Chapter 4

Three Fast Volume Rendering Algorithms

This chapter describes three new volume rendering algorithms using the shear-warp factorization. They are all based on the object-order algorithm presented in Section 3.1 of the previous chapter: translate each slice of the volume, composite the slices together, and warp the result. In this chapter we add data structures that allow the rendering algorithms to exploit coherence. The first new algorithm is optimized for parallel projections and assumes that the opacity transfer function does not change between renderings. The second algorithm supports perspective projections. Both of these algorithms are suitable for interactively manipulating a previously-classified volume's viewing and shading parameters. Finally, the third algorithm is designed for interactively classifying volumes. It allows the opacity transfer function to be modified as well as the viewing and shading parameters, with a moderate performance penalty.

4.1 Parallel Projection Rendering Algorithm

4.1.1 Overview of the Parallel Projection Algorithm

Property 1 of the shear-warp factorization states that voxel scanlines in the sheared volume are aligned with pixel scanlines in the intermediate image, which means that the volume

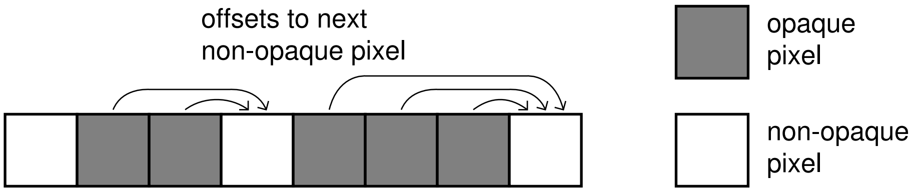

Figure 4.1: Offsets stored with opaque pixels in the intermediate image allow occluded voxels to be skipped efficiently.

and image data structures can be traversed simultaneously in scanline order. Scanline-based coherence data structures such as run-length encoded representations are therefore a natural choice.

The first data structure we use is a precomputed run-length encoding of the voxel scanlines. This data structure allows us to skip over transparent voxels during rendering. A run-length encoding of a scanline consists of a series of runs represented by a run length and a data value for each run [Foley et al. 1990, Ch. 17]. In our data structure the encoded scanlines consist of two types of runs, transparent and non-transparent, defined by a user-specified opacity threshold. We build this data structure during a preprocessing step that scans through the original volume, classifies each voxel (i.e. assigns an opacity), compares the voxel opacities to the threshold, and outputs the run-length encoded representation.

During rendering we use the run-length encoding to determine which parts of the volume are transparent. The rendering algorithm traverses the encoded volume slice-by-slice, just as in the brute-force shear-warp algorithm described earlier. At the beginning of the data for each voxel slice the algorithm uses the shear matrix to compute the translation for the slice. The algorithm then streams through the encoded runs, translates and resamples only the voxels in the non-transparent runs, and composites the resampled voxels into the intermediate image. Using this algorithm the run-length encoded volume can be decoded, sheared, resampled and composited into an image in a single loop without explicitly constructing the undecoded volume or the sheared volume. Furthermore, the algorithm eliminates work in the transparent portions of the volume.

Second, to take advantage of coherence in the image, we use a run-length encoding of the intermediate image. The algorithm constructs this data structure as it computes the intermediate image during rendering. The run-length encoding consists of an offset stored with

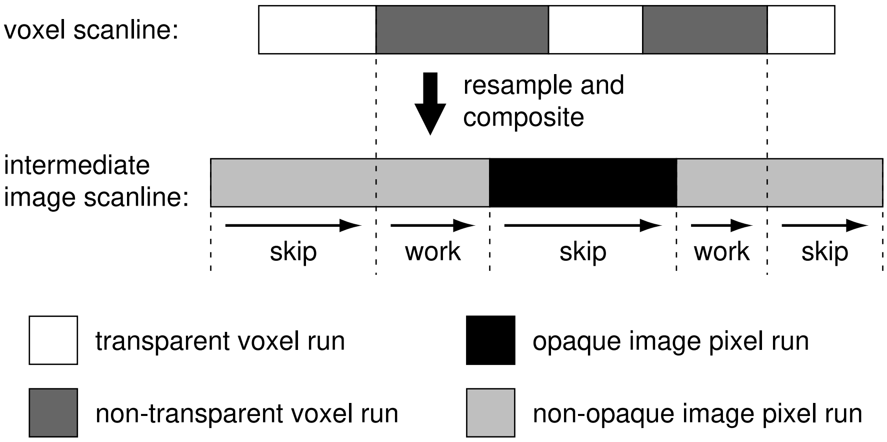

Figure 4.2: Resampling and compositing are performed by streaming through both the voxels and the intermediate image in scanline order, skipping over voxels that are transparent and pixels that are opaque.

