Chapter 7

Extensions

Data visualization is a process that requires experimentation. To support this experimentation a visualization application must provide flexible tools such as user-defined shading functions and techniques for focusing attention on a particular region of interest. The more control the user has, the more useful the application will be for a wide range of visualization problems.

The preceding chapters discuss the core component of a volume rendering application: fast, flexible rendering algorithms for volume data. This chapter discusses a collection of extensions to the rendering algorithms that provide a variety of visualization tools. Section 7.1 describes a shading algorithm, i.e. a method for computing the color of each voxel in a volume. The algorithm uses a technique proposed by Abram & Whitted [1990] to combine the flexibility of a shade-tree language [Cook 1984] with the speed of a lookup-table implementation.

Section 7.2 describes a fast depth cueing algorithm. Depth cueing is a technique for simulating fog in a rendered scene. The further an object is from the viewer the more the fog obscures it, thereby helping a viewer to distinguish between foreground and background objects. We present a new algorithm that uses a factorization of the depth cueing function to reduce the cost of computing the distance between the viewer and each voxel.

Section 7.3 describes a new algorithm for creating volume renderings with shadows. It uses the shear-warp rendering algorithm both to compute the illumination at each voxel and to render the image efficiently. Furthermore, it is a one-pass algorithm that requires significantly less memory than traditional two-pass methods that require a 3D buffer as large as the volume to store intermediate results.

Section 7.4 describes an extension of the shear-warp rendering algorithm for data sets containing mixtures of volume data and geometric data. Mixed data sets are useful for applications in which some of the objects to be rendered are analytically defined. The algorithm presented in this section has not yet been implemented.

Finally, Section 7.5 describes how to add support for arbitrary clipping planes to the shear-warp rendering algorithm. Clipping planes allow a user to render cut-away views of the interior of a volume or to focus attention on a subset of a volume. This extension has not been implemented either.

7.1 Flexible Shading With Lookup Tables

7.1.1 Shade Trees and Lookup Tables

Shading languages such as RenderMan [Upstill 1990] are widely used in photo-realistic rendering systems because they allow the user to specify a rich variety of shading functions. In the context of a visualization system a flexible shading language can be used to tailor the shading function for a specific data set. For example, a user might specify a function to highlight important aspects of the data or to produce appropriate perceptual cues. Languages that have been used for shading in volume rendering systems include a graphical network editor [Abram & Whitted 1990] for building shade trees [Cook 1984], the C programming language [Montine 1990], a specialized language called Vexpr that can be translated into C [Palmer et al. 1993], and an extension of RenderMan tailored for volume data [Corrie & Mackerras 1993a].

However, flexibility often comes at the price of slower execution. Most shading systems that support a general shading language operate by compiling a shading language expression into machine code or an intermediate language and then evaluating the compiled function for every rendered voxel. Compiler optimizations help to reduce the cost of evaluating the function [Hanrahan & Lawson 1990] but the cost increases as the number of operations in the expression increases. Unless the shading expression is trivial, this solution is too computationally expensive for interactive applications.

An optimization commonly used to reduce the cost of a shading function is to tabulate values of the function in a precomputed lookup table. The cost of evaluating a tabulated function depends only on the number of parameters to the function, not on the number of operations in the original function.

Implementing a shading function with a lookup table has two potential problems. First, there is a tradeoff between the size of the table and the quality of the results since the shading function must be discretized. In practice this problem is less severe for volume visualization applications because volume data is typically noisy, unlike analytically-defined polygonal data, so the banding artifacts caused by quantization errors are less noticeable. Thus we can represent a shading function with a smaller table than would be necessary for polygon-based applications.

The second problem with lookup table implementations is the combinatorial explosion that results as the number of parameters to the shading function increases. A shading function with a large number of parameters requires a lookup table with many dimensions, resulting in excessive memory consumption and high computational costs whenever the table must be recomputed.

We adopt the following solution: instead of using a single multi-dimensional table, we precompute several tables with low dimensionality and at run time we evaluate a simple expression to combine values from the individual tables.

We will represent a shading function by a tree called a *shade tree* [Cook 1984]. Each leaf node in the tree represents a constant or a voxel parameter, each internal node represents an operator that combines its children into a single value, and the root node represents the final value produced by the shading function. Before rendering the volume we precompute lookup tables for selected subtrees of the shade tree. The result is a collapsed shade tree in which many of the nodes have been replaced by the lookup tables so the cost to evaluate the tree at rendering time is small. This method was proposed by Abram & Whitted [1990] and used for volume rendering by Westover [1990].

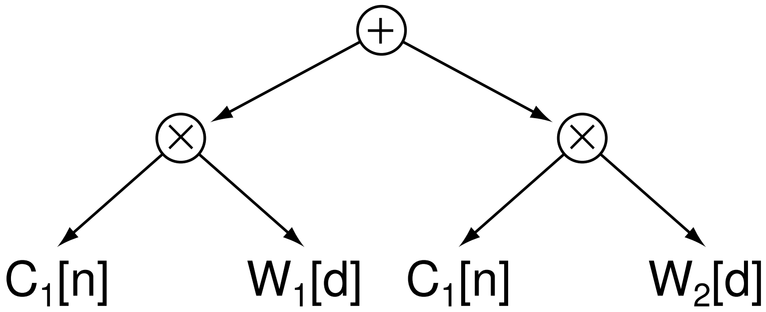

Figure 7.1: An example shade tree for the multiple-material shading model with two materials. Each leaf node is a lookup table indexed by either the scalar value d or the normal vector $\mathbf{n}$ stored with a voxel.

7.1.2 Implementation of Shading Functions

This section illustrates our implementation of shading functions with a specific example from our volume rendering system. The implementation described here is also the shader we used in our performance tests in Chapter 5.

We have used the combination of shade trees and lookup tables to implement the multiple-material shading model proposed by Drebin et al. [1988]. This shading model assumes that each voxel consists of a mixture of several material types, for instance skin, bone and fat in a medical data set. We use a lookup table indexed by the scalar value of a voxel to produce a set of weights indicating the fraction of each material type in the voxel. Next, we use one lookup table per material type to produce the color of that material. Finally, to compute the color of a voxel the shader looks up the color of each material type present in the voxel and then combines the colors in proportion to the weights.

