Editors: Claudio Silva, csilva@cs.utah.edu
Joel E. Tohline, tohline@rouge.phys.lsu.edu

DIRECT VOLUME RENDERING

A 3D PLOTTING TECHNIQUE FOR SCIENTIFIC DATA

By Steven P. Callahan, Jason H. Callahan, Carlos E. Scheidegger, and Claudio T. Silva

Direct volume rendering is an effective method for plotting 3D scientific data, but it's not used as frequently as it could be. Here, the authors summarize direct volume rendering and discuss barriers to taking advantage of this powerful technique.

The use of plotting techniques to comprehend 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 scalar field's dimensionality increases, plotting becomes harder. For 2D scalar functions, many of us rely on more complex plotting functionality, similar to that available in certain scientific packages, such as the Visualization Toolkit (www.vtk.org) and matplotlib (http://matplotlib.sourceforge.net).

In general, available 2D plotting techniques are based on creating contour or density data plots. In either case, the algorithm samples the function in some way, which turns the problem into a discrete one. Contour plots show the function indirectly by generating a set of closed curves called *level sets*. Density plots show the function directly by mapping the scalar values through a set of colors. Figure 1 displays contour and density plots for a slice of 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 support only 3D contour plots, not density plots. To visualize 3D density plots, we must use *direct volume rendering*, the term indicating that no intermediate representations are computed. Instead, direct volume rendering determines pictures directly from the function $f(x, y, z)$, as Figure 1 shows. Volume visualization as a discipline started in the early 1980s, owing mostly to the medical community's needs for handling 3D data from computed tomography (CT) and magnetic resource imaging (MRI) scanners. That work has grown into a major research problem in the scientific visualization community.[1]

Direct volume rendering primarily offers flexibility—we can use it to obtain an initial overall view of the data, and, by changing transfer functions (which are directly analogous to color maps), we can incrementally focus on the data's particular features. In the past, direct volume rendering was too slow and cumbersome to be widely used as a plotting technique, but this hasn't been the case for several years. 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. Here, we attempt to show the technique's overall simplicity, its power, and how to best employ existing hardware and software solutions.

A Volume-Rendering Primer

In rendering volumetric data directly, we consider it a participating medium composed of semitransparent material that can emit, transmit, and absorb light, thereby letting us "see through" (or see inside) the data. By changing the material's optical properties, we can achieve different lighting effects.

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 doesn't necessarily make the visualization any clearer, we generally ignore it to make the algorithms more tractable. For a ray of light passing through the volume, we can compute the light's intensity I using the standard volume-rendering integral[2]

$$I(D) = I_0 e^{-\int_0^D \rho(t) A dt} + \int_0^D C(s) \rho(s) A e^{-\int_s^D \rho(t) A dt} ds \quad (1)$$

for position $s = 0$ at the back of the volume and $s = D$ at the eye; particles of area A and density per unit ρ; and emmisive glow C per unit of projected area. Because computers can efficiently perform volume rendering incrementally, we commonly use a discretized form of the equation in practice. We can derive an approximation to the equation using a Riemann sum, which divides the integral into n

 1521-9615/08/$25.00 ©2008 IEEE

equal segments of size Δx:

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

where

$$t_i = e^{-\rho(i\Delta x)A\Delta x},$$
$$g_i = C(i\Delta x)\rho(i\Delta x)A. \quad (4)$$

These equations require prior steps to compute the current step through the volume. Thus, we perform integration by sampling the volume incrementally, in order.

Direct volume-rendering algorithms consist of three major components: *sampling*, *classification*, and *compositing*. Sampling deals with selecting the piecewise steps that we take 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 we blend these classified steps together to form an image.