each opaque intermediate image pixel (Figure 4.1). The offset points to the next non-opaque pixel in the same scanline. An image pixel is defined to be opaque when its opacity exceeds a user-specified threshold, in which case the corresponding voxels in yet-to-be-processed slices are occluded. The offsets associated with the image pixels are used to skip runs of opaque pixels without examining every pixel. We store an offset with every pixel, not just the first pixel in a run, since it is possible to jump into the middle of a run of opaque pixels after traversing a run of transparent voxels.

These two data structures and Property 1 of the factorization lead to a fast scanline-based rendering algorithm (Figure 4.2). The algorithm streams through both the voxels and the intermediate image in scanline order, using the data structures to avoid work. By marching through the volume and the image simultaneously in scanline order we reduce addressing arithmetic. By using the run-length encoding of the voxel data to skip voxels that are transparent and the run-length encoding of the image to skip voxels that are occluded, we perform work only for voxels that are both non-transparent and visible.

For voxel runs that are not skipped we use a tightly-coded loop that performs shading, resampling and compositing. Properties 2 and 3 of the factorization allow us to simplify the resampling step in this loop since the resampling weights are the same for every voxel in a slice. We use a shading system based on lookup tables to compute the color of each voxel, as described in Chapter 7.

After the volume has been composited the intermediate image must be warped into the final image. Since the 2D image is smaller than the volume this part of the computation is relatively inexpensive.

4.1.2 The Run-Length Encoded Volume

The run-length encoded volume must be constructed in a preprocessing step before rendering. The preprocessing algorithm traverses the volume in storage order, computes the opacity of each voxel, and then compares each voxel's opacity to a threshold opacity to determine if it is transparent or non-transparent. The algorithm uses the threshold only to determine which voxels have negligible opacity; the unthresholded opacities are retained for rendering. As the preprocessing algorithm classifies voxels it collects them into runs. A run consists of a sequence of contiguous voxels that are all transparent or all non-transparent. The algorithm stores the lengths of the runs along with the data for the non-transparent voxels. The transparent voxels are discarded.

The data structure for the encoded volume consists of three arrays (Figure 4.3): an array of run lengths, an array of non-transparent voxels, and an array of pointers into the first two arrays that is used to find the beginning of each slice of the original volume. In the run length array every other entry contains the length of a run of transparent voxels. The remaining entries contain the lengths of the intervening runs of non-transparent voxels. Each run length is represented as an 8-bit byte. If the length of a run exceeds the maximum representable length (255 in the current implementation) then the run is split into several runs interspersed with zero-length runs of the opposite type. Runs are also split at the end of a voxel scanline to ensure that every voxel scanline begins with a new run.

The voxel array simply contains all of the non-transparent voxels, packed contiguously. The transparent voxels are not stored since they are not needed during rendering and memory consumption can be reduced by omitting them. The run length array makes it possible to determine where transparent voxels must be inserted in the compressed voxel array to reconstruct the volume.

Each voxel in the voxel array contains an opacity and any data required for shading. The