Figure 7.1 shows a shade tree for the multiple-material shading model with two materials. The leaf nodes labeled C_i are the lookup tables containing colors for each material and the leaf nodes labeled W_i are the lookup tables containing material weights. To shade a voxel the shader looks up one value from each table, multiplies each color by its associated weight, and adds all of the weighted colors. Our implementation supports an arbitrary number of materials.

The lookup tables containing material weights are defined by the user. The lookup tables for computing the color of each material contain values precomputed with the Phong illumination model [Foley et al. 1990]. These tables are indexed by the surface normal vector associated with the voxel, which we will describe below. The equation used to compute each color is:

$$L = L_a + \sum_i L_i k_d(\mathbf{n} \cdot \mathbf{l_i}) + L_i k_s(\mathbf{n} \cdot \mathbf{h_i})^s$$

where:

L is the radiance reflected by the material from the light sources to the viewer. The value of the color C used in the volumetric compositing equation (Equation 1.4) equals L.

$\mathbf{n}$ is the normalized surface normal vector at the voxel.

$\mathbf{l_i}$ is a unit vector specifying the direction to light source i.

$\mathbf{h_i}$ is a unit vector equal to the surface normal that results in the maximum specular highlight for light source i:

$$\mathbf{h_i} = \frac{\mathbf{l_i} + \mathbf{v}}{|\mathbf{l_i} + \mathbf{v}|}$$

where $\mathbf{v}$ is a unit vector that specifies the direction from the voxel to the viewer.

L_i is the radiance emitted by light source i.

L_a is the radiance reflected by the material from ambient light.

k_d is the diffuse reflection coefficient for the material.

k_s is the specular reflection coefficient for the material.

s is the shinyness exponent for the material.

For color images this equation is evaluated once for each color channel to produce an RGB lookup table. Each material type has a different set of material parameters (L_a, k_d, k_s and s).

We make the assumption that all of the light sources are infinitely distant so that the vectors $\mathbf{l_i}$ are constant for every voxel. For parallel projections the viewer is also infinitely distant, so $\mathbf{v}$ and hence $\mathbf{h_i}$ are also constant. For perspective projections we can define $\mathbf{v}$ to be the direction of the central viewing ray; this definition will result in inaccurate specular highlights but the approximation may be acceptable unless the viewer is close to or inside the volume. Under these assumptions the value of L and hence the value of C_i in each of the subtrees depends on only one variable, the surface normal vector $\mathbf{n}$. Thus each subtree can be tabulated in a one-dimensional lookup table indexed by the surface normal vector.

The surface normal at any point in the volume is defined to be a unit vector parallel to the local gradient of the voxel scalar value $d(x, y, z)$:

$$\mathbf{n}(x, y, z) = \frac{\nabla d(x, y, z)}{|\nabla d(x, y, z)|}$$

The gradient is approximated using the central difference gradient operator:

$$\begin{aligned}\nabla d(x, y, z) = & \quad \frac{1}{2}\left[d(x+1, y, z) - d(x-1, y, z)\right]\mathbf{i} \\ & + \frac{1}{2}\left[d(x, y+1, z) - d(x, y-1, z)\right]\mathbf{j} \\ & + \frac{1}{2}\left[d(x, y, z+1) - d(x, y, z-1)\right]\mathbf{k}\end{aligned}$$

The surface normal vectors are precomputed and stored with the voxels. Instead of storing all three components of the normal as floating point numbers we use an encoded representation that requires less storage and provides a convenient integer index for the lookup tables [Glassner 1990b, Fletcher & Robertson 1993]. The general method for encoding normal vectors works as follows:

1. Choose a set of unit vectors that are approximately uniformly distributed over all directions.

2. Assign a unique integer code to each vector in the set.

3. Compute the gradient at each voxel and find the closest unit vector in the predetermined set of unit vectors. Represent the normal by the integer code for the chosen unit vector.

The set of unit vectors in our implementation is found by normalizing the solutions of the equation:

$$|n_x| + |n_y| + |n_z| = 1$$

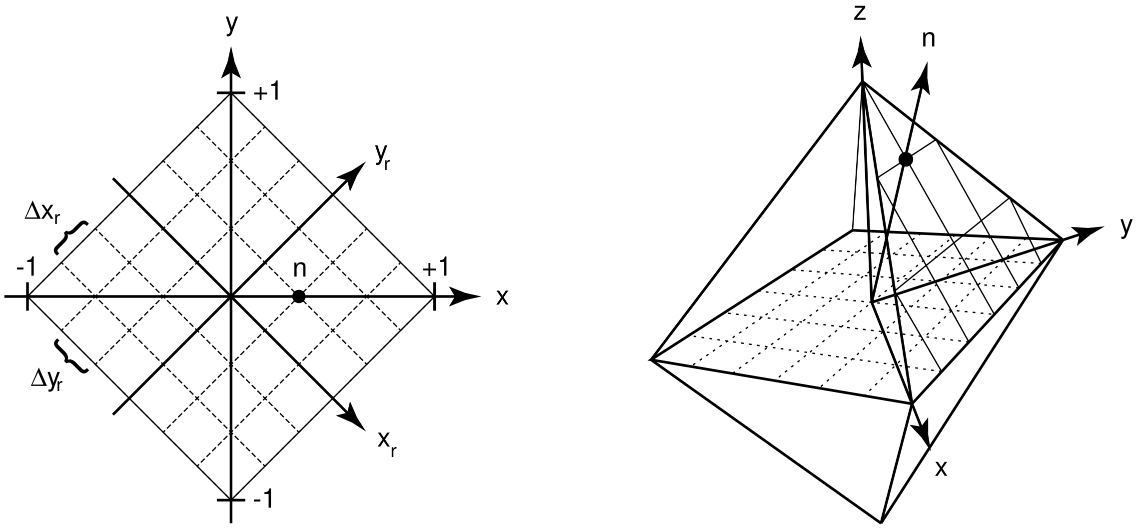

Figure 7.2: Grid locations for quantizing normal vectors. The grid (left) forms the base of a pyramid (right) with equation $|n_x| + |n_y| + |n_z| = 1$. Each vector is constrained to pass through the origin and a point on the surface of the pyramid that projects along the z axis onto a grid point.

