An Efficient Method for Volume Rendering using Perspective Projection

Kevin L. Novins, François X. Sillion and Donald P. Greenberg
Program of Computer Graphics
Cornell University, Ithaca, NY 14850.

Abstract

Use of perspective projection adds important perceptual cues for image comprehension. However, it has not been widely used in volume rendering because of the lack of efficient computational algorithms and concern over the nonuniform sampling rate imposed by perspective ray divergence. This paper introduces two new techniques to help make perspective projection more feasible in rendering volume data. First, a method is presented for efficient slice-by-slice processing of volume data, allowing high resolution data sets by eliminating typical memory constraints. Second, an adaptive "ray splitting" approach is described which ensures that the entire volume is sampled with user-specified limits. Additionally, we present results using distributed ray tracing to achieve depth of field effects.

CR Categories and Subject Descriptors: I.3.0 [Computer Graphics]: General; I.3.3 [Computer Graphics]: Picture/Image Generation; J.3 [Computer Applications]: Life and Medical Sciences
Additional Keywords and Phrases: volume rendering, scientific visualization, ray tracing, perspective projection, depth of field

1 Introduction

In recent years, volume rendering has emerged as a powerful technique for visualizing scientific data. The majority of images produced in this manner have employed orthographic projections. While quite effective in many circumstances, orthographic views have, in a sense, held scientists at an infinite distance from their data. Orthographic projections also often result in depth ambiguities and reversals of apparent depth [Image 1]. Since our visual system is tuned to comprehend the perspective views projected onto our retinas, scene geometry and depth are more readily apparent in simulated perspective views. In addition, data flythroughs and lens effects become available to aid in the comprehension of spatial relationships within the data when perspective is used.

To date, two major stumbling blocks have prevented the widespread use of perspective projection. These are the lack of algorithms which are both space-efficient and time-efficient, and the nonuniform sampling inherent in perspective ray divergence. In this paper, we present new techniques for creating perspective images from volume data which address both these problems. In addition, we present our results in using lens effects to highlight a portion of the dataset while maintaining a sense of the overall three dimensional structure.

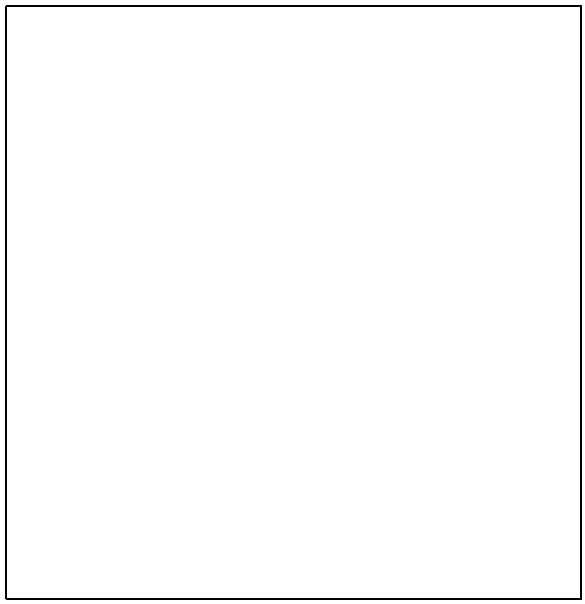

Image 1: A comparison of a perspective (top) and an orthographic (bottom) view of two equal-sized spheres connected with a cylinder. In the orthographic view, lack of perspective depth cueing causes size and orientation ambiguity.

2 Previous Work

A variety of techniques for *volume rendering* (rendering three dimensional grids of volume densities or scalar fields without the use of intermediate representations) have been introduced recently [4, 5, 2, 12]. The volume rendering engine described here follows most closely after the image space mapping technique of [5]. Voxel opacity and color arrays are computed independently via classifications and shading operations. Rays are then case into these arrays and opacities and colors are computed for equally spaced ample points along the ray using trilinear interpolation. A final pixel value is calculated by compositing the opacities and colors encountered at the sample points.

Despite an emphasis on the use of orthographic projection, several researchers have explored the use of perspective in volume rendering. However, to date none have described efficient, single pass slice-by-slice processing techniques. Most ray casting implementations have assumed random access to the volume dataset [9, 10, 8, 11]. If the volume data has n voxels on a side, this approach requires $O(n^3)$ storage space, which quickly outstrips RAM capacity for high resolution arrays. Random access to large virtual memory spaces causes an unacceptable amount of disk paging. In [2] and [3], a perspective view is realized by casting parallel rays into an appropriately transformed copy of the volume grid, however the transformation step is quite expensive. The cell-by-cell processing method of [10] assumes that the data can be accessed slice-at-a-time along any of the three coordinate axes. This presents a difficulty because volume data is usually stored so that slice data may be efficiently accessed on only one axis.

