Direct Volume Rendering: A 3D Plotting Technique for Scientific Data

Steven P. Callahan Jason H. Callahan Carlos E. Scheidegger Cláudio T. Silva

October 22, 2007

Abstract

Direct volume rendering is an efficient method for plotting three dimensional scientific data. However, the technique is not as frequently used as it could be. In this article, we summarize direct volume rendering and discuss the barriers that need to be overcome to take advantage of this powerful technique.

1 Introduction

The use of plotting techniques for the comprehension of scalar functions is ubiquitous in science and engineering. An effective plot uses features such as first and second derivatives to convey critical and inflection points, which help portray the overall behavior of functions around a region of interest. As the dimensionality of the scalar field increases, plotting becomes harder. For 2D scalar functions, many of us rely on more complex plotting functionality of the type available in certain scientific packages, *e.g.,* VTK and matplotlib [6, 10].

Without going into details, most general 2D plotting functionality available is based on creating contour plots or density plots of the data. In either case, the function is sampled in some way, which turns the problem into a discrete one. In the two-dimensional case, contour plots generate a set of closed curves called *level sets*. Density plots show the two-dimensional function directly by mapping the scalar values through a set of colors. Yet another commonly used technique for plotting 2D scalar fields is to "lift" the function to one dimension higher, that is, draw $(x, y, f(x, y))$. Figure 1 displays contour and density plots for a volume created from the implicit function $f(x,y,z) = x^2 + y^2 + z^2 - x^4 - y^4 - z^4$.

For 3D scalar fields this gets considerably more complicated, and most packages only support 3D contour plots, but not density plots. The visualization of 3D density plots is often called *direct volume rendering*, a term that is used to indicate that no intermediate representations are computed. Instead, pictures are directly determined from the function $f(x,y,z)$, as shown in Figure 1. Volume visualization as a discipline started in the early 1980s, mostly from needs from the medical community to be able to handle 3D data being generated by CT and MRI scanners. That work has grown into one of the major research problems in the scientific visualization community [5].

The primary advantage of direct volume rendering is its flexibility. It is possible to use volume rendering to obtain an initial overall view of the data, and by changing transfer functions (which are direct analogs of color maps), we can incrementally focus on particular features of our data. In the past, direct volume rendering was too slow and cumbersome to be widely used as a plotting technique, but this has not been