where n_x and n_y are constrained to locations on a grid centered at the origin and rotated 45 degrees, as shown in Figure 7.2. The solutions are of the form:

$$\begin{aligned} n_x &= \frac{y_r + x_r}{\sqrt{2}} \\ n_y &= \frac{y_r - x_r}{\sqrt{2}} \\ n_z &= \pm(1 - \frac{|y_r + x_r| + |y_r - x_r|}{\sqrt{2}})) \\ \text{where} & \\ x_r &\equiv x_i \Delta x_r \\ y_r &\equiv y_i \Delta y_r \end{aligned}$$

for integers x_i, y_i and grid spacings Δx_r, Δy_r (shown in the figure). Each vector can be encoded by concatenating x_i, y_i, and a bit indicating the sign of n_z. This particular definition for the set of quantized normal vectors allows for easy conversion between the encoded and unencoded representations and has a nearly-uniform distribution of directions. We have

Shader Implementation	Shading Time (sec.)	Total Time (sec.)	Normalized Time
No Lookup Tables	13.3	14.5	5.2
Lookup Tables	2.3	3.4	1.2
Lookup Tables w/ Inlined Procedures	1.6	2.8	1.0

Table 7.1: Performance results for three implementations of the shading function in Figure 7.1 to produce the image in Figure 5.5: evaluation without lookup tables, evaluation with lookup tables, and evaluation with lookup tables and an inlined shading procedure.

found that 13 bits (6 bits for each of x_i and y_i and 1 bit for n_z) is sufficient to avoid objectionable quantization errors.

The lookup tables for the color of each material (C_i) are recomputed whenever the user modifies any of the constants in the lighting model, such as the viewing direction or the material parameters. Each table consists of 8192 entries (the total number of surface normal vectors), so in large volumes the cost of computing the tables is significantly smaller than the cost of shading every rendered voxel.

Table 7.1 compares the cost of three implementations of the shade tree shown in Figure 7.1 for the case of two materials, one light source, and a color (RGB) image. In the first implementation, for every voxel the renderer calls a shading procedure that evaluates the entire tree without using lookup tables for C_i. In the second implementation the shading procedure uses the lookup tables. In the third implementation the lookup-table shading procedure is inlined into the rendering procedure to eliminate the overhead of a procedure call (including register spills due to the procedure calling conventions). The first column is the time for shading alone, the second column is the total time to render an image, and the third column is the total time normalized by the time to compute the image with the fastest implementation.

For the case of the shading expression in this example the lookup table implementations are five times faster than the implementation without precomputation, and the inlined procedure is about 20% faster than the non-inlined procedure. This example shows that even with a fairly simple shading model the computational costs can be very high in a procedural implementation, but a lookup-table implementation can reduce the cost to acceptable levels.

7.1.3 General Shade Trees

The current implementation allows shade trees of the form shown in Figure 7.1: a sum-of-products expression with an arbitrary number of product terms and one-dimensional lookup tables at the leaves. The implementation uses predefined macros that can be inlined into the rendering code. Even with this limited model many shading functions can be specified by changing the contents of the lookup tables. More general shading expressions could be supported in a straightforward manner by using a shading language system with a compiler and a dynamic linking facility as described in [Corrie & Mackerras 1993a]. The user would specify hints to indicate which subtrees should be tabulated (as proposed by [Abram & Whitted 1990]).

An area for future research is to develop automatic partitioning algorithms that can choose an optimal set of subtrees for tabulation. An optimal set minimizes the number of computations in the collapsed tree and the total size of the tables. The compiler should produce two procedures: a procedure for precomputing the lookup tables, and a procedure for rendering a volume with an inlined shading function implementing the collapsed shade tree (along the same lines as the specializing shaders of Guenter et al. [1995]). Such a system could provide the flexibility of existing shading languages with nearly the same performance as hand-optimized hard-wired shading systems.

7.2 Fast Depth Cueing

7.2.1 Depth Cueing

Depth cueing is a technique for simulating atmospheric attenuation of light, for instance due to uniform absorbing, non-scattering fog. The fog causes objects far from the viewer to appear darker than foreground objects. In visualization applications the technique can be used to emphasize depth relationships between different parts of a data set. Figure 7.3 shows two volume renderings with and without depth cueing.

Depth cueing can be implemented in a straightforward manner by computing the distance from the viewer (the depth) for each rendered voxel and attenuating the voxel's color

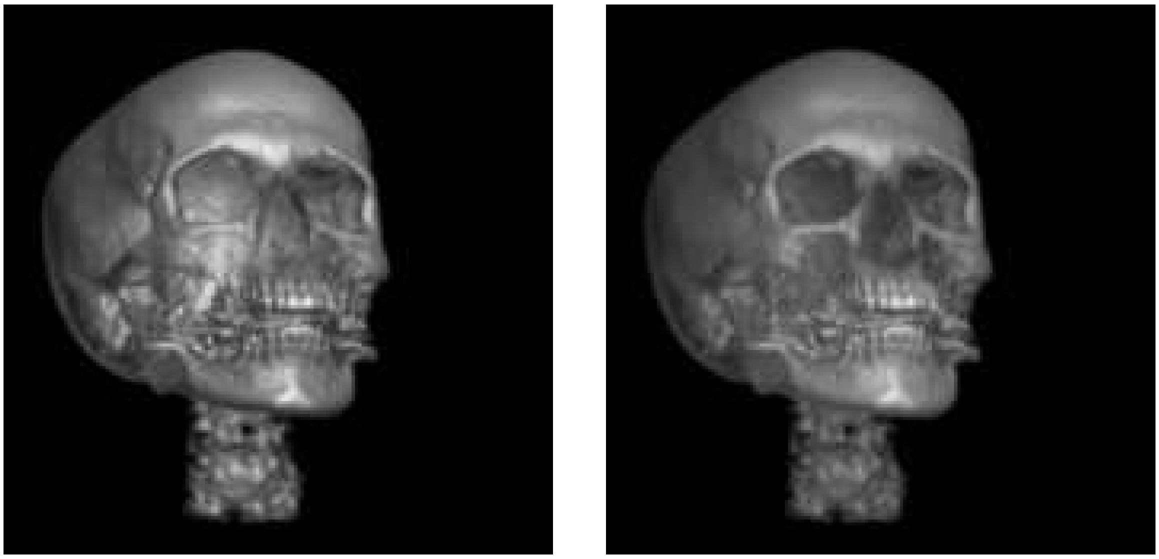

Figure 7.3: The left image is a rendering of a CT data set without depth cueing, and on the right is the same rendering with depth cueing. The more distant surfaces are darker in the depth cued image.

by a function of the depth. One standard depth cueing function is an exponential decay function:

$$T_{\text{fog}}(d) = T_0 e^{-\rho d}$$

where $T_{\text{fog}}(d)$ is the fraction of light transmitted after traveling a distance d through the fog, and T_0 and ρ are user-defined constants. If the color of a voxel before depth cueing is C then after depth cueing the color is $C \cdot T_{\text{fog}}(d)$. To avoid the cost of evaluating the exponential during rendering, the depth cueing function can be precomputed and stored in a lookup table.