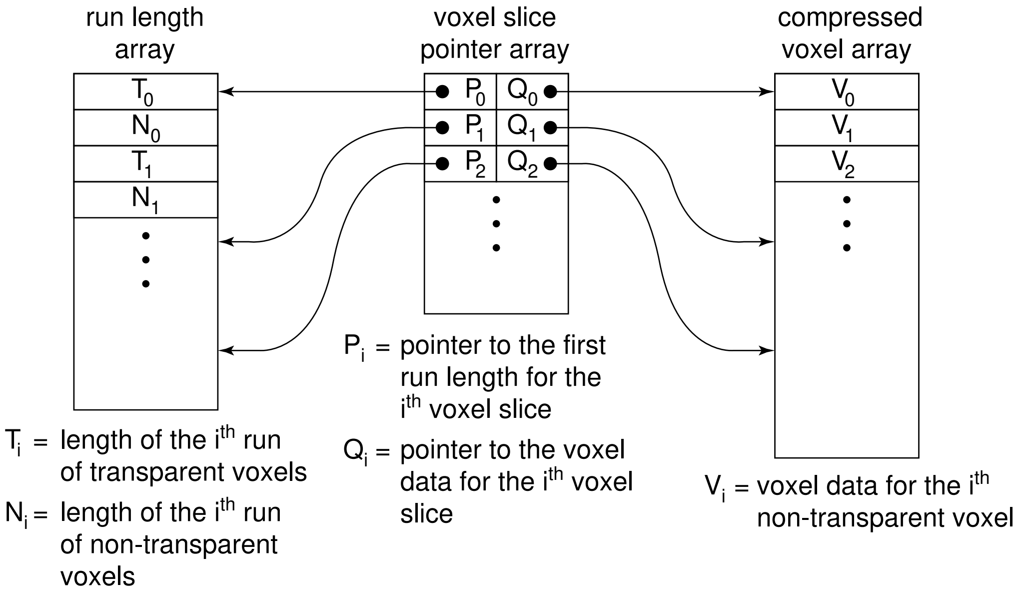

Figure 4.3: The run-length encoded volume consists of three data structures: an array of run lengths (left), an array of pointers that provides random access to voxel slices (middle), and an array containing the non-transparent voxels (right).

shading data will be used by the rendering algorithm to compute the color of the voxel. The color cannot be precomputed since it may depend on the viewpoint and the location of the light sources, both of which might change from frame to frame in a rendering sequence. In the current implementation the opacity data is stored in one byte and the shading data is user-configurable. Typically three bytes are used for shading data (two bytes for a precomputed surface normal and one byte for the original scalar value), for a total voxel size of four bytes per voxel.

The run lengths and the voxel data are stored in separate arrays to avoid byte alignment problems. On some processor architectures particular data types must be aligned to a particular byte boundary in memory. If single-byte run lengths were interspersed with voxels containing larger data types then additional padding bytes would be required.

One difficulty introduced by run-length encoded representations is that random access to the encoded data is not possible. The voxels must be decoded from the beginning of the data structure. The third array in our representation, which contains pointers to the other two arrays, solves this problem. Each entry in the third array points to the first run length and the first non-transparent voxel for a given slice of the volume, allowing random access to any voxel slice. Random access to individual scanlines, which will be necessary for the parallel rendering algorithm described in Chapter 6, can be supported by including one entry per voxel scanline in this array.

A second difficulty with a run-length encoded volume representation is that the volume can only be traversed in the order that the voxels are encoded. The rendering algorithm may need to use a transposed version of the volume data, depending on the principal axis of the viewing transformation. To solve this problem we precompute three run-length encodings, one for each of the three principal viewing axes. The rendering algorithm chooses the appropriate copy depending on the viewing transformation. It is best to load all three copies into memory simultaneously to avoid page faults when the viewing direction switches to a new principal axis, but maintaining three copies in memory does not impact cache performance since only one copy is needed to render each frame. Furthermore, because transparent voxels are not stored the total size of the encoded volume is typically smaller than the original volume, even with three-fold redundancy.

The combination of the three transposes of the run-length encoded volume form a view-independent spatial data structure that can be precomputed before rendering, provided that the classification function is known. Section 4.3 later in this chapter describes an alternative data structure that does not require specifying the classification function.

4.1.3 The Run-Length Encoded Image

The intermediate image is also a run-length encoded data structure: it encodes runs of opaque and non-opaque pixels. However it has different requirements than the run-length encoded volume data structure since the volume can be precomputed whereas the intermediate image must be modified during rendering. We therefore need a data structure that supports dynamic creation of opaque pixel runs and merging of adjacent runs, as well as a fast method to find the end of a run.