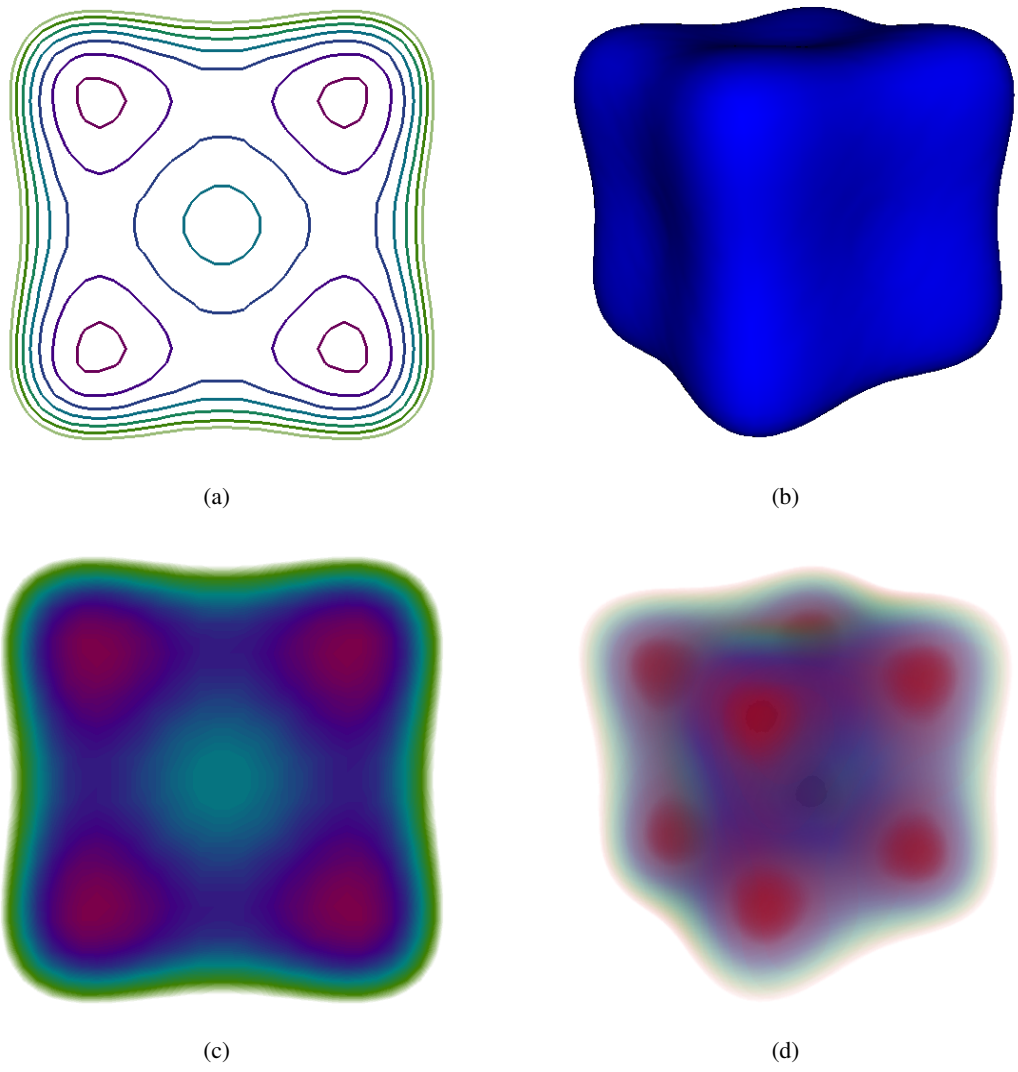

Figure 1: Plotting methods for a volume created from an implicit function. Indirect methods create intermediate geometry before rendering as shown with 2D contours (a) and a 3D isosurface (b). Direct methods render the data directly either as a density plot in 2D (c) or a volume rendering in 3D (d).

the case for several years now. Many improvements, and the wide availability of hardware and software platforms that support volume rendering, make it a very attractive visualization technique, and one that we feel is somewhat underused in the scientific community. The main point of this article is to show the overall simplicity of the technique, its power, and how to best employ existing hardware and software solutions. In this column, we are only covering the basics: direct volume rendering of regular grid data using simple specification techniques. In the future, we hope to cover the most commonly applicable versions of this technique.

2 A Volume Rendering Primer

In rendering volumetric data directly, the data is considered a participating medium that is composed of semi-transparent material that can emit, transmit, and absorb light, thereby allowing one to "see through" (or see inside) the data. By changing the optical properties of the material, different lighting effects can be achieved.

The typical optical model used for volume rendering in scientific visualization is to consider the volume as a medium that both emits and absorbs light, like a cloud. In a physically-based model the light would also scatter, but because the effect does not necessarily make the visualization any clearer, it is generally ignored to make the algorithms more tractable. For a ray of light passing through the volume, the intensity I of the light can be computed using the standard volume rendering integral [7]:

$$I(D) = I_0 e^{-\int_0^D \rho(t)Adt} + \int_0^D C(s)\rho(s)Ae^{-\int_s^D \rho(t)Adt}ds \tag{1}$$

for position $s = 0$ at the edge of the volume and $s = D$ at the eye, particles of area A and density per unit ρ, and emmisive glow C per unit projected area. Because volume rendering is performed incrementally, a discretized form of the equation is commonly used in practice. An approximation to the equation is derived using a Riemann sum, which divides the integral into n equal segments of size Δx:

$$I(D) \approx I_0 \prod_{i=1}^{n} t_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} t_j \tag{2}$$

where

$$t_i = e^{-\rho(i\Delta x)A\Delta x}, \tag{3}$$

$$g_i = C(i\Delta x)\rho(i\Delta x)A. \tag{4}$$

These equations require prior steps for the computation of the current step through the volume. Thus, the integration is performed by sampling the volume incrementally, in order.

Direct volume rendering algorithms consists of three major components: sampling, classification, and compositing. *Sampling* deals with selecting the piecewise steps that are taken through the volume, *classification* is the process of computing a color and opacity for each step using the volume rendering integral, and *compositing* is how these classified steps are blended together to form an image.

