. If the changed element is located outside the specified ranges, then the stored information needs updating. The new sampling ranges can then be used to skip empty space while performing the ray castings and thus the rendering can be further accelerated.

Sample Caching: During the ray casting, the local lighting calculation, which includes the trilinear interpolation and shading computation, constitutes the most expensive part of the process. Ma et al. [13] proposed a sample caching technique which stores the interpolated data value and local shading information at each sample point along a ray. This allows users to interactively change mapping parameters, such as color and opacity, so that one only need to composite the cached information to update the image.

To apply the sample caching technique, initially the sampled value and lighting information at each sample point of a ray is saved. For a changed data element in the subsequent time step, only the sample points inside the surrounding regions of that new data element need to be resampled. The new sampled values are then used to update the sample cache. The correct position to insert or replace the new sampled result can be found from the corresponding pixel and the distance between the new data element and the image plane. After resampling the changed data elements and updating the sample cache, one needs to composite the new sample cache of the changed pixels to update the image. By combining this technique with our differential volume rendering algorithm, we can reduce the complexity of the sampling process and thus further accelerate the ray casting process.

Results and Discussion

We have implemented our differential volume rendering on two sets of simulation data, one from a biomedical application and the other from a computational fluid dynamics application. All the comparisons below consist of the performances of our ray casting software with and without the differential rendering capability. The performance measurements were evaluated on a single 100 MHz MIPS R4000 processor. Note that the focus should be on the relative performances of the ray casting algorithms with and without differential capability instead of on any one technique's efficiency.

In the electrical wave propagation simulation, we attempted to simulate the electrical impulse conduction in the heart using an anatomically accurate cellular automation model. Simulation data consisted of the state histories of all the elements in the model over the duration of the simulation. Scientists were interested in following the activation wavefront and studying phenomena that facilitate, promote and/or terminate abnormal propagation. Appropriate colors and opacities were assigned to the different states when volume rendering was performed. Data is computed at each time step within $128 \times 128 \times 128$ cubic elements. To test the differential rendering algorithms, we computed data at 100 time steps. For display we used a 256×256 image plane. Figure 3 depicts the volume rendered images of the propagation of electrical activity within the heart.

In the computational fluid dynamics simulation [10], numerical results were computed by software which uses the MacCormack method to solve the three dimensional unsteady compressible Navier-Stokes equations. The results simulated a laminar flow entering a rectangular region. The region had a small inlet at one end and a fully wide open outlet at the other end. Data is computed at $64 \times 64 \times 64$ regular grid points and displayed in a 128×128 image plane. Figure 4 shows the volume rendered images of the laminar flow propagation.

Time Step	Changed Elements	Regular Ray Casting	Differential Ray Casting
0	100%	100%	100%
20	3.505%	100%	3.33%
40	4.531%	100%	4.93%
60	1.369%	100%	1.58%
80	0.379%	100%	0.36%

Table 1: Percentage of rays being cast at selected time steps for both regular and differential ray casting methods: electrical wave propagation simulation.

On the analysis of our simulation data, we noticed that a maximum of 4.77% of the elements in the electrical wave propagation simulation, and a maximum of 2.57% in the laminar flow simulation, changed states between time steps. Therefore, we could exploit the data coherency.

The disk space used by the 100 time steps of $128 \times 128 \times 128$ volume data in the electrical wave propagation simulation was 2.10 M Bytes $\times$ 100 = 210 MB. Using the differential volume rendering algorithm, which only requires 2.08 MB for a differential file and 2.1 MB for the first time step of volume, we were able to save more than 95% in storage costs. In the laminar flow simulation, the disk space requirement was 4.09 MB for a differential file and 0.262 MB for the first $64 \times 64 \times 64$ volume. The regular rendering algorithm for 130 time steps needed 0.262 MB $\times$ 130 = 34.06 MB. This amounts to a savings of 88%.