3 Slice-Based Perspective Projection

We present an algorithm for slice-based perspective projection in two parts. First, a scheme to allow ray tracing on a slice-by-slice basis is introduced. Next, we describe a slice processing order which requires that each slice of the dataset be loaded into memory only once.

3.1 Slice by Slice Ray Tracing

A typical ray tracer has an outer loop which iterates once for each ray. Each ray is traced until it terminates, yielding a final pixel value. Processing then concentrates on the next ray. This scheme is extremely inefficient with regard to memory access because each ray must take samples from across the entire dataset.

To increase efficiency the algorithm is inverted, so that iti is driven by the data currently in memory. Because we rely on trilinear interpolation at sample points, at least two slices of data are required in memory at one time. We call each pair of slices a *slab*. The system maintains a queue of rays waiting for each slab so that they may compute their next sample values. This data structure is called a *slab queue*. During initialization, each ray is traced to its first sample point in the volume array. The slab index for this sample point is calculated, and the ray is paced on the appropriate queue. During processing, whenever a slab is loaded into memory, it activates each ray in its queue in turn. The ray processes its sample points until its next sample lies in another slab, at which time the ray is placed on the appropriate queue [Figure 1] [1]

In order to maintain the ability to stop and restart the tracing of a ray, a data structure must be maintained for each ray which contains all information pertinent to its processing. A pointer to this data structure is stored on the slab queue when a ray is queued. The current implementation stores the ray's current sample point, the difference vector between sample points, and a color and opacity for each ray. Additionally, the current sample number and maximum sample number within the volume are stored so that the exit point of the ray can be detected quickly. This amounts to 36 bytes of information per ray. Thus, the ray data structures and one slab of opacity and color values require $36m^2 + 8n^2$ bytes of storage, where m is the screen resolution and n is the data resolution. This compares to $8n^3$ bytes of storage when the entire dataset is stored in memory. If we assume screen resolution equal to data resolution, the method presented here is more memory efficient for $n > 6$. If space is at a premimum, the number of bytes per ray can be reduced by eliminating storage of the current sample point, which can be derived from the eyepoint, sample number, and difference vector.

[1] For ease of comprehension, this figure and all other figures in this paper are presented in two dimensions.

```
/* Initialization */
For each ray {
        Trace ray to first sample point in volume grid
        Find slab which contains this sample point, and
                place ray on the appropriate queue
}


/* Processing */
While rays remain in slab queue {
        Choose a slab to load into memory [Described Below]
        For each ray on queue for current slab {
                do {
                        Compute sample value
                        Compute location of next sample point
                } While sample point is still in slab
                If sample point is still inside the volume boundary
                        Place ray on queue for sample point's slab
        }
}
```

Algorithm 1: Slice by slice ray tracing.

The slab queues are maintained as linked lists through a pointer field in the ray data structure. The ray queuing and dequeueing operations require only two pointer reassignments each, since objects are added and removed from the front of the queue. Thus, the overhead for maintaining the queues is negligible.

3.2 Slab Processing Order

Because perspective rays diverge from a single eyepoint, a slab processing order which requires that each slab be loaded into memory only once is easily defined. We derive a processing order for front-to-back processing of ray sample points. The reverse order is optimal for back-to-front processing. Assume without loss of generality that the data is accessed as slices perpendicular to the z axis. Let E_2 be the z-coordinate of the eyepoint. Regardless of view direction, all rays emanating from the eyepoint must fall in to one of three categories. First, the ray may stay in the $z = E_z$ plane, thus requiring access to only one slab. Second, it may diverge in the $+z$ direction from the eyepoint. In this case it requires access to the slab containing the $z = E_z$ plane and to the slabs with successively larger z values, in order. Similarly in the third case, if the ray diverges in the $-z$ direction from the eyepoint, it requires access to slabs starting with the $z = E_z$ plane and continuing with slabs with successively smaller z values. Thus the optimal slab processing order is as indicated in Figure 2. Combining the slice-by-slice processing technique with the slice processing order, we obtain an algorithm for ray-tracing volume datasets using a perspective projection which requires only $O(m^2)$ storage, makes a single pass through the dataset, and requires only a negligible amount of computation time over that required by algorithms requiring $O(n^3)$ storage. Adaptive refinement may be achieved by displaying the image after each slab is processed. The effectiveness of the intermediate results depends on the orientation of the slabs relative to the image plane.