2.1 Sampling

A structured volume can be represented as a simple 3D array of scalar values that implicitly defines a grid. Eight neighboring values in the volume define the basic volume element, a *voxel*. Since there are a discrete number of voxels within the grid, the volume integration is performed in a piece-wise manner by sampling incrementally through the volume. Finding the value at an arbitrary sample within a voxel can be easily performed using trilinear interpolation from the neighboring values. The specific manner in which the volume is sampled is dependent on the volume rendering algorithm, and is discussed in Section 3.

One obvious choice for the position of the samples throughout the volume are on the faces that define the voxel boundaries. However, sampling theory tells us that one sample per voxel will not be sufficient. Thus, in practice, multiple samples are taken inside a voxel as well. The frequency of sampling can be adapted depending on the homogeneity of the volume or on the user's preference for speed of interaction over quality of results.

2.2 Classification

Classification for volume rendering is the assignment of color and opacity for a discrete step, defined by two samples, through the volume. Color and opacity are assigned to a scalar within the volume through a user specified mapping called a *transfer function*. The contribution of one step through the volume can thus be computed with the two samples and the distance between them using the volume rendering integral. Additional lighting effects can be obtained for a sample in the volume by attenuating the intensity with a standard lighting model, as with surface rendering. Unlike surface rendering, however, the surface normal at the sample is not defined by geometry, it is defined by differential properties of the scalar field—in this case, the gradient.

The bottleneck for volume rendering performed in this manner is the computation of the integral for each classification step. To address this issue, *pre-integration* of the volume rendering integral replaces the expensive integral computations with a simple table lookup [4]. In practice, the pre-integration is stored in a 3D table that can be indexed using trilinear interpolation using the value at the front scalar, the value at the back scalar, and the distance between the samples. With recent hardware advances, this 3D table can even be stored as a texture which can be efficiently accessed during rendering.

2.3 Compositing

After a sample has been classified, the last step before moving to the next sample is to blend it with the previous samples using alpha compositing [9]. Just as rearranging plates of different colored glass will change the color of objects seen through the plates, the order in which the data is composited will change the results of the volume rendering. Thus, the samples are generally traversed either back-to-front or front-to-back through the volume. The standard compositing algorithm for back-to-front traversal are computed as a function of RGB color $\mathbf{c}$ and opacity α:

$$\mathbf{c}_i = \mathbf{c}_i\alpha_i + \mathbf{c}_{i+1}(1-\alpha_i) \tag{5}$$

or similarly for front-to-back compositing

$$\mathbf{c}_i = \mathbf{c}_{i-1} + \mathbf{c}_i\alpha_i(1-\alpha_{i-1}) \tag{6}$$

$$\alpha_i = \alpha_{i-1} + \alpha_i(1-\alpha_{i-1}) \tag{7}$$

for the steps before $(i-1)$ or after $(i+1)$ the current step (i).

In practice, front-to-back compositing is used most frequently because it facilitates acceleration techniques such as *early-ray termination*, which prevents compositing after a threshold opacity has been reached (*e.g.,* 95% opaque). This method avoids unnecessary computation by skipping regions of the volume that are obscured in the current view.

3 GPU or CPU? Taking Advantage of the Latest Hardware

When it comes to actually implementing volume rendering, there are many possible algorithms. As we will see, two techniques in particular are straightforward to implement and offer significant computational advantages. The first technique is called *ray casting*, and it appropriate for CPUs and recent multicore architectures. The second technique, known as *texture slicing*, takes advantage of the special purpose hardware present in recent graphics processing units (GPUs).