This problem is equivalent to the UNION-FIND problem (also called the disjoint-set union problem) [Aho et al. 1974, Cormen et al. 1990]. The problem involves a collection of objects grouped into disjoint sets. In our case the objects are opaque pixels and the sets

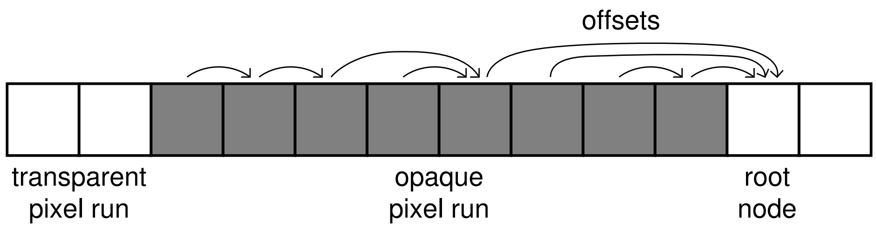

Figure 4.4: A tree data structure to represent an opaque pixel run: each opaque pixel has an offset that points to another pixel in the run or to the end of the run.

are runs of opaque pixels. Each set contains a single "representative" that serves as the name of the set. We will always define the last pixel in a run to be the representative. Three operations are defined: (1) CREATE, which creates a new set containing a single element, (2) UNION, which merges two sets into a new set with a single representative, and (3) FIND, which finds the representative associated with a set, given any element in the set.

A common data structure for representing the sets is a forest of trees. In our algorithm we represent each opaque pixel run by a tree (Figure 4.4). We implement the trees using offsets stored with each pixel in the intermediate image. The intermediate image data structure is a 2D array of pixels. Each pixel contains a color, an opacity, and a relative offset. In the current implementation the color and opacity are stored in single-precision floating point format. The relative offset is a one-byte value containing the number of pixels to skip to reach the pixel's parent node. The root of each tree is the non-opaque pixel at the end of the opaque pixel run (although the non-opaque pixel is not actually part of the run). Each non-opaque pixel has an offset equal to zero.

The rendering algorithm uses the intermediate image data structure as follows. The algorithm initializes the intermediate image by making all of the pixel opacities transparent and all of the offsets zero. During rendering, before the algorithm composites a voxel into a pixel it checks the value of the pixel's offset. If the offset is non-zero then the pixel is opaque, so the algorithm performs a FIND operation to find the end of the run and then skips to it. On the other hand, if the offset is zero then the algorithm completes the compositing operation. After the compositing operation, if the new opacity exceeds the maximum opacity threshold then the algorithm performs a CREATE operation to create a new opaque pixel run, and possibly a UNION operation to merge the run with adjacent runs.

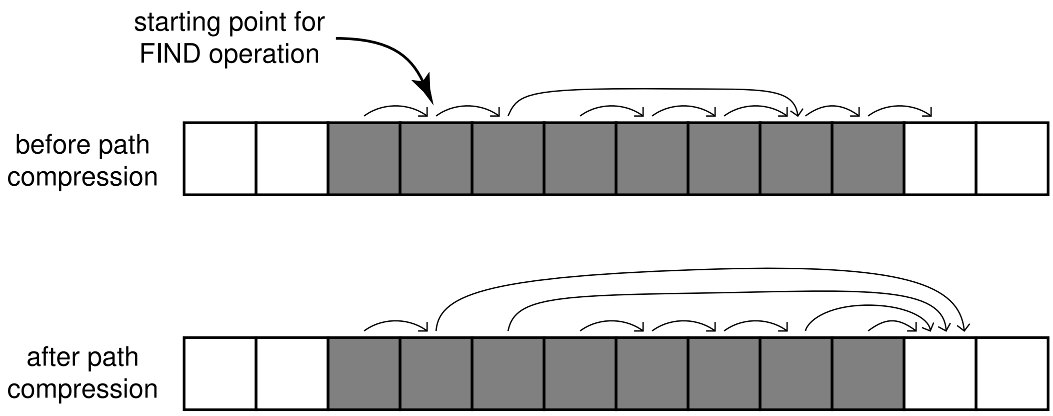

Figure 4.5: After the rendering algorithm traverses a chain of offsets to find the end of an opaque pixel run (top), it traverses the same chain a second time and updates each offset to point to the end of the run (bottom). This optimization is called path compression.

The CREATE operation has a simple implementation: to create a new run when a pixel becomes opaque the algorithm just sets the pixel's offset to one. Furthermore, the UNION operation is free. If there is already a run of opaque pixels to the left of the new run then its final offset already points into the new run. Similarly, the offset in the new run points to the next pixel; if that pixel is opaque then the new run automatically becomes part of the following run.