4 Sampling Issues

Loss of object-space resolution toward the back of the scene being viewed is inherent in the perspective projection. This effect is fa-

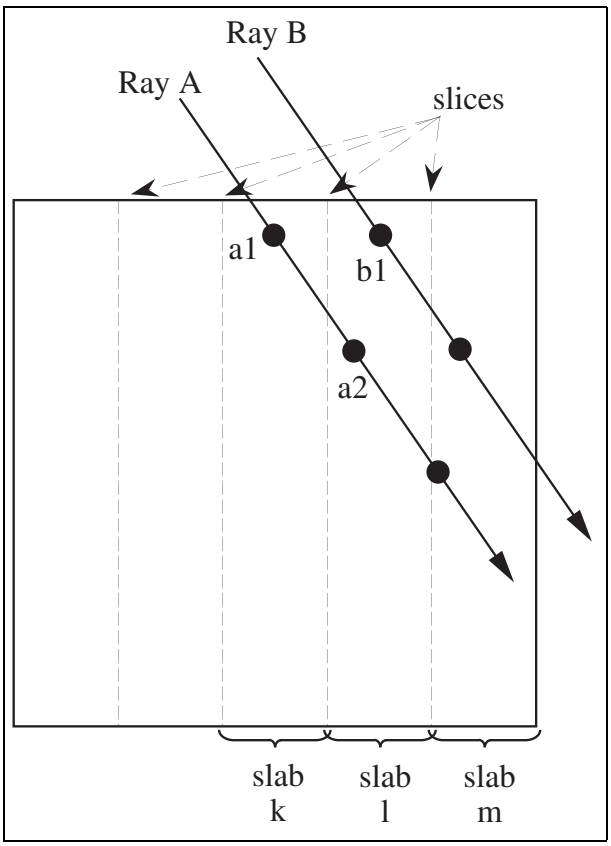

Figure 1: During initialization, ray A is placed on the queue for slab k, which contains its first sample point, $a1$. Similarly, ray B is placed on the queue for slab 1. During processing, when slab k is loaded into memory, ray A comes off the queue and computes $a1$. The ray is then placed on the queue for slab 1. When slab 1 is loaded, rays A and B are activated in turn to compute $a2$ and $b1$.

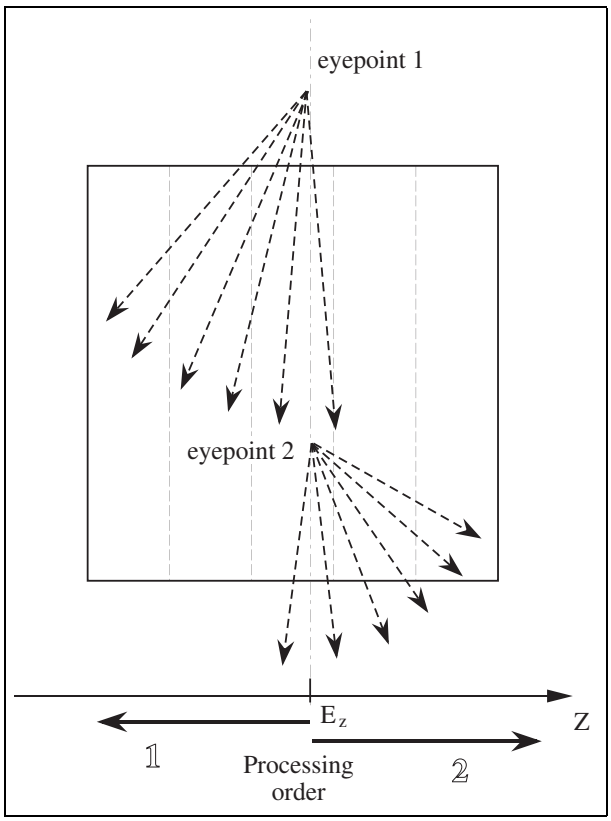

Figure 2: No matter where the eyepoint is located or what the view direction is chosen, rays encounter slices with z values less than that of the eyepoint in decreasing order and slices with greater z values in increasing order.

miliar in conventional rendering and photography as well as in our own visual system. If increased resolution is necessary for objects near the back of the dataset, the user may move his eyepoint closer to the object of interest, rotate his eyepoint to look at the back of the dataset, or simply increase image resolution.

Of greater concern is the undersampling of the volume data as perspective rays diverge. This can result in severe aliasing, which is visually unappealing, and more seriously, the erroneous omission of small objects from the image entirely [Figure 3]. For example, consider the case of a 56-degree field of view looking directly onto a face of the volume bounding box. One ray is hot per pixel through a screen equal in resolution to a slice of the dataset. In this case the first slice of the dataset is sampled at one ray per voxel, but the last slice is sampled a one ray per two voxels in the worst case. Assuming that it is desirable to hold the image size constant, one could naïvely eliminate undersampling by firing enough uniformly distributed rays per pixel so that the back slice is sampled at the desired rate [Figure 4]. This simple oversampling will result in blurring in the back of the region rendered, but every voxel is guaranteed to contribute to the final image. Unfortunately, this approach wastes computation by oversampling the parts of the dataset close to the eyepoint.