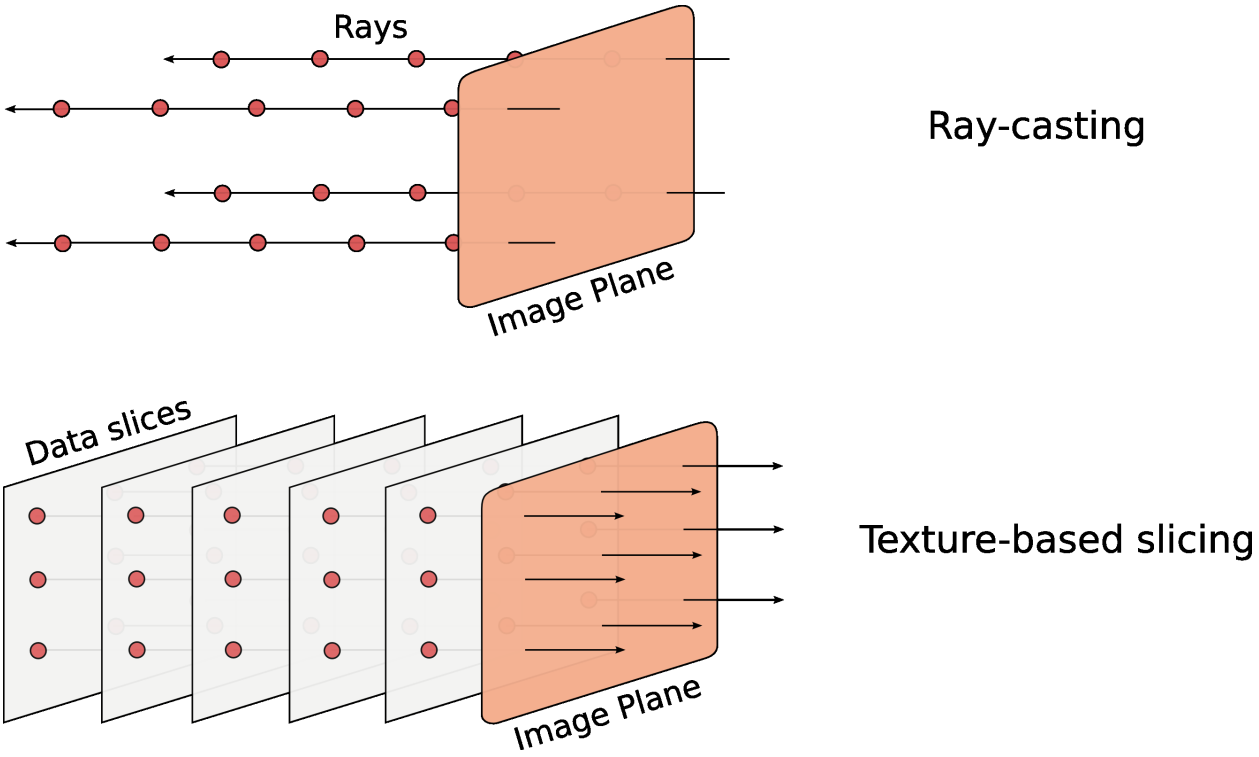

Figure 2: Volume rendering using ray casting vs texture slicing. On CPUs, it is typically faster to cast rays through data, computing these independently. On GPUs, independent slabs of the data (or *slices*) are rendered using special-purpose hardware, making volume rendering extremely fast.

Ray casting [2] is an algorithm that performs a direct geometrical interpretation of Equation 2. The Riemann sum approximation becomes a set of colinear line segments through the volume. These *rays* are cast from the image plane through the dataset, accumulating color and opacity according to the given transfer function.

```
# ray-casting
R = all_rays_in_screen()
for ray in R:
   result = 0
   for step in number_of_steps_through_ray:
       result = composite_step(step, ray, result)
   set_value(ray, result)
```

Ray casting is a very natural implementation for volume rendering that also happens to be computationally desirable. In particular, ray casting is embarassingly parallel. There are no dependencies between different rays in an image. Each pixel in the image plane corresponds to a different ray, and so parallel architectures can be used very effectively to speed up ray casting algorithms. The recent shift towards multi-core architectures makes it a very appealing algorithm for a CPU implementation.