The algorithm uses the FIND operation to find the end of a run starting from any pixel within the run. This operation can be accomplished by following the chain of offsets up to the first non-opaque pixel, which is the first pixel with an offset of zero. However this implementation is inefficient because the CREATE and UNION operations only produce offsets equal to one, resulting in long chains. There is a very effective optimization called *path compression* that greatly reduces the average length of the chains [Cormen et al. 1990]. The optimization can be implemented as follows: after the algorithm traverses a chain of offsets to find the end of a run, it traverses the same chain a second time and updates each of the offsets to point to the end of the run (Figure 4.5). This optimization is effective because it shortens paths that are later reused.

Without path compression the asymptotic complexity of updating and traversing the intermediate image is $O(m^2)$ operations per scanline, where m is the total number of calls to CREATE, UNION and FIND. However, with path compression the complexity drops to

$O(n_f \log_{(1+n_f/n_c)} n_c)$ where n_c is the number of CREATE operations and n_f is the number of FIND operations [Cormen et al. 1990]. Furthermore, our performance measurements in Chapter 5 show that in a real implementation the cost of traversing opaque pixel runs drops to a negligible fraction of overall execution time.

An additional optimization, called *union by rank*, can be used to further reduce the asymptotic complexity (although for common problem sizes the real execution time is slightly worse). This optimization applies to the UNION operation: when two sets are merged the smaller set should be made to point to the larger set. The size (or rank) of the set is defined to be the number of levels in the tree representing the set. This optimization tends to keep the trees balanced, thereby reducing the maximum number of steps from a leaf node to the root. To add the union-by-rank optimization to our algorithm the rank of each tree must be stored in the pixel located at the root of the tree and the implementation of UNION must be changed.

The combination of path compression and union-by-rank results in an asymptotic complexity of $O(m\,\alpha(m, n_c))$ where $\alpha(m, n_c)$ is the inverse of Ackermann's function—a function that grows so slowly it equals at most four for all reasonable problem sizes. However, the additional cost of the UNION operation results in a higher constant in the cost polynomial for the algorithm. With both optimizations our overall rendering algorithm is 5-15% slower than with path compression alone. Furthermore, for volumes with 256 or 512 voxels per side, a $\log$ term is no worse than $\alpha(m, n_c)$ in the complexity expression: both grow extremely slowly for practical problem sizes. We therefore use only path compression in our current implementation.

The dynamic screen data structure proposed by Reynolds et al. [1987] (described in Section 2.2.2) has the same function as our array of opaque pixel links but uses linked lists with explicit pointers to represent runs of image pixels. In their implementation each image scanline consists of a linked list of opaque and non-opaque runs, initially containing a single non-opaque run. Their rendering algorithm simultaneously scans the linked lists for a run-length encoded voxel scanline and the corresponding image scanline to avoid compositing transparent or occluded voxels, just as in our algorithm.

However, the linked list must be accessed sequentially unlike our array of offsets which can be indexed in random order. As a result their algorithm must traverse every run in an image scanline even if most of the voxels projecting onto the scanline are transparent. Furthermore, updating the linked list requires using a dynamic memory allocator. The cost of maintaining a pool of free memory blocks adds overhead, and the linked lists are not likely to have good spatial locality. Thus their algorithm may have more computational and memory overhead than ours.

Our algorithm does have somewhat more overhead than early ray termination in a ray caster since our algorithm must traverse the run-length encoded data structures even in occluded regions of the volume. Furthermore, the effectiveness of the opaque pixel links depends on coherence in the opaque regions of the image. The runs of opaque pixels are typically large, so many pixels can be skipped at once and the number of pixels that are individually examined is relatively small. In Chapter 5 we present quantitative results demonstrating that our algorithm compares favorably to a ray caster with early ray termination.

4.1.4 Resampling the Volume

The rendering algorithm must resample each voxel slice when it translates the slice into sheared object space. The current implementation uses a bilinear interpolation filter, so two scanlines of voxel data are required to produce one resampled voxel scanline. The rendering algorithm uses a gather-type convolution (backward projection), which requires that the algorithm traverse and decode the two input scanlines simultaneously. As a result, each voxel scanline must be traversed twice (once for each of the resampled scanlines to which it contributes).

An alternative implementation is to use scatter-type convolution (forward projection): the algorithm traverses each input voxel scanline once and splats its contribution into two resampled voxel scanlines. However the partially-computed resampled voxels must be stored in a temporary buffer until they have been completely computed to avoid reordering the compositing and reconstruction filtering steps. Moreover, the temporary buffer must be run-length encoded to retain the benefit of the other coherence data structures.