An alternative approach to maintaining adequate sampling rates with perspective projection would be to use multiresolution filterings of the data as in the 3D-mipmap algorithms [13, 6]. In these schema, instead of firing additional rays to obtain the average value across a larger region, samples are taken from successively blurred versions of the volume data. This has the positive effect of reducing the number of samples taken during ray tracing. In particular, it reduces computation effort in distant areas which do not contribute much to the final image. However, it adds a level of interpolation between the raw data and the ray sampling process. It also adds some complexity to the ray tracer which must pass through several sets of slabs and vary the sampling rate along the ray direction as it moves to different resolution filterings.

We have adopted an adaptive supersampling approach whereby a ray is split into four divergent rays at the point just before the volumetric sampling rate would drop below a user-specified threshold. Later, the ray fragments are composited to yield a final pixel value [Figure 5]. In the current implementation, all ray fragments are averaged uniformly during composition, however any filter function could be used in its place. As in [5], the sampling rate along the ray axis is maintained by taking samples at regular intervals along the ray. The sampling rate in the remaining two dimensions is maintained by considering the projected area of the screen pixel, which grows with distance from the eyepoint. At the point where the projected area becomes larger than the square of the linear sampling rate, the ray is split. This process is continued recursively until the ray exits the volume boundary.

This adaptive supersampling scheme works well in conjunction with the slice-by-slice processing algorithm presented above. The split points (and thus each ray fragment's start point and length) may be computed during initialization by considering the solid an-

```
For i = E_z to maximum slab index step +1 {
        If slab i's queue is not empty {
                Load slab i and process its rays
        }
}

For i = E_z − 1 to minimum slab index step -1 {
        If slab i's queue is not empty {
                Load slab i and process its rays
        }
}
```

Algorithm 2: Slab processing order.

gle subtended by each screen pixel. Thus, all rays may be queued during initialization and the slice processing order remains the same. Each ray holds a child pointer, so that if its accumulated opacity approaches unity, it may halt the processing of its child rays by removing them from the slab queue.

Using this method, the ray tracer automatically computes the number of child rays necessary to maintain a given sampling rate in a manner that is independent of screen resolution. Through ray splitting, no computation effort is wasted by oversampling the region close to the eyepoint. While supersampling is not always necessary for good image quality, this technique can be implemented quite easily as an option to any ray tracer, thus providing a guarantee that all voxels are taken into account when such precision is absolutely necessary.

5 Lens Effects

If the user defines an area of interest within the volume data, one may improve image comprehension by focusing the virtual camera on that region. The result is better depth perception because objects are blurred as distance from the focal region increases. The blurred remnants of distant objects help to maintain global perspective.

If we use a perspective projection, a minor modification to the ray tracer allows the computation of images with depth of field [7, 1]. Rays are distributed over a finite aperture to blur the image outside the focal region. Because this effectively involves the jittering of the eyepoint, the slice processing order must be modified to take into account the minimum and maximum z extents of the eyepoint. This necessitates making two passes through the slices within this extent [Figure 6].

6 Future Work

The slice processing order presented here is optimized so that each slice of memory is loaded only once. One could imagine alternative slab processing orders which would provide more useful intermediate images for adaptive refinement. One technique would be to give each ray an importance weight, and to always process the slab with the most weight in its queue. A ray could be assigned a high weight if, for example, it is known to pass through a region of interest.

The slice-by-slice processing technique could be modified to take advantage of shared memory coarse-grained parallelism by distributing the rays on a given slab queue to several processors. If memory is not shared, all processors must load the current slab into memory, and thus parallelism should only be used if the slab queue is longer than a certain threshold.

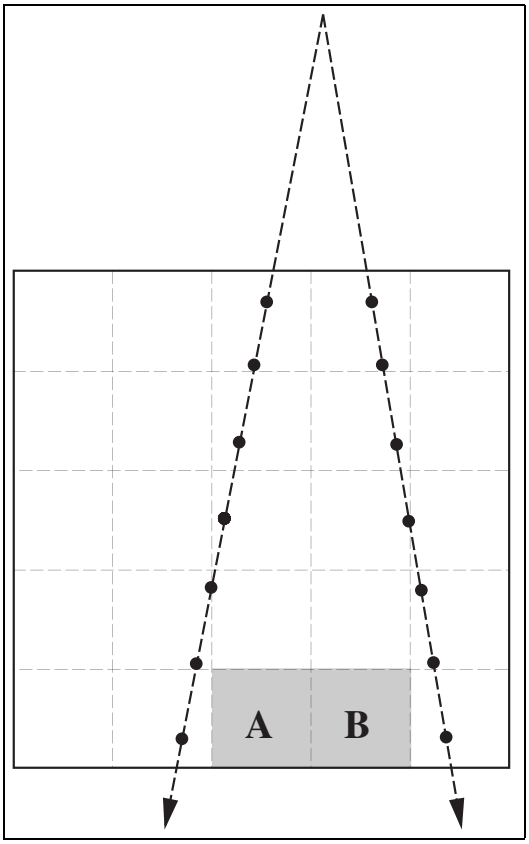

Figure 3: Divergence of perspective rays causes nonuniform sampling of the volume data. In this case, although each voxel in the first row is sampled, voxels A and B and not sampled at all. This results in aliasing and the omission of small details toward the back of the image.