Notice that the only data dependency in the algorithm is that when compositing step n, all the steps from 1 to $n-1$ need to have been composited already. Across rays, however, there is no dependency. By changing the order of the computation, we arrive at a different scheme that is usually referred to as *texture slicing* [1]. While raycasting generates one pixel at a time marching the entire ray before moving to the next ray, slice-based techniques generate *all the pixels* simultaneously by marching all the rays through the volume one step at a time.

```
# slice-based volume rendering
R = all_rays_in_screen()
for step in number_of_steps_through_ray:
  for ray in R:
    current = get_value(ray)
    new_value = composite_step(step, ray, current)
    set_value(ray, new_value)
```

This reordering looks harmless, but it actually makes volume rendering trivial to implement in graphics processors. Graphics processors have become faster at a much faster pace than general purpose CPUs, so algorithms that exploit GPUs tend to perform extremely well [3]. In GPUs, the computation of each slice through the data becomes a single call to an API that is implemented directly in hardware over parallel logic units, leading to an extremely fast implementation. In addition, GPUs transparently rearrange the data layout to improve cache coherency. Finally, the trilinear interpolation and alpha compositing are natively implemented in hardware. Because of these factors, volume rendering can now be performed at interactive rates for essentially any structured grid that fits in the main memory of a graphics card.

4 Finding a Needle in a Haystack: Feature Finding for Volume Rendering

One of the drawbacks to rendering the complete volume is that it may result in information overload. Features of interest within the volume can easily become obscured by regions of little interest. One way to remove superfluous regions is with clipping planes that are inserted into the volume. These planes cull away the parts of the volume on one whole side of the plane. Though efficient, clipping planes are not powerful enough to isolate general homogenous regions within the volume. For this, transfer functions are used.

A transfer function is a simple mapping from scalar values to color and opacity, or more formally, it maps $\Re \rightarrow \Re^4$ (*i.e.,* $s \rightarrow (r,g,b,a)$). A transfer function is generally represented as a lookup table that is accessed using the scalar values and uses bilinear interpolation to represent a continous range with a finite number of entries. Thus, scalar ranges that are deemed important can be given a higher opacity and scalar ranges of little interest can be removed by specifying them as fully transparent. As an example, Figure 3 shows a plot of the transfer function used for the volume rendering in Figure 1. Notice, values below 180 are left transparent to simplify the function and remove unwanted regions of the volume.

Specifying transfer functions can be difficult and the topic continues to be an area of research in the visualization community [8]. Though techniques have been introduced to assist in the specification, the process of feature finding is still very manual. Prior knowledge of the data being visualized can be of great help. For instance, Computed Tomography (CT) medical scans provide densities of a scanned object, and for human tissue, these densities are well classified. Thus, finding regions of interest may only require highlighting the scalar ranges that correspond to those regions. In general, the goal of the visualization should drive the transfer function specification. If the goal is to find boundaries between materials in the

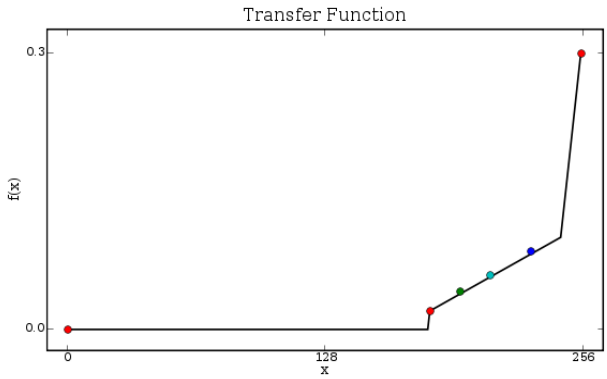

Figure 3: A simple representation of the transfer function $f(x)$ for the implicit dataset shown in Figure 1. The black lines show the opacity function and the points show changes in color.

volume, a tool that uses differential properties of the scalar field (*e.g.,* gradient magnitude) may be important. However, if the goal is to find one or more homogenous regions, a tool that shows the 1D histogram of the scalar values may be of more use.

In general, because volume rendering is often used as an exploratory task, the process of specifying transfer functions requires a lot of user intervention. Though this process can be time consuming, it can also be of great benefit because it can give insight to the data that cannot be gained using other 3D plotting techniques.

Acknowledgments

This work was partially supported by the National Science Foundation, the Department of Energy, an IBM Faculty Award, an IBM Ph.D. Fellowship, and a University of Utah Seed Grant.

References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Workshop on Volume Visualization*, pages 91–98, 1994.

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

[3] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. AK Peters, 2006.

[4] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the EG/SIGGRAPH Workshop on Graphics hardware*, pages 9–16, 2001.

[5] C. R. Johnson and C. Hansen, editors. *The Visualization Handbook*. Academic Press, 2004.