During resampling the rendering algorithm skips over regions where both input scanlines contain transparent runs. Furthermore, the algorithm uses the opaque pixel links to skip over occluded voxels, so only voxels that are non-transparent and non-occluded are actually resampled. The cost of resampling can be further reduced since the resampling weights are the same for every voxel in the slice, allowing the algorithm to compute the weights only once per slice.

We will discuss filter quality issues in Section 5.1.2.

4.1.5 Warping the Intermediate Image

The image computed by the compositing phase of the algorithm is an oblique projection of the volume. In general, an arbitrary affine warp must be applied to the intermediate image to produce the correct final image. The second factor of the shear-warp factorization determines the necessary transformation (Section 3.2).

Implementation of the 2D image warp is straightforward (see the surveys by Heckbert [1986] and Wolberg [1990]). The current implementation uses a one-pass gather (backward projection) algorithm with a bilinear filter.

4.1.6 Opacity Correction

In our algorithm the resolution of the volume determines the sampling rate that is used to evaluate the volumetric compositing equation (Equation 1.4). As a result the distance between samples along a viewing ray is constant in object space, but varies in image space depending on the current view transformation. This fact leads to a problem. Recall from Section 1.1.2 that the opacity at a sample point is defined as:

$$\alpha_i = 1 - e^{-\phi_i \Delta x}$$

where Δx is the sample spacing in image space. Since the sample spacing changes with viewpoint so must the opacities, and thus the precomputed opacities stored in each voxel must be corrected for a particular viewing direction.

Figure 4.6 illustrates the problem. Consider a cube-shaped volume filled with identical low-opacity voxels. If a viewer looks straight at the center of one face of the volume the distance between sample points in image space along the viewing ray equals the length of one side of a voxel. If the viewer rotates the volume 45 degrees around its center axis then

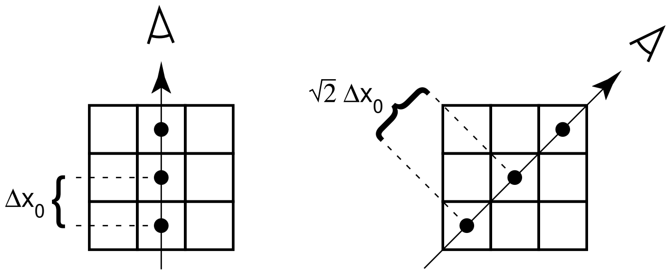

Figure 4.6: The spacing between samples on a viewing ray depends on the viewing direction, so voxel opacities must be corrected to account for the sample spacing.

the number of samples stays the same but the distance between sample points in image space increases (unlike with most ray casters, which maintain the same sample spacing regardless of the viewing direction). The attenuation along the ray should also increase, hence the need for opacity correction. Other object-order volume rendering algorithms must also correct opacities for the viewing angle [Laur & Hanrahan 1991].

Let Δx_0 be the width of a voxel and assume that the opacity computed by the opacity transfer function and stored with a voxel (α_{stored}) uses Δx_0 as the sample spacing:

$$\alpha_{\text{stored}} = 1 - e^{-\phi \Delta x_0}$$

Then for some other sample spacing Δx the corrected opacity can be computed as follows:

$$\begin{aligned} \alpha_{\text{corrected}} &= 1 - e^{-\phi \Delta x} \\ &= 1 - \left[e^{-\phi \Delta x_0}\right]^{\frac{\Delta x}{\Delta x_0}} \\ &= 1 - [1 - \alpha_{\text{stored}}]^{\frac{\Delta x}{\Delta x_0}} \end{aligned}$$

The corrected opacity is a function of the stored opacity and the sample spacing ratio $\Delta x / \Delta x_0$, but the function is the same for every voxel. Figure 4.7 shows a plot of the correction function for several values of the sample spacing ratio. Without opacity correction a cube-shaped voxel appears 30% more transparent than it should when viewed from a 45 degree angle, a difference that is visually significant.