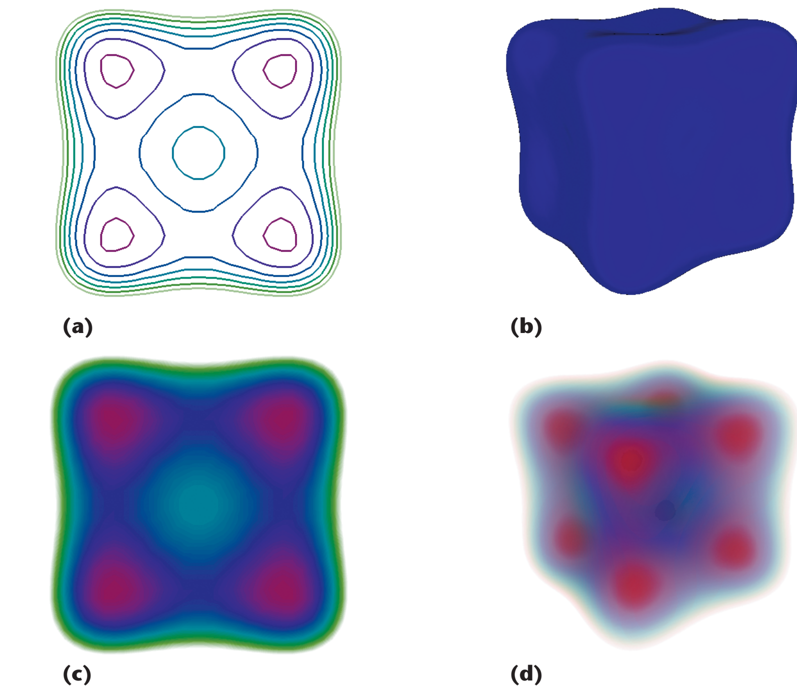

Figure 1. Plotting methods for a volume created from an implicit function. Indirect methods create intermediate geometry before rendering as shown with (a) 2D contours and (b) a 3D isosurface. Direct methods render the data directly either as (c) a density plot in 2D or (d) a volume rendering in 3D.

Sampling

We can represent a structured volume 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*. Because a discrete number of voxels exists within the grid, we perform volume integration in a piecewise manner by sampling incrementally through the volume. We can easily find the value at an arbitrary sample within a voxel using trilinear interpolation from the neighboring values. The specific manner in which we sample the volume depends on the volume-rendering algorithm; we discuss this concept further in a later section.

One obvious choice for the samples' positions throughout the volume is on the faces that define the voxel boundaries. However, sampling theory tells us that one sample per voxel won't be sufficient. Thus, in practice, we take multiple samples inside a voxel as well. We can adapt sampling frequency depending on the volume's homogeneity or on the user's preference for interaction speed over result quality.

Classification

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

The bottleneck for volume rendering performed in this manner occurs when we must compute the integral for each classification step. To address this issue, *pre-integrating* the volume-rendering integral replaces the expensive redundant computations with a simple table lookup.[3] In practice, we store the pre-integration in a 3D table that we can index via 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, we can even store this 3D table as a texture, which we can efficiently access during rendering.

Compositing

After we've classified a sample, the last step before moving to the next sample is to blend it with the previous samples using alpha compositing.[4] Just as re-

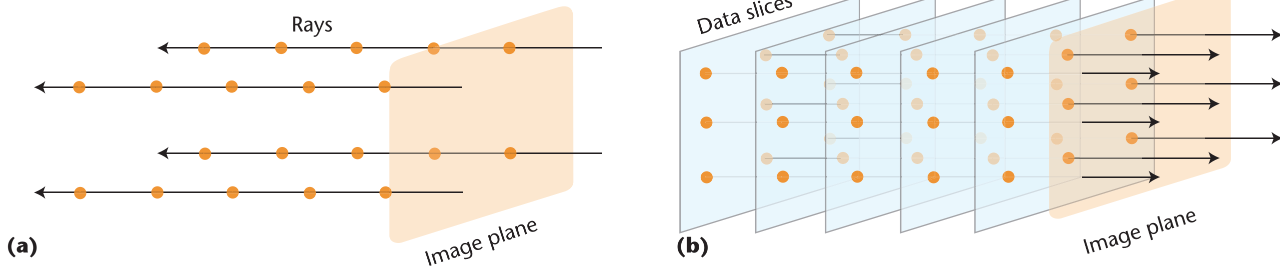

Figure 2. Volume rendering using (a) ray casting vs. (b) texture slicing. On CPUs, it's typically faster to cast rays through data, computing these independently. On GPUs, we can render independent slabs of the data (or slices) using special-purpose hardware, making volume rendering extremely fast.

arranging plates of different colored glass will change the color of objects seen through them, the order in which we composite the data will change the volume-rendering results. Thus, we must traverse the samples either back to front or front to back through the volume. We use the standard compositing algorithm for front-to-back traversal as a function of RGB color c and opacity α:

$$c_i = c_{i-1} + c_i\alpha_i\,(1 - \alpha_{i-1}) \quad (6)$$
$$\alpha_i = \alpha_{i-1} + \alpha_i(1 - \alpha_{i-1}) \quad (7)$$

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

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

Taking Advantage of the Latest Hardware

When it comes to actually implementing volume rendering, we can use 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's appropriate for CPUs, especially in recent multicore architectures. The second technique, known as *texture slicing*, exploits the special-purpose hardware present in recent graphics processing units (GPUs). Figure 2 shows a conceptual overview of these two techniques.