Table 1 lists the average percentages of rays being cast both with and without use of the differential capability at different time steps of electrical wave propagation data. The algorithm without the differential capability always shot 100% of rays at any time step. That is, the alogirithm requires every pixel to fire a ray to sample the volume data. In the differential ray casting method, after the first rendering which needs to cast 100% of rays to obtain the initial image, the subsequent 99 renderings cast rays only when necessary. This significantly reduced the number of rays cast. Table 2 shows the results for rendering the laminar flow propagation data.

Table 3 and Table 4 list the average rendering time required for both simulations for a single image at se-

Time Step	Changed Elements	Regular Ray Casting	Differential Casting
0	100%	100%	100%
20	1.82%	100%	3.48%
40	2.45%	100%	4.65%
60	1.46%	100%	2.79%
80	0.20%	100%	1.44%
100	0.04%	100%	0.32%
120	0.005%	100%	0.15%

Table 2: Percentage of rays being cast at selected time steps for both regular and differential ray casting methods: laminar flow simulation.

Time Step	Changed Elements	Regular Ray Casting	Diff. Ray Casting	
			Time	Pct.
0	100%	13.399	13.399	100%
10	1.148%	13.398	0.446	3.47%
20	3.505%	13.389	0.733	5.47%
30	4.435%	13.347	0.880	6.59%
40	4.531%	13.315	0.924	6.94%
50	3.619%	13.377	0.807	6.03%
60	1.369%	13.390	0.422	3.15%
70	0.607%	13.401	0.291	2.17%
80	0.379%	13.397	0.265	1.97%
90	0.089%	13.398	0.234	1.75%

Table 3: Rendering time (in seconds) at selected time steps using both the regular and differential ray casting algorithms: electrical propagation simulation data

Time Step	Changed Elements	Regular Ray Casting	Diff. Ray Casting	
			Time	Pct.
0	100%	2.245	2.245	100%
20	1.82%	2.245	0.23	10.24%
40	2.45%	2.248	0.28	16.27%
60	1.46%	2.239	0.19	8.48%
80	0.20%	2.241	0.08	3.56%
100	0.04%	2.241	0.04	1.78%
120	0.005%	2.246	0.02	0.89%

Table 4: Rendering time(in seconds) at selected time steps using both the regular and the differential ray casting algorithms: laminar flow simulation data

Time Step	Changed Elements	Differential Ray Casting		
		Pixel Calculation	Ray Casting	Total
1	0%	0	2.372	2.372
3	10%	0.226	0.173	0.439
5	20%	0.516	0.282	0.798
7	30%	0.765	0.413	1.178
9	40%	1.019	0.529	1.548
11	50%	1.265	0.641	1.906
13	60%	1.515	0.771	2.286
15	70%	1.766	0.884	2.650
17	80%	2.017	1.013	3.030
19	90%	2.266	1.116	3.382

Table 5: Rendering time (in seconds) for different amount of changed elements in a $64 \times 64 \times 64$ volume

lected time steps using our ray casting software both with and without adopting the differential capability.

The rendering time for the first image (time step 0) in the differential volume rendering was the same as that in the regular ray casting algorithm. However, for subsequent renderings, there was a significant reduction in rendering time. For the electrical wave propagation data, the average time to render 100 images without adopting the differential method was approximately $13.39 \times 100 = 1339$ seconds. Differential volume rendering achieved the same amount of rendering and same image quality in only $13.5 + 52 = 65.5$ seconds. For the laminar flow simulation, the differential volume rendering method took 17.98 seconds to render 130 time steps. The rendering algotithm without the differential algorithm needed approximately $2.45 \times 130 = 318.5$ seconds. Our algorithm for rendering both sets of data achieved a savings of more than 90%.

To understand the limitations and robustness of the differential volume rendering algorithm, we used a $64 \times 64 \times 64$ volume, 128×128 image plane and increased the number of changed elements at each time step. Table 5 lists the rendering times of our algorithm for different percentages of changed data elements. The experiment was designed such that the flow started propagating from one corner of the volume and eventually occupied 95% of the whole volume in 20 time steps. The number of changed elements increased as the propagation proceeded.