The ray tracing algorithms assumes that the data is stored as slices along one axis of the data. This algorithm should be extended to allow efficient processing of oct-trees, another common data structure used in volume rendering.

7 Results

Images 2 through 8 show 256×256 renderings of a $119 \times 112 \times 73$ CT study of a human pelvis. The viewpoint is on the left side and slightly toward the back of the body. Image 2 shows a perspective view with one ray cast per screen pixel. It tool 356 seconds to render on a Hewlett-Packard 835 workstation. Just over 2 seconds were spent enqueuing and dequeueing rays as described in section 3. These operations could be made substantially faster if they were implemented as macros rather than as procedure calls. Image 3 shows an orthographic view from the same view angle and screen resolution as Image 2. Note that in the orthographic view, the size of the femurs (leg bones) does not vary with distance, so the orientation of the pelvis is difficult to discern. Image 4 uses uniform supersampling to ensure that at least two samples are taken per voxel. Approximately 12 million samples were taken inside the volume array. Image 5 uses adaptive supersampling to ensure the same minimum sampling rate while taking only about 6 million samples inside the volume array. Three images with different aperture sizes are presented in Images 6-8. The images are focused on the base of the spine using a simulated normal lens at $f/2, f/5.6$ and $f/11$.

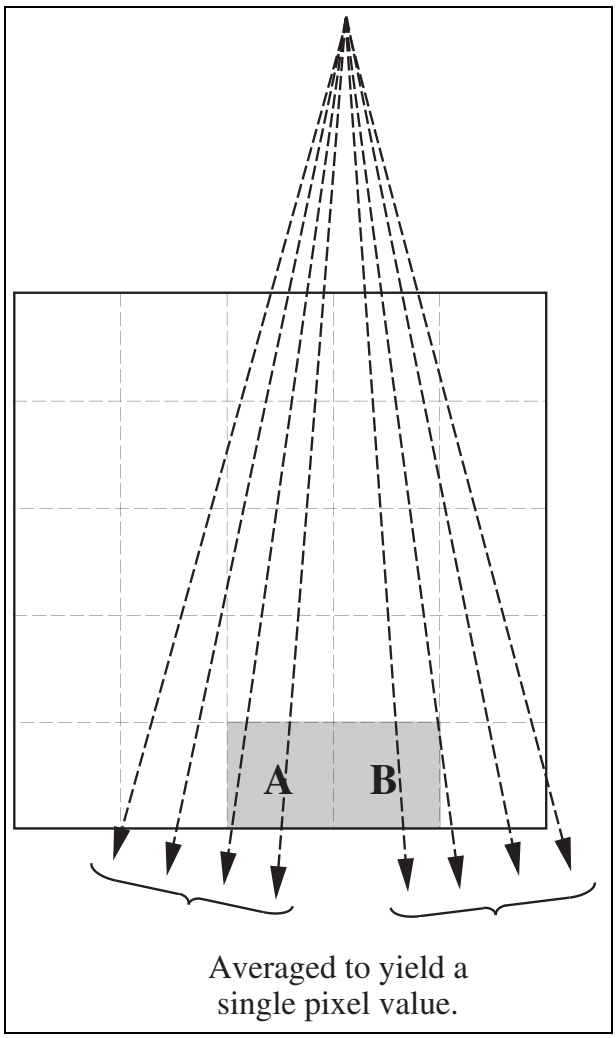

Figure 4: A simple solution to the problem illustrated in Figure 3 is to use pixel supersampling. Enough rays are shot through each pixel so that the most distant voxels are sampled adequately. This results in wasted computation as voxels close to the eye are oversampled.

8 Conclusion

An efficient ray tracing technique for volume rendering with perspective projection has been presented. By using perspective projection, familiar distortion with depth provides a more readily comprehensible image over orthographic projection. In addition, greater freedom in placing the eyepoint and the ability to focus the virtual camera allows a scientist to get "closer" to the data being rendered. By using slice-by-slice processing and volume coherence, the algorithm runs efficiently even when the full dataset cannot be stored in memory, thus eliminating a typical constraint. This factor will become increasingly important as new technology scanning equipment will make very high resolution scientific data available. In addition, we have presented an adaptive supersampling technique which ensures that the 3D sampling rate is never too high or too low. Lastly, the technique can be easily extended to provide depth of field effects.