The computational cost of depth cueing is a table lookup and a multiply for each rendered voxel plus the cost of computing the voxel depths. Either the depths can be computed from the viewing transformation matrix for every rendered voxel, or the depths can be computed incrementally by adding a delta every time the rendering algorithm moves to a new voxel. The incremental approach is usually less expensive: for a parallel projection the depth delta is constant, so a single addition per voxel is required. For a perspective projection a divide is also necessary.

However, in the shear-warp algorithm and other object-order algorithms that use spatial data structures the depth must be updated not only for rendered voxels but also every time a

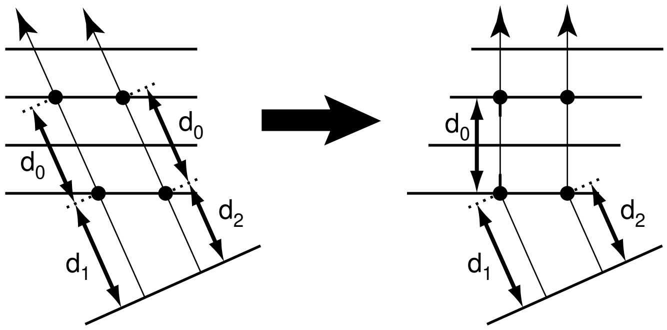

Figure 7.4: Factoring the depth cueing function: The distance from a voxel to the image can be split into two pieces, the distance from the voxel to the intermediate image and the distance from the intermediate image to the image. For a parallel projection the distance to the intermediate image is constant for every voxel in a slice.

run of transparent voxels or opaque pixels is skipped. The cost is therefore proportional to the number of rendered voxels plus the number of runs. When depth cueing is implemented using incremental calculation of the depth with floating-point arithmetic the time to render the depth cued image in Figure 7.3 increases by 20% compared to rendering without depth cueing.

7.2.2 Factoring the Depth Cueing Function

In this section we develop a faster depth cueing algorithm using a factorization of the depth cueing function. As shown on the left side of Figure 7.4, the distance d from a voxel to the image plane can be broken into two pieces: the distance d_0 from the voxel to the front voxel slice, plus the distance d_i $(i = 1, 2, \ldots)$ from the front slice to the image. The exponential depth cueing function can be written as a product of two factors involving these two distances as follows:

$$T_{\text{fog}}(d) = T_0 e^{-\rho d} = e^{-\rho d_0} \cdot T_0 e^{-\rho d_i}$$

The right side of the figure shows that for a parallel projection the distance d_0 is constant for every voxel in a slice. As a result, depth cueing can be performed using a two-step process:

1. During rendering, multiply the color of every rendered voxel in a given slice by a constant depth cueing factor. The factor is:

$$T_{\text{slice}} = e^{-\rho d_0}$$

2. After all of the voxel slices have been composited together, multiply the color of every pixel in the intermediate image by a second depth cueing factor which must be computed individually for every pixel:

$$T_{\text{pixel}} = T_0 e^{-\rho d_i}$$

The first step of the algorithm accounts for light attenuation as the voxels are projected onto the plane of the intermediate image (which is coincident with the front voxel slice), and the second step accounts for light attenuation from the intermediate image plane to the final image plane.

The advantage of the algorithm is that the depths of individual voxels need not be computed. During the projection stage of the algorithm T_{slice} must be computed only once per slice. The cost to apply the depth cueing factor is a single multiply for each rendered voxel. During the warping stage the cost of applying the second depth cueing factor is proportional to the number of pixels in the intermediate image. The cost per pixel is one add (to incrementally compute the depth of the pixel), one table lookup (to retrieve the value of the depth cueing function), and one multiply. The 2D intermediate image is generally much smaller than the volume so the cost of applying the second depth cueing factor is also small.

For perspective projections the value of d_0 is not constant for every voxel in a slice. However, it can be computed inexpensively from the sample spacing used for opacity correction (see Section 4.2.3):

$$d_0 = k \cdot \Delta x$$

where k is the voxel slice number (counting from zero for the front slice) and Δx is the distance between slices along a viewing ray in image space. Δx must be stored with each intermediate image pixel for the opacity correction algorithm. Thus the depth cueing factor can be computed using one multiply and one table lookup per rendered voxel, in addition

Algorithm	Depth Cueing Time (msec.)	Total Time (msec.)	Normalized Time
No Depth Cueing	0	780	1.00
Fast Depth Cueing	40	820	1.05
Standard Depth Cueing	160	940	1.21

Table 7.2: Performance results for fast depth cueing compared to straightforward depth cueing and rendering without depth cueing. The first column is the time for depth cueing alone, the second column is the total time to render an image, and the third column is the total time normalized by the time to compute the image without depth cueing. These times are for rendering the data set in Figure 7.3 with a parallel projection.

to the work already required for opacity correction.

7.2.3 Implementation of Fast Depth Cueing

Table 7.2 shows the performance of the fast depth cueing algorithm compared to the standard depth cueing algorithm with incremental calculation of the voxel depths, and the rendering algorithm without depth cueing. The overhead of the fast depth cueing algorithm is only 5%, compared to 20% for the standard algorithm. Although the performance gain is not large compared to the benefit of coherence accelerations and fast shading algorithms, the simplicity of the fast depth cueing algorithm and the modest performance boost make it worth implementing.

A potential pitfall in the implementation of the fast depth cueing algorithm is that the intermediate image pixels must be represented using floating point numbers rather than a fixed-point representation with a small dynamic range. A large dynamic range is necessary because the first factor of the depth cueing equation sometimes "over attenuates" the color of a voxel, possibly resulting in a very small value which will later be restored to a higher value by the second depth cueing factor. This case occurs if the distance from the intermediate image to the final image (d_i) is negative for any intermediate image pixels. If an eight-bit integer representation is used for intermediate image pixels then the intermediate intensities may be incorrectly rounded to zero. We use a floating point representation both to eliminate this problem and to reduce round-off errors while compositing voxels into the intermediate image.