[6] Matplotlib: a python 2d plotting library. http://matplotlib.sourceforge.net.

[7] N. L. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[8] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. S. Avila, K. Martin, R. Machiraju, and J. Lee. The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21(3):16–22, 2001.

[9] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, 1984.

[10] VTK: The visualization toolkit. http://www.vtk.org.

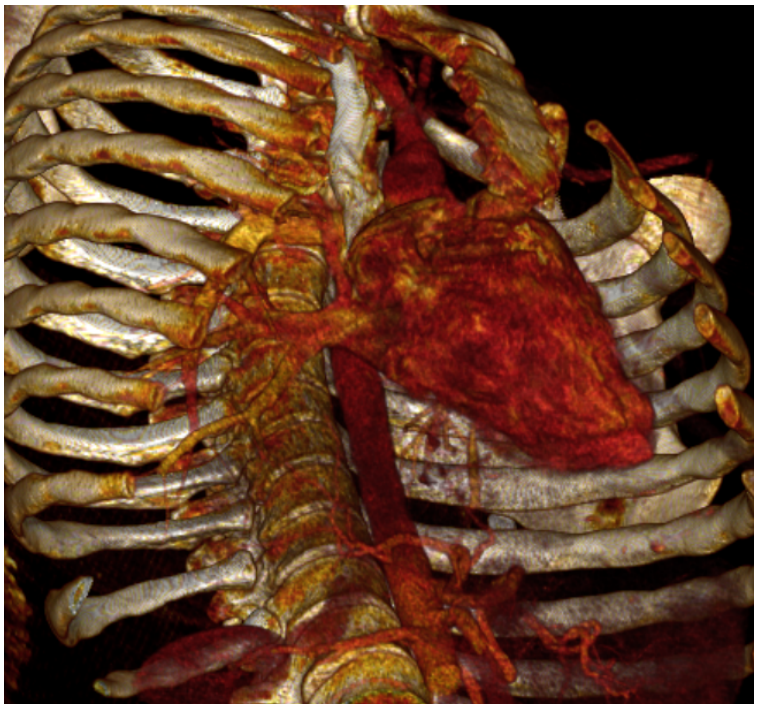

Figure 4: Direct volume rendering example of a CT dataset of a chest.

Sidebar: A Step-By-Step Example

There are many tools available for direct volume rendering of structured data. Here, we walk through a step-by-step example of creating a visualization from scratch using the Visualization Toolkit [10] with the VisTrails system. Since interacting with the data itself is can be an efficient learning tool, we recommend the reader to explore it directly. This example, as well as all other visualizations shown in this article, can easily be reproduced using the VisTrails software available at www.vistrails.org. The data and metadata associated with each visualization in this, and previous articles, is available at www.vistrails.org/index.php/CiSE.

The volumetric dataset that we chose for this example is a CT scan of a chest as shown in Figure 4. With this dataset we are able to simultaneously pull out various recognizable features, including the bones, heart, and lungs. The exploratory steps to create this visualization are as follows:

- The first step is to create a pipeline that can read the data from file and render it to the screen. Because of the large data size, the choice in rendering algorithms is important. We chose to use texture slicing for quick interactions during the exploratory task. Once the necessary modules have been added and connected, the pipeline can be executed to view the data using a default transfer function. The resulting image in an opaque cube.

- The next step to rendering the data is to make sure that it is scaled in an intuitive way. A factor of 1.5 is used in the axial direction to account for the spacing between slices in the original scan.

- The volume contains superfluous regions that can easily be removed with clipping planes, such as the table. In addition, to expose the internal organs, a clipping plane can be added to the front of the chest

to remove the skin and bone from the visualization. With these clipping planes, we begin to reduce the amount of data being shown and internal features start to become apparent.

- The next, and most time consuming step, is to create a transfer function to emphasize the desired materials inside the volume. Because densities within CT volumes are similar with different scans, the specification is simplified because opacity can be easily limited to relevant scalar ranges. These ranges are also given colors to distinguish them (*e.g.,* white for bone, red for tissue).
- The final step in the volume rendering process is the direct interaction with the volume. This can be done by changing the viewing parameters, clipping planes, or the transfer function itself to explore other features within the volume.