The rendering algorithm performs opacity correction on each voxel just before resampling the voxel. The correction can be implemented efficiently with a precomputed lookup

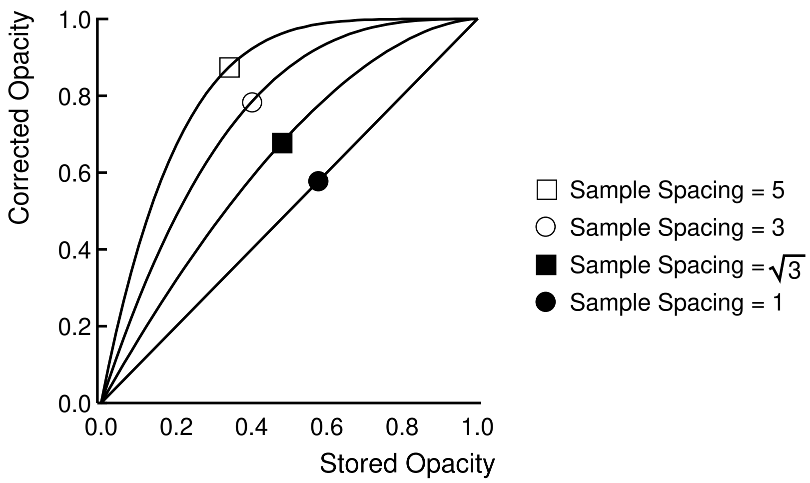

Figure 4.7: Plot of the opacity correction function. Unless the volume has been scaled non-uniformly, the sample spacing lies between 1 and $\sqrt{3}$ (corresponding to viewing angles of 0 and 45 degrees in each direction relative to the voxel slices). Plots for larger sample spacings are shown only to emphasize the shape of the curve.

table that maps uncorrected opacities to corrected opacities. The table must be recomputed whenever the viewpoint changes, but the same table can be used for every voxel.

4.1.7 Implementation of the Parallel Projection Algorithm

Figure 4.8 is a pseudo-code description of the shear-warp volume rendering algorithm for parallel projections. The *RenderVolume* routine is the highest-level routine. It first calls *Factor* to compute the shear and warp factors of the view transformation matrix, and *InitOpacityTable* to precompute the lookup table for opacity correction. The call to *Clear* initializes the intermediate image, *TmpImage*. Then the loop on line 5 composites each slice of the volume into the intermediate image in front-to-back order by calling *CompositeSlice*. Finally, *RenderVolume* warps and displays the image.

CompositeSlice implements the core of the rendering algorithm. It begins by initializing *voxel_ptr*, an array containing pointers to two adjacent scanlines of run-length encoded voxel data. These two scanlines will be filtered to produce one scanline of resampled voxels (aligned with the intermediate image grid). The routine *FindVoxelRuns* uses the voxel slice pointer array in the run-length encoded volume to find the first two voxel scanlines in

```
procedure RenderVolume()
    Factor(ViewMatrix);
    InitOpacityTable();
    Clear(TmpImage);
    for (k = 1 to VolumeSize[Z])
        CompositeSlice(k);
    Image = Warp(TmpImage);
    Display(Image);
end
procedure CompositeSlice(k)
    voxel_ptr[1..2] = FindVoxelRuns(k, 2);
    for (v = 1 to VolumeSize[Y])
        (im_x, im_y) = (XTranslate(k), YTranslate(k) + v);
        while (not EndOfScanline(voxel_ptr))
            if (IsOpaque(TmpImage[im_x][im_y]))
                SkipPixelRun(im_x, im_y, voxel_ptr);
            else
                run_length = min(voxel_ptr[1].run_length, voxel_ptr[2].run_length);
                if (IsTransparent(voxel_ptr[1]) and IsTransparent(voxel_ptr[2]))
                    SkipVoxelRun(im_x, voxel_ptr, run_length);
                else
                    ProcessVoxelRun(im_x, im_y, voxel_ptr, run_length);
                endif
            endif
        end
    end
end
procedure SkipVoxelRun(im_x, voxel_ptr, run_length)
    AdvanceVoxelPtr(voxel_ptr, run_length);
    im_x = im_x + run_length;
end
```