7.3 Rendering Shadows with a 2D Shadow Buffer

7.3.1 Algorithms for Rendering Shadows

The volume rendering equation (Equation 1.3) and the local illumination models described earlier in this chapter do not model occlusions between the voxels and the light sources. As a result, images produced using these methods do not contain shadows. Shadows are useful in certain visualization applications because they help the user to perceive depth relationships, and with properly-arranged lighting shadows can enhance small details by improving contrast.

Shadows can be incorporated into the basic volume rendering model by adding a second pass to the rendering algorithm: during the first pass through the volume the algorithm computes the illumination reaching each voxel, and during the second pass the algorithm generates an image. An additional pass is necessary for each light source if there is more than one light. The original version of this algorithm is due to [Kajiya & Von Herzen 1984] who used it in a ray caster. First, for each light source their algorithm casts rays through the volume from the light source and accumulates opacity along the rays. The accumulated opacity at each sample point along a light ray determines the illumination reaching that point in the volume. These illumination values (or equivalently, the accumulated opacities) are stored in a 3D array with the same dimensions as the volume. The array is called a *shadow buffer*. During the final rendering pass the shader uses the illumination values from the shadow buffer to compute the color of each voxel. Levoy [1989] also describes an implementation of this algorithm.

The primary disadvantages of the algorithm are that it requires a 3D shadow buffer as big as the volume to store the illumination values and it requires multiple passes through the volume. However, Grant [1992] describes an alternative method for computing the illumination using a much smaller 2D shadow buffer. This algorithm performs a single pass through the volume, computing illumination and the image simultaneously.

Grant's algorithm operates by sweeping the 2D shadow buffer through the volume in a direction that visits each voxel in depth-sorted order with respect to both the light source and the viewer, as shown in Figure 7.5. As in Kajiya and Von Herzen's algorithm the buffer contains illumination values. The algorithm sweeps the 2D shadow buffer through space until it

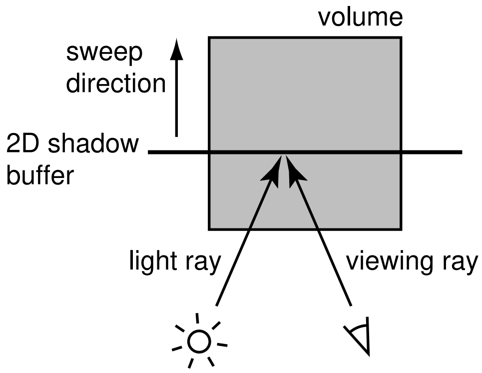

Figure 7.5: Diagram of Grant's shadow mask sweep algorithm [Grant 1992]. The 2D shadow buffer stores the illumination reaching the current slice of voxels. The rendering algorithm sweeps the buffer through the volume and updates the illumination values at the same time as the image is computed.

reaches a new slice of voxels. It uses the illumination values in the buffer to shade the voxels and then projects the voxels into the image along the viewing direction. The algorithm then projects the voxels into the shadow buffer along the light direction and updates the illumination in the buffer. Because the traversal order is depth-sorted with respect to both the light and the viewer, the information necessary for shading a voxel is always present in the depth buffer at the required time.

In order for the algorithm to work properly the volume must be traversed in front-to-back order with respect to the light source. If the light source and the viewer are both on the same side of the volume than the traversal order will also be front-to-back with respect to the viewer, as shown in the figure. If the angle between the light direction and the view direction is greater than 90 degrees then the traversal order may be back-to-front with respect to the viewer, which would make it impossible to implement early-ray termination during image formation.

Our shadow rendering algorithm combines the 2D shadow buffer with the shear-warp volume rendering algorithm. The algorithm sweeps the shadow buffer through sheared object space parallel to the slices of the sheared volume. The algorithm uses two shear-warp factorizations simultaneously: one for the viewing transformation matrix, and one for the light projection matrix. Each voxel slice must be translated and resampled twice, first to composite it into the intermediate image for the image formation computation and then to

```
procedure RenderVolumeWithShadows()
    view_factors = Factor(ViewMatrix);
    light_factors = Factor(LightMatrix);

    Clear(TmpImage);
    Clear(ShadowBuffer);

    foreach (voxel_slice from front to back)
        light_attenuation = Resample(ShadowBuffer, light_factors);
        TmpImage = TmpImage over Resample(Shade(voxel_slice, light_attenuation), view_factors);

        ShadowBuffer = ShadowBuffer over Resample(voxel_slice, light_factors);
    end

    Image = Warp(TmpImage);
    Display(Image);
end
```

Figure 7.6: Pseudo-code for the shadow rendering algorithm.

composite it into the shadow buffer for the illumination computation. The algorithm uses a run-length encoded representation for the shadow buffer (the same data structure as for the intermediate image) so that the illumination computation can take advantage of coherence in the shadow buffer: voxels that are already in shadow need not by composited into the shadow buffer.

7.3.2 Implementation of the Shadow Rendering Algorithm

Figure 7.6 is a pseudo-code description of the shadow rendering algorithm for an infinitely-distant light source. First, the algorithm factors the viewing transformation matrix and the lighting transformation matrix (which transforms the light direction vector from the coordinate system of the light source into the image coordinate system). The rendering algorithm cannot proceed unless the principal axis is the same for both factorizations, i.e. both factorizations shear the same set of slices of the volume. We discuss this limitation later in this section.

Next, the algorithm clears the intermediate image (TmpImage) and the shadow buffer (ShadowBuffer). The shadow buffer is used to store the accumulated opacity of the voxels between the light source and the voxel slice currently being rendered; the illumination can be easily computed from this opacity.

The algorithm then loops through the slices in front-to-back order with respect to the light source. Inside the loop the first step is to determine how much light reaches each voxel. The algorithm determines the opacity between a voxel and the light source (light_attenuation) by sampling a value from the shadow buffer. Because the algorithm is based on the shear-warp factorization, the transformation from the shadow buffer to the voxel slice only requires a translation. The factored lighting transformation (light_factors) determines how much the shadow buffer must be translated. Resampling is required since the translation distance might not be an integer (but no scaling or rotation is necessary). Next the algorithm uses the light attenuation to shade each voxel. Finally the algorithm resamples the shaded voxels and composites them into the intermediate image, which completes the image formation step.