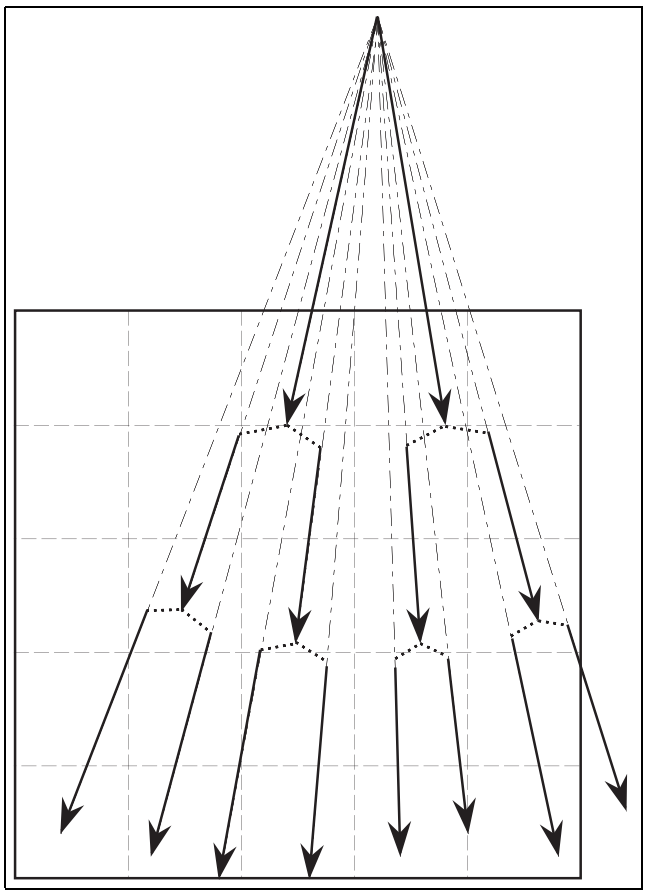

Figure 5: Using adaptive supersampling, each ray is split when divergence causes sampling below a threshold value. The ray fragments are composited to yield a pixel value. Using this method, the three dimensional sampling rate can be maintained within a user-specified tolerance.

9 Acknowledgements

This work was supported by NSF grans "Interfactive Computer Graphics Input and DIsplay Techniques" (CCR-8617880) and "Visualization for Supercomputing: A Graphics WOrkstation Approach" (ASC-8715478). The authors gratefully acknowledge the generoius equipment grant from Hewlett Packard Corporation on whose workstations the research was conducted. All images were displayed on an HP835 Turbo SRX workstation. The CT data was provided by the Department of Radiation Oncology at the North Carolina Memorial Hospital. Thanks to Daniel Kartch for help in reviewing this manuscript, Emil Ghinger for the photography, and Leonard Wanger for modelling and rendering Image 1.

References

[1] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[2] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.

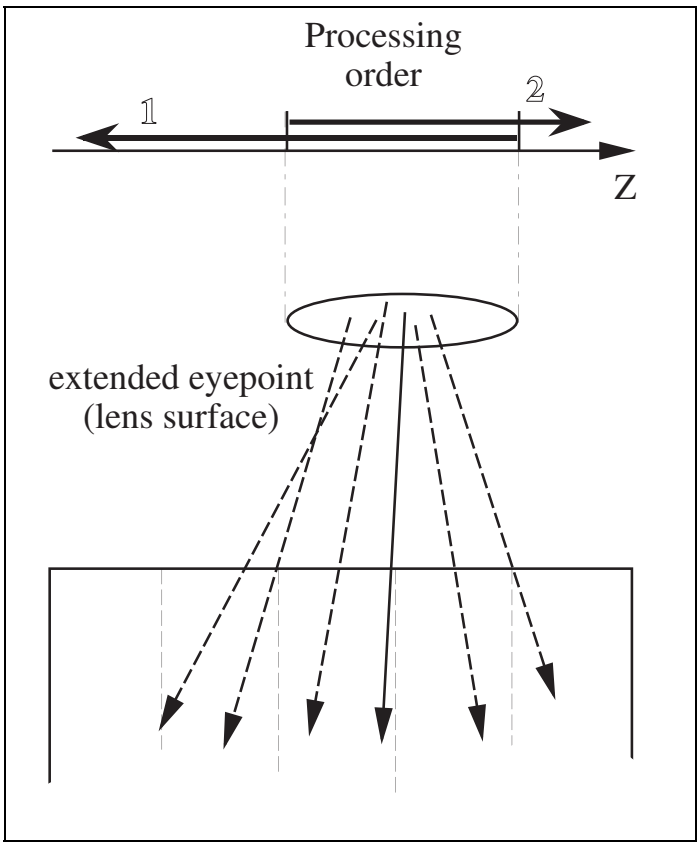

Figure 6: When jittering samples over a lens surface, the procesing order must be modified to account for the shifting ray origins. Slabs between the minimum and maximum z extents of the lens must be processed twice.

[3] Karl Heinz Höhne, Michael Bomans, Andreas Pommert, Martin Riemer, Carsten Schiers, Ulf Tiede, and Gunnar Wiebecke. 3D visualization of tomographic data using the generalized voxel model. In *Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization*, pages 51–57, May 1989.