Except for the first rendering time, which was the same as that in the regular ray casting, the differential ray casting time at each time step was shorter than the regular ray casting time. Although our ray casting time remained smaller than the regular ray casting time, the pixel calculation time increased with the increasing number of changed elements. When the number of changed elements increased, the time of pixel calculation increased to the point of diminishing returns. From our experiments, when the percentage of changed elements was over 50%, the performance of the differential volume rendering method became worse than the regular ray casting method.

Although our algorithm has limitations when the number of changed elements exceeds 50%, for most flow visualization applications the number of changed elements during the propagation at each time step constitutes only a small fraction of the whole volume. Furthermore, these changed elements tend to cluster together. Therefore, differential volume rendering represents an attractive technique for scalar field flow visualization.

Summary

We have presented a differential volume rendering algorithm which exploits data coherency between consecutive time steps of data to achieve fast volume animation. This method can potentially save tremendous amounts of disk space and CPU time over existing methods. The algorithm begins by preprocessing the simulation data over time and extracting the differential information between sequential time steps. At each time step, the pixel locations from which new rays need to be cast can be calculated and the ray casting process is invoked to update the image. Our algorithm has been successfully applied to both biomedical and computation fluid dynamics applications. We are currently working on a parallel version of the algorithm.

Acknowledgments

This work was supported in part by the Whitaker Foundation. The authors would like to thank P. Gharpure for the heart wave propagation data, and K. Ma for his CFD simulation software. We would also like to thank Professors R. Yagel, J. Painter and K. Coles for their helpful comments and suggestions. Furthermore, we appreciate access to facilities which are part of the NSF STC for Computer Graphics and Scientific Visualization.

References

[1] J.L. Helman and L. Hesselink. Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications*, 11(3):36–46, 1991.

[2] A. Globus, C. Levit, and T. Lasinski. A tool for visualizing the topology of three dimensional vector fields. In *Proc. of Vis. '91*, pages 33–40. IEEE CS Press, 1991.

[3] K.-L. Ma and Smith P.J. Virtual smoke: An interactive 3d flow visualization technique. In *Proc. of Vis. '92*, pages 46–53. IEEE CS Press, 1992.

[4] A. Kaufman. *Volume Visualization.* IEEE CS Press, Los Alamitos, CA, 1990.

[5] N. Max, R. Crawfis, and D. Williams. Visualizing wind velocities by advecting cloud textures. In *Proc. of Vis. '92*, pages 171–178. IEEE CS Press, 1992.

[6] P.G. Swann and S.K. Semwal. Volume rendering of flow-visualization point data. In *Proc. of Vis. '91*, pages 25–32. IEEE CS Press, 1991.

[7] R. Crawfis and N. Max. Texture splats for 3d scalar and vector field visualization. In *Proc. of Vis. '93*, pages 261–265. IEEE CS Press, 1993.

[8] N. Max, Becker B., and R. Crawfis. Flow volume for interactive vector field visualization. In *Proc. of Vis. '93*, pages 19–23. IEEE CS Press, 1993.

[9] P. Ghapure and C.R. Johnson. A 3d cellular automata model of the heart. In *Proc. of 15th Annual IEEE EMBS Int. Conf.* IEEE Press, 1993.

[10] K.-L. Ma and K. Sikorski. A distributed algorithm for the three-dimensional compressible navier-stokes equations. *Transputer Res. and App.*, 4, 1990.

[11] R Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *Proc. of Vis. '93*, pages 62–69. IEEE CS Press, Oct. 1993.

[12] R Yagel and A. Kaufman. Template-based volume viewing. In *Proceedings of EUROGRAPHICS '92*, pages 153–157. Blackwell, Cambridge, England, Sept. 1992.

[13] K. L. Ma, M. F. Cohen, and J. S. Painter. Volume seeds: A volume exploration technique. *J. of Vis. and Comp. Animation*, 2:135–140, 1991.

[14] J. Foley and A. van Dam. *Computer Graphics: Principles and Practices.* Addison-Wesley, 1990.