The remainder of the voxel processing loop handles the illumination step. The algorithm resamples the voxel slice using the factored lighting transformation and composites the result into the shadow buffer. When the compositing loop completes, the algorithm warps the intermediate image and displays the result as in the standard shear-warp rendering algorithm.

Either the parallel projection algorithm or the perspective projection algorithm can be used in the image generation step inside the rendering loop. Similarly, the parallel projection algorithm can be used in the illumination step for infinitely-distant light sources or the perspective projection algorithm can be used for finite-distance light sources. In the case of finite-distance lights the transformation from the shadow buffer to the voxel slice includes a uniform scale as well as a translation (since the shadow buffer must be scaled to larger sizes as the distance from the light source increases), so the resampling step on line 7 becomes more expensive than for the infinite-distance case. A potential problem occurs if the size of the volume is comparable to or larger than the distance to the light source, since the scale factor may then become large enough to cause distortion. The current implementation does not support finite-distance light sources.

Multiple light sources can be supported by the algorithm provided that the light directions are all consistent with the same principal axis and a separate shadow buffer is used for each light source.

All shadow rendering algorithms for volume data have a problem caused by surfaces shadowing themselves. Voxelized surfaces in band-limited volumes are several voxels thick, so voxels on the outer edge of a surface cast shadows on the partially-visible voxels inside the same surface. As a result the surfaces appear darker than expected, and self-shadowing can also introduce severe aliasing artifacts. We solve this problem by translating the shadow buffer a few voxels away from the light source during the image formation step [Williams 1978, Reeves et al. 1987, Levoy 1989].

Implementation of the shadow bias in the shear-warp shadow rendering algorithm is straightforward: in the compositing loop, instead of compositing the same slice of voxels (slice number k for example) into both the intermediate image and the shadow buffer, the algorithm composites slice number k into the intermediate image and slice number $k-\Delta k$ into the shadow buffer where Δk is the shadow bias. The shift by Δk slices in the illumination pass translates the slice along the direction of the light rays, as required. The shadow bias reduces the accuracy of shadows produced by small objects, but it also reduces the aliasing artifacts associated with self-shadowing.

The shadow rendering algorithm cannot be parallelized using the task partition in Chapter 6 because of the additional ordering constraints introduced by interleaving the illumination and image formation steps.

7.3.3 Performance of the Shadow Rendering Algorithm

Figure 7.7 shows an image generated with the shear-warp shadow rendering algorithm. The data set is the 256x256x225 voxel CT head data set. The image took 1.6 sec. to render, which is 2x longer than the image without shadows. This slowdown is typical since each voxel slice must be resampled and composited twice, although the volume must only be traversed once. The memory required by the algorithm is eight bytes per shadow buffer pixel (for the opacity and opaque pixel links), in addition to the memory required for the standard algorithm (see Table 5.3).

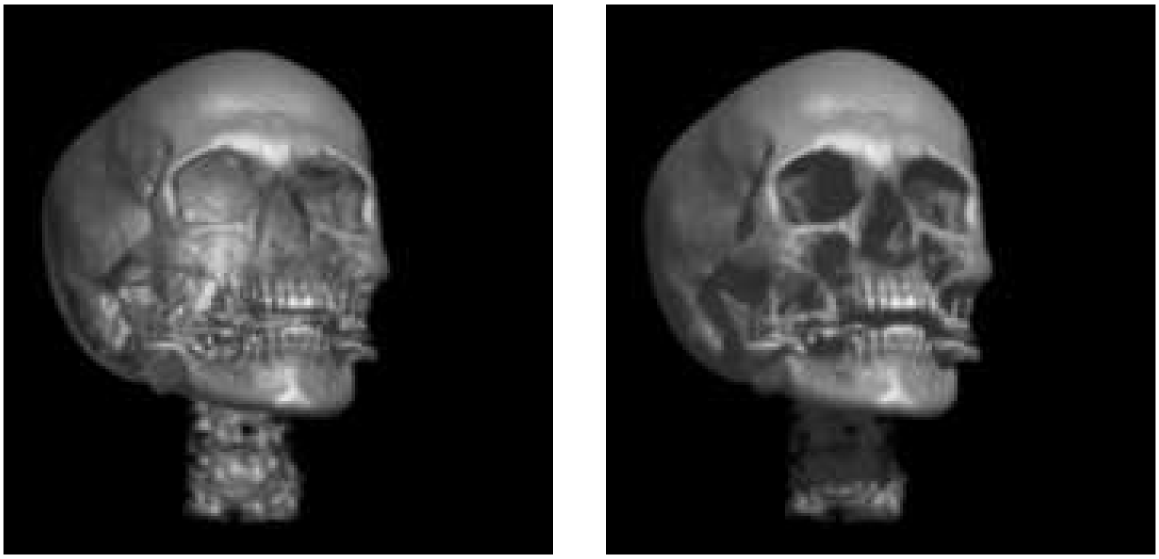

Figure 7.7: Volume rendering of a CT scan with the standard shear-warp rendering algorithm (left) and the shadow rendering algorithm (right). These renderings are not depth cued.

Compared to Grant's algorithm, the shear-warp shadow rendering algorithm is more efficient because filtering the values in the shadow buffer is less expensive. In Grant's algorithm, to compute the illumination required for shading a pixel fragment the algorithm must project the pixel fragment into the shadow buffer and the projected area must be sampled. This procedure may require decimating an arbitrarily-large area of the shadow buffer. Standard sampling acceleration techniques such as summed-area tables [Crow 1984] and mip-maps [Williams 1983] cannot be used since they require precomputation, whereas the shadow buffer is constantly modified during rendering. In the shear-warp algorithm the shear-warp factorization produces a simple mapping from voxels to shadow buffer pixels. For infinite-distance light sources the transformation is simply a translation, so filtering is trivial, and for finite-distance light sources the factorization guarantees that the shadow buffer will only be scaled to larger sizes (which results in less-expensive resampling than for the case of decimation).