[4] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(3):165–173, July 1984.

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

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

[7] M. Potmesil and I. Chakravarty. Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics*, 1(2):85–108, April 1982.

[8] Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, 22(4):51–57, August 1988.

[9] Daniel S. Schlusselberg, Wade K. Smith, and Donald J. Woodward. Three-dimensional display of medical image volumes. In *Proccedings of NCGA '86*, pages 114–123. National Computer Graphics Association, March 1986.

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

[11] Robert E. Webber. Ray tracing voxel data via biquadratic local surface interpolation. *The Visual Computer*, 6(1):8–15, 1990.

[12] Lee Westover. Interactive volume rendering. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*, May 1989.

[13] L. Williams. Pyramial parametrics. *Computer Graphics*, 17(3):1–11, July 1983.

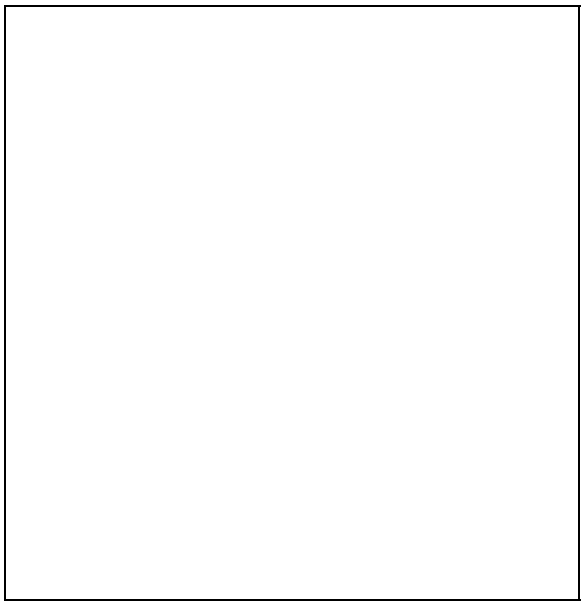

Image 2: A 256×256 perspective view of a $192 \times 112 \times 73$ CT study of a human pelvis.

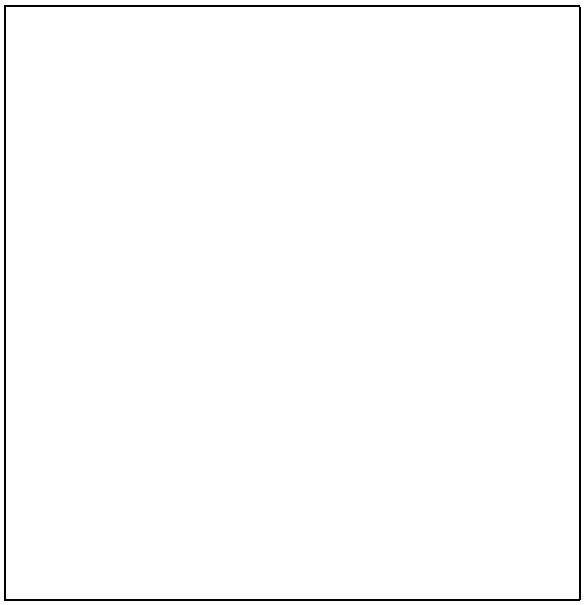

Image 3: An orthographic view of the dataset presented in Image 2. Noth that the orientation of the pelvis is difficult to discern.

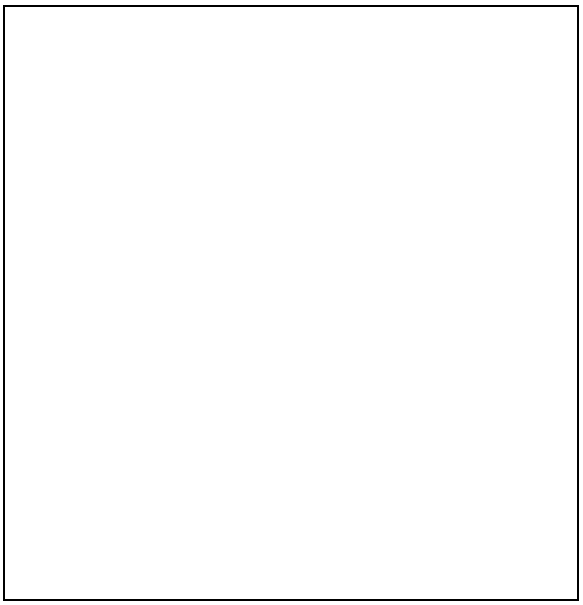

Image 4: Uniformly supersampled image to ensure that at least 2 samples are taken per voxel.