Ray casting[5] is an algorithm that performs a direct geometrical interpretation of Equation 2. The Riemann sum approximation becomes a set of co-linear line segments through the volume. These rays are cast from the image plane through the data set, 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 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 embarrassingly parallel—no dependencies exist between different rays in an image. Each pixel in the image plane corresponds to a different ray, so we can use parallel architectures very effectively to speed up ray-casting algorithms. The recent shift toward multicore 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, we must have already composited all the steps from 1 to $n - 1$. Across rays, however, there is no dependency. By changing the computation order, we arrive at a different scheme that's usually referred to as *texture slicing*.[6] Whereas ray casting generates one pixel at a time, marching the entire ray through the volume before moving to the next ray, slice-based techniques generate all the pixels simultaneously, marching all the rays through the volume one step at a time:

```
# slice-based
R = all_rays_in_screen()
for step in 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 minor, but it actually makes volume rendering trivial to implement in graphics processors. GPUs have become faster at a much faster pace than general-purpose CPUs, so algorithms that exploit GPUs tend to perform extremely well.[7] In GPUs, the computation of each slice through the data becomes a single call to an API that's 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, we can now perform volume rendering at interactive rates for essentially any structured Cartesian grid that fits in a graphics card's main memory.

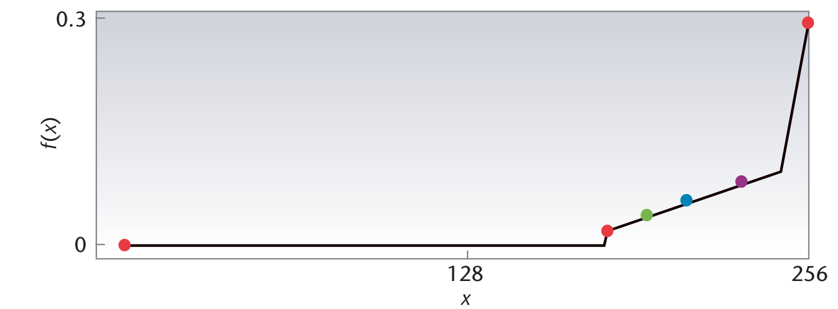

Figure 3. A simple representation of the transfer function $f(x)$ for the implicit data set in Figure 1. The black line shows the opacity function, and the points show changes in color.

Feature Finding for Volume Rendering

One of the drawbacks to rendering the complete volume is that it might result in information overload. Within the volume, features of interest can easily become obscured by regions of little interest. One way to remove superfluous regions is to insert clipping planes into the volume; these planes cull away parts of the volume on one entire side of the plane. Although efficient, clipping planes aren't powerful enough to isolate general homogenous regions within the volume—for this, we use transfer functions.

A transfer function is a simple mapping from scalar values to color and opacity—or, more formally, it maps $\Re \rightarrow \Re^4$ (that is, $s \rightarrow (r, g, b, a)$). A transfer function is generally represented as a lookup table that we can access using scalar values and that uses linear interpolation to represent a continuous range with a finite number of entries. Thus, we can give scalar ranges that we deem important a higher opacity and remove scalar ranges of little interest by specifying them as fully transparent. Figure 3 shows a plot of the transfer function used for the volume rendering in Figure 1. Note that we left values less than 180 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] Although researchers have introduced techniques to assist in specification, feature finding is still very manual. Prior knowledge of the data being visualized can help—for instance, CT scans provide a scanned object's densities, and for human tissue, such densities are well classified, thus finding regions of interest might only require highlighting the scalar ranges that correspond to those regions. In general, the goal behind the visualization should drive the transfer function specification. If the goal is to find boundaries between materials in the volume, a tool that uses differential properties of the scalar field (such as gradient magnitude) might be important. However, if the goal is to find one or more homogenous regions, a tool that shows the scalar values' 1D histogram might be more useful.

In general, because volume rendering is often exploratory, specifying transfer functions requires a lot of user intervention. Although this process can be time-consuming, it can also be beneficial because it gives insight to the data that we can't achieve using other 3D plotting techniques.

In this installment of Visualization Corner, we covered only the basics: direct volume rendering of regular grid data using simple specification techniques. In the future, we hope to cover the details in more depth and provide specific case studies of direct volume rendering in practice.