Compared to a 3D shadow buffer algorithm the shear-warp shadow rendering algorithm places restrictions on the viewing direction and the light direction. As noted above, the shear-warp factorization for the view transformation matrix and the lighting transformation matrix must have the same principal axis. This requirement leads to cases that cannot be

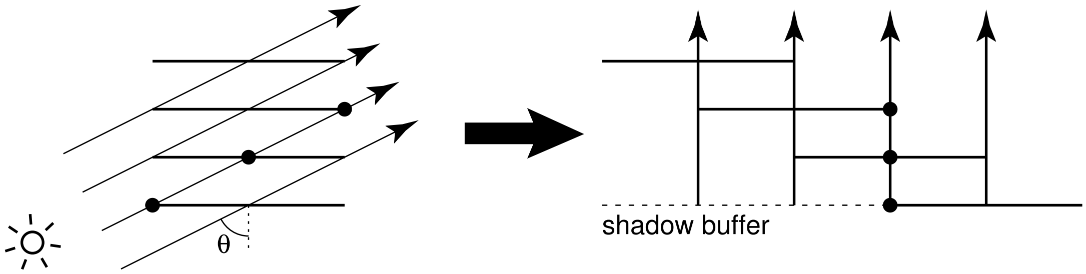

Figure 7.8: Limitations on the volume sampling rate in the shadow rendering algorithm: As the angle between the light vector and the principal viewing axis increases the volume shear increases. As a result the distance between samples along a viewing ray increases.

rendered. For example, if the view direction is perpendicular to one face of the volume and the light direction is perpendicular to another face then there is no consistent choice for the principal axis.

Furthermore, for a given viewing direction and a moving light source, as the light direction approaches an unrenderable orientation the image quality degrades. A larger angle between the light direction and the principal viewing axis corresponds to a larger shear in the shear-warp factorization. A larger shear is equivalent to a lower sampling rate along a viewing ray in a ray casting algorithm (Figure 7.8). As the sampling rate decreases below the Nyquist rate for the volume, aliasing artifacts appear in the shadow buffer and carry over into the rendered image.

The maximum acceptable angle between the light source and the principal viewing axis depends on the volume. For the CT head data set shown in Figure 7.7, the classified bone surfaces are roughly 2-3 voxels thick so the distance between image-space samples must not drop below about two voxels. This estimate agrees with our experiments: we achieve high-quality renderings of the data set with light-direction angles up to 60 degrees, which corresponds to a sample spacing of two voxels along a light ray (Figure 7.9). This restriction specifies the maximum angle between the light vector and the principal viewing axis. The maximum angle between the viewing direction and the principal viewing axis is 45 degrees, so for this volume the maximum angle between the light source and the viewing direction may range between 15 degrees and 105 degrees, depending on the viewing direction.

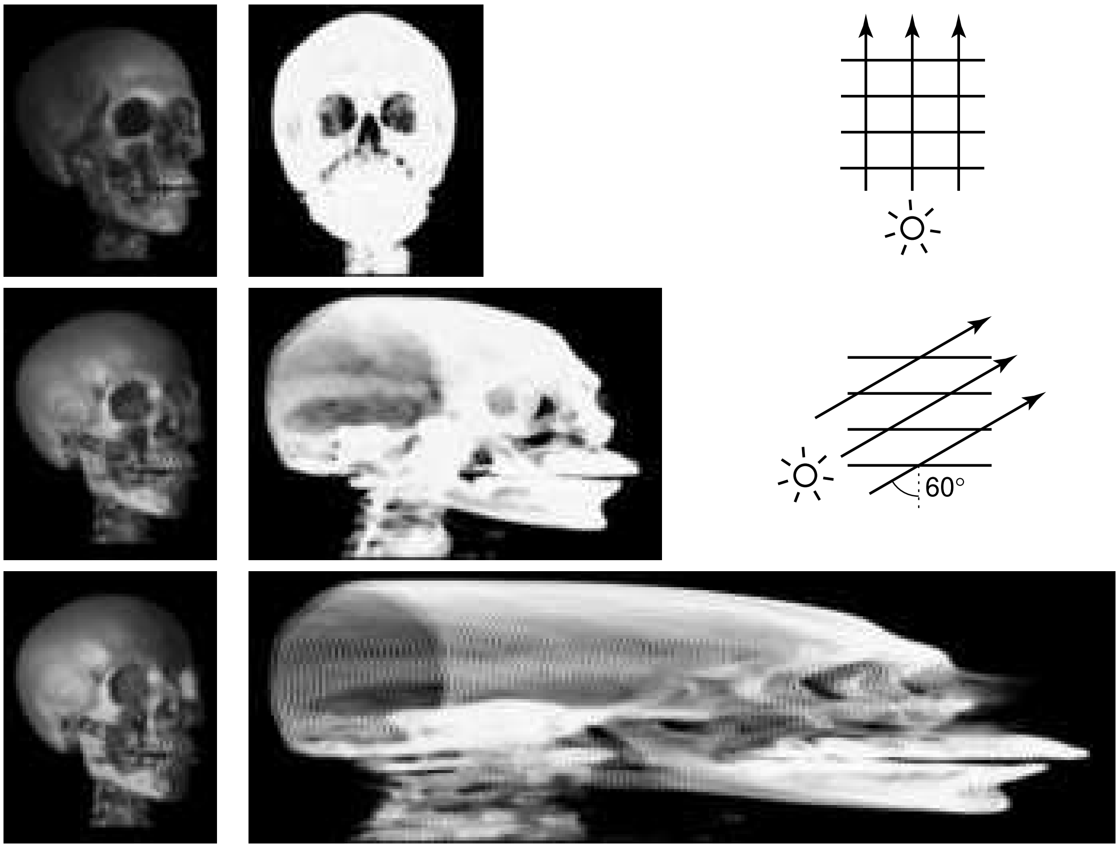

Figure 7.9: Aliasing in the shadow buffer: The first column of images shows three renderings of the CT head data set using the shadow algorithm. The viewing direction is constant for all three images but the angle between the light direction and the principal viewing axis varies (0 degrees for the top row, 60 degrees for the middle row, 78 degrees for the bottom row). The second column of images shows the contents of the shadow buffer after rendering half of the voxel slices in the volume. A large angle results in a large shear between adjacent voxel slices, so as the light angle approaches 90 degrees artifacts appear: in the bottom rendering the shadow behind the eye socket in the center of the image has stripes. The white stripes in the shadow buffer come from slices of the volume that have been translated far enough relative to each other to reveal gaps. This problem cannot be corrected unless a 3D resampling filter is used. However, up to an angle of 60 degrees there are no severe aliasing artifacts.