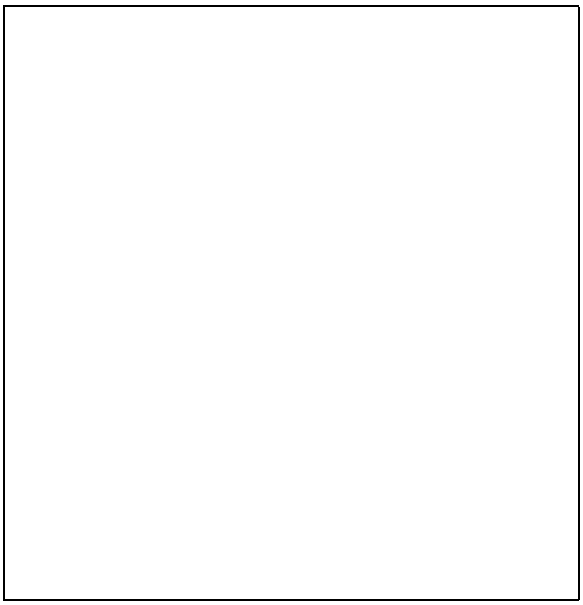

Image 5: Adaptively supersampled image to ensure that at least 2 samples are taken per voxel.

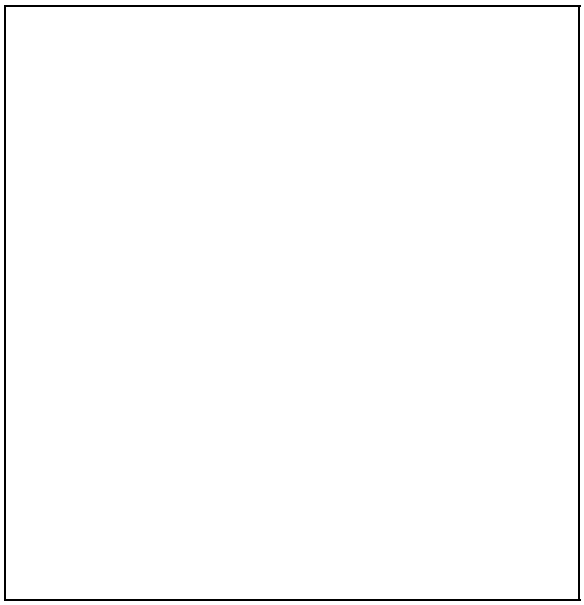

Image 6: Image focused on the base of the spine using a virtual camera set to $f/2$.

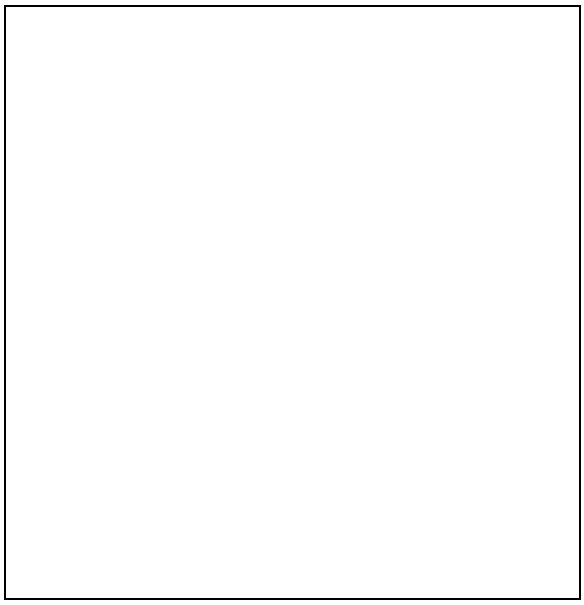

Image 7: Image focused on the base of the spine using a virtual camera set to $f/5.6$.

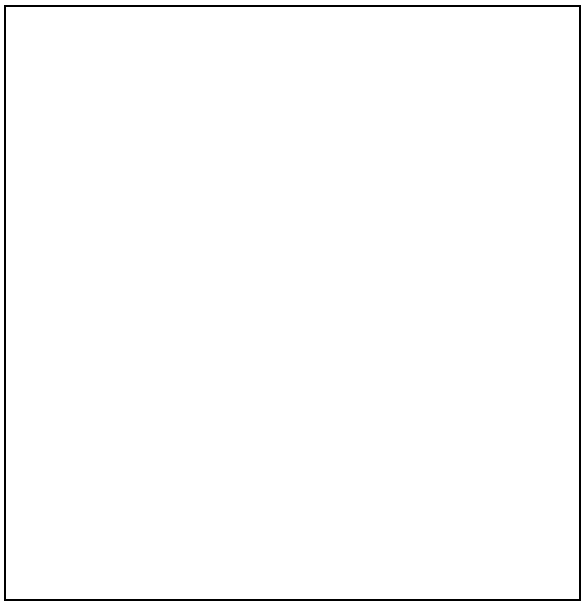

Image 8: Image focused on the base of the spine using a virtual camera set to $f/11$.