Acknowledgments

The US National Science Foundation, the US Department of Energy, an IBM faculty award, and an IBM PhD fellowship all partially supported this work.

References

1. C.R. Johnson and C. Hansen, *The Visualization Handbook*, Academic Press, 2004.
2. N.L. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 2, 1995, pp. 99–108.
3. K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading," *Proc. EG/SIGGRAPH Workshop Graphics Hardware*, ACM Press, 2001, pp. 9–16.
4. T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics*, vol. 18, no. 3, 1984, pp. 253–259.
5. R. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics*, vol. 22, no. 4, 1988, pp. 65–74.
6. B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *Proc. Symp. Volume Visualization*, ACM Press, 1994, pp. 91–98.
7. K. Engel et al., *Real-Time Volume Graphics*, AK Peters, 2006.
8. H. Pfister et al., "The Transfer Function Bake-Off," *IEEE Computer Graphics and Applications*, vol. 21, no. 3, 2001, pp. 16–22.

Steven P. Callahan is a research assistant and PhD candidate at the University of Utah. His research interests include scientific visualization, visualization systems, and computer graphics. Callahan has an MS in computational science and engineering from the University of Utah. Contact him at stevec@sci.utah.edu.

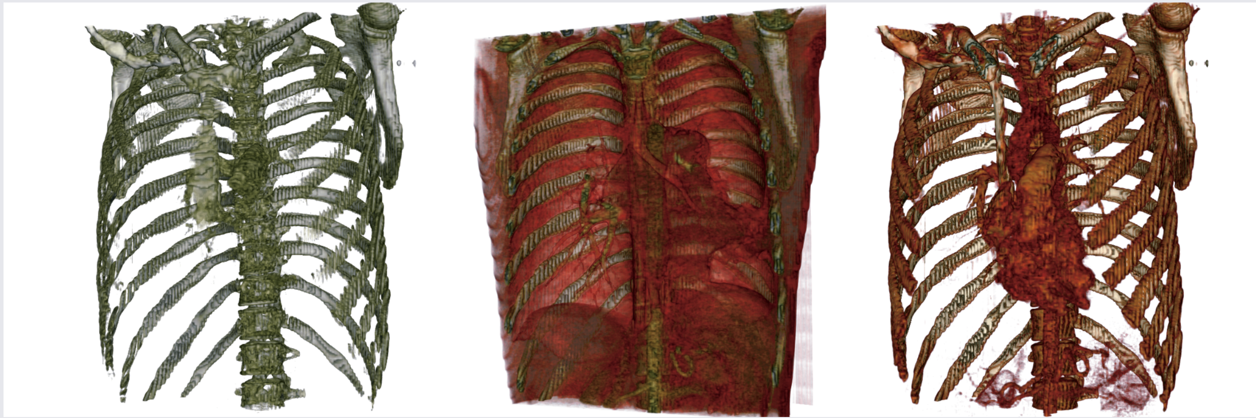

Figure A. Direct volume rendering example showing several different transfer functions. The data set comes from a computed tomography (CT) scan of a chest.

A Step-by-Step Example

Many tools are available for direct volume rendering of structured data. Here, we walk through a step-by-step example of how to create a visualization from scratch using the Visualization Toolkit (www.vtk.org) with the VisTrails system. Because interacting with the data itself can be an efficient learning tool, we recommend that readers explore it directly. You can easily reproduce this example, as well as all the visualizations in the main text, using the VisTrails software available at www.vistrails.org.

The volumetric data set that we chose for this example is a computed tomography (CT) scan of a chest (see Figure A). With this data set, we can simultaneously pull out various recognizable features, including the bones, heart, and lungs. The exploratory steps for creating this visualization are as follows:

- First, we create a pipeline that can read the data from a file and render it to the screen. Because of the large data size, the rendering algorithm we choose is important. We chose to use texture slicing for quick interactions during the exploratory task. Once we add and connect the necessary modules, we can execute the pipeline to view the data using a default transfer function. The resulting image is an opaque cube.
- Next, we make sure the data is scaled in an intuitive way. We use a factor of 1.1 in the axial direction to account for the spacing between slices in the original scan.
- The volume contains superfluous regions that we can easily remove with clipping planes, such as the table. In addition, to expose the internal organs, we can add a clipping plane 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 in CT volumes are similar with different scans, the specification is simplified because we can easily limit opacity to relevant scalar ranges. We also give these ranges colors to distinguish them (for example, white for bone and red for tissue).
- The final step is direct interaction with the volume. We can do this by changing the viewing parameters, clipping planes, or the transfer function itself to explore other features within the volume.

The data and metadata associated with each visualization in this—and previous—articles is available at www.vistrails.org/index.php/CiSE.

Jason H. Callahan is a research assistant and undergraduate student at the University of Utah. His research interests include scientific visualization and computer graphics. Contact him at jason@sci.utah.edu.

Carlos E. Scheidegger is a research assistant and PhD candidate at the University of Utah. His research interests include scientific visualization, geometry processing, and computer graphics. Scheidegger has a BS in computer science from the Federal University of Rio Grande do Sul, Brazil. Contact him at cscheid@sci.utah.edu.

Claudio T. Silva is an associate professor at the University of Utah. His research interests include visualization, geometry processing, graphics, and high-performance computing. Silva has a PhD in computer science from SUNY at Stony Brook. He is a member of the IEEE, the ACM, Eurographics, and Sociedade Brasileira de Matematica. Contact him at csilva@cs.utah.edu.