The angle restriction limits the usefulness of the shear-warp shadow algorithm for rendering arbitrary scenes. However, in a visualization application the user chooses the location of light sources to achieve an informative visualization rather than to simulate a real-world environment, so restrictions may be acceptable. Furthermore the speed of the algorithm makes it attractive for an interactive visualization application.

7.4 Rendering Mixtures of Volume Data and Polygons

Visualization applications sometimes require rendering mixtures of volume data and polygons. An example application area is radiation treatment planning: a radiation beam is defined analytically and therefore can be represented by a polygon mesh, whereas patient anatomy is most easily described via a scanned volume [Levoy et al. 1990]. Existing methods for rendering such mixed data sets include scan-converting the polygons into a volumetric representation that can be rendered with a conventional volume renderer [Wang & Kaufman 1994] and using a ray tracing algorithm adapted for both types of primitives [Levoy 1990b].

The shear-warp volume rendering algorithm can be combined with a modified version of a scanline polygon rendering algorithm to support mixed data sets directly. The algorithm has not been implemented, but in this section we describe a possible design.

The algorithm operates by alternating between two phases: rendering a slice of volume data, and rendering the portion of the polygon database lying between two voxel slices (Figure 7.10). The polygon rendering algorithm is a modified version of a scanline algorithm which renders each polygon incrementally, so the renderer can suspend work on a polygon after scan-converting the portion lying between two voxel slices. The algorithm also requires special cases to properly render polygons that intersect non-transparent voxels.

The standard scanline algorithm for rendering a polygonal scene works as follows [Foley et al. 1990]. The algorithm computes the image one scanline at a time using two data structures: an edge table and an active edge table. The edge table is an array of buckets, one bucket per scanline, each containing a list of polygon edges that start on a particular scanline. The active edge table is a list containing the edges that cross the scanline currently being rendered. The algorithm processes each scanline in turn by updating the active edge

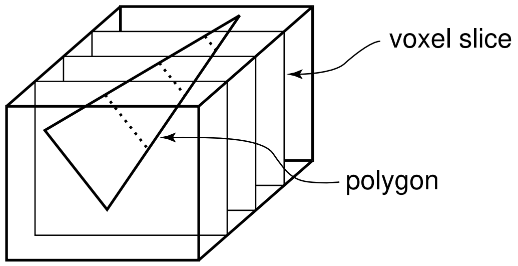

Figure 7.10: Mixtures of volume data and polygons can be rendered using an algorithm that alternates between two phases. During one phase the algorithm renders a slice of voxel data. During the alternate phase the algorithm incrementally scan-converts the portion of each polygon lying between the current voxel slice and the next.

table (adding newly active edges from the edge table and deleting completed edges) and then painting the pixels between pairs of active edges from the same polygon. To compute the intersection of each edge with the current scanline the algorithm uses the intersection with the previous scanline and the slope of the edge, both of which are stored with the edge. Visibility can be computed using the z-buffer algorithm or the painter's algorithm.

The algorithm for rendering mixed data sets processes the volume slice-by-slice, as in the standard shear-warp algorithm. Each slice of voxels represents a slab of the volume that intersects a subset of the polygons. The rendering algorithm first shades and resamples the voxel slice, then scan converts the intersecting portions of the polygons, and finally composites the resampled voxels and the scan-converted polygons into the intermediate image. The polygons are scan converted using a scanline algorithm with two modifications: the algorithm uses an additional active edge table to sort the polygons by the first intersecting voxel slice, and the endpoints of a horizontal pixel span for a particular polygon may be bounded by either the polygon edges or by the intersections of the polygon with the current voxel slab.

Two important issues must be addressed to produce high-quality images. First, the polygon scan conversion algorithm must be designed to avoid aliasing artifacts. The fixed sampling rate in the intermediate image may not be high enough to avoid aliasing, but the problem can be reduced by supersampling the polygons during scan conversion.

Second, visibility must be correctly computed when a polygon intersects a semi-transparent voxel. One approximate solution is to divide the voxel into pieces based on the location of the intersection and then to composite contributions from the pieces and from the polygon into the image in front-to-back order [Levoy 1990b].

We expect this algorithm could achieve rendering times close to the rendering time of the standard shear-warp algorithm plus the time to render the polygons with a standard scanline algorithm implemented in software.

7.5 Clipping Planes

Clipping planes are commonly used in volume visualization applications to provide cutaway views of interior structures in a data set or to remove extraneous information while focusing on a particular part of a data set. Clipping planes with arbitrary orientations can be added to the shear-warp volume rendering algorithm by using a 3D scan conversion algorithm to convert the clipping planes into bounds for the rendering loops.

The 3D scan conversion algorithm can use data structures similar to the mixed-data-set rendering algorithm: an edge table and active edge tables to keep track of which clipping planes intersect the current voxel scanline. The slopes of the clipping planes can be used to calculate the intersection points incrementally as the algorithm iterates through the voxel scanlines. The intersection points determine the bounds for the loop that iterates over voxels in a voxel scanline.

Voxels that intersect or abut a clipping plane must be treated specially. In particular, to avoid aliasing the clipping algorithm must not cause voxel opacities to drop abruptly to zero. A suitable solution is to attenuate voxels as a function of their distance from the cutting plane using a smooth ramp a few voxels wide. It is also useful to let the user apply a special shading function to voxels near a cutting plane. Levoy et al. [1990] identifies several possibilities, including texture-mapping the original voxel values onto the cutting plane or recomputing voxel gradients after applying the cutting plane.

We have not yet implemented clipping planes. With clipping planes enabled the number of rendered voxels may decrease, resulting in improved rendering performance. The fast rendering algorithm must still traverse all of the voxel runs in the run-length encoded volume regardless of the location of the clipping planes, but as we saw in Chapter 5 this cost accounts for only a small fraction of the overall rendering time.

7.6 Chapter Summary

This chapter has discussed a number of extensions to the shear-warp volume rendering algorithm. These extensions show that the basic shear-warp algorithm can support a wide range of visualization options, including flexible user-defined shaders, fast depth cueing, shadows, mixed data sets and clipping planes. The implementations described in this chapter also show that complex shading and lighting effects can be achieved without sacrificing the fast rendering performance provided by the shear-warp algorithm.