Figure 4.8: Pseudo-code for the parallel projection algorithm *(continued on next page)*.

```
procedure SkipPixelRun(im_x, im_y, voxel_ptr)
    run_length = 0;
    while (TmpImage[im_x + run_length][im_y].offset > 0)
        run_length = run_length + TmpImage[im_x + run_length][im_y].offset;
    end
    path_length = 0;
    while (TmpImage[im_x + path_length][im_y].offset > 0)
        offset = TmpImage[im_x + path_length].offset;
        TmpImage[im_x + path_length].offset = run_length - path_length;
        path_length = path_length + offset;
    end
    AdvanceVoxelPtr(voxel_ptr, run_length);
    im_x = im_x + run_length;
end

procedure ProcessVoxelRun(im_x, im_y, voxel_ptr, run_length)
    for (i = 1 to run_length)
        if (IsOpaque(TmpImage[im_x][im_y]))
            return;
        (color, alpha) = Sample(voxel_ptr);
        CompositeVoxel(im_x, im_y, color, alpha);
        AdvanceVoxelPtr(voxel_ptr, 1);
        im_x = im_x + 1;
    end
end

procedure CompositeVoxel(im_x, im_y, color, alpha)
    if (not IsTransparent(alpha))
        TmpImage[im_x][im_y] = TmpImage[im_x][im_y] over (color, alpha);
        if (IsOpaque(TmpImage[im_x][im_y]))
            TmpImage[im_x][im_y].offset = 1;
    endif
end
```

Figure 4.8 (cont.): Pseudo-code for the parallel projection algorithm.

a particular slice of the volume.

The outer loop iterates over the scanlines of the voxel slice. Line 13 computes the coordinates of the top-left intermediate image pixel that the current pair of voxel scanlines projects onto. Then the inner loop resamples and composites runs of voxel data onto the intermediate-image scanline until it reaches the ends of the voxel scanlines. The body of the loop contains three cases: skip a run of opaque pixels, skip a run of transparent voxels, or do useful work. If the current intermediate image pixel is opaque then the loop calls *SkipPixelRun* to skip over a run of opaque pixels and the corresponding voxels. Otherwise, the algorithm computes *run_length*, the number of voxels until the end of the next voxel run in either of the two voxel scanlines. If both of the current runs are transparent then *CompositeSlice* calls *SkipVoxelRun* to skip over the voxels; otherwise, it calls *ProcessVoxels* to resample and composite the voxels into the intermediate image.

Once the inner loop terminates, the pointers in *voxel_ptr* have each advanced by one voxel scanline and thus they are ready for computing the next intermediate image scanline. The outer loop continues processing scanlines until all of the voxels runs in the slice have been traversed.

SkipVoxelRun is the routine that skips over a run of transparent voxels. It simply calls *AdvanceVoxelPtr* to advance the pointers for the two voxel scanlines and then advances the current position in the intermediate image scanline.

SkipPixelRun is the routine that skips over a run of opaque pixels. The loop on line 34 follows the opaque pixel offsets until the next non-opaque pixel (which must have an offset equal to zero). The loop on line 38 performs path compression as described in Section 4.1.3. Finally, *SkipPixelRun* advances the voxel pointers and the current pixel index (*im_x*) by the length of the opaque pixel run.

ProcessVoxelRun is the routine that resamples and composites a run of voxels. The loop iterates over the sample points in the resampled voxel scanline. First it checks if the sample point projects to an opaque pixel, in which case the beginning of an opaque pixel run has been reached and the routine returns to *CompositeSlice*. Next, *ProcessVoxelRun* computes the value of the volume at a sample point by calling *Sample*. This routine loads four voxels (two from each of the voxel scanlines), performs opacity correction and shading, multiplies the colors and opacities by the precomputed resampling filter weights, and returns the resampled voxel's color and opacity. *Sample* uses the information in the run-length encoded voxel data structure to avoid processing any of the four voxels that may be transparent. Next, the *CompositeVoxel* routine composites the resampled voxel into the intermediate image. Finally, the loop advances to the next voxel sample point.

CompositeVoxel simply uses the "over" operator to composite a voxel into the intermediate image. It only composites a voxel if its resampled opacity exceeds the opacity threshold that was used to create the run-length encoded volume. The procedure also creates a new opaque pixel run if the intermediate image pixel becomes opaque.

The algorithm we have presented in this section uses the shear-warp factorization to take advantage of optimizations from both image-order and object-order algorithms, thereby benefiting from the performance advantages of both. In the next section we will generalize the algorithm to support perspective projections.

4.2 Perspective Projection Rendering Algorithm

Most volume rendering research has focused on parallel projection algorithms. However, perspective projections provide additional cues for resolving depth ambiguities [Novins et al. 1990] and are essential to correctly compute occlusions in such applications as radiation treatment planning, which requires computing an accurate image from the viewpoint of a radiation source [Levoy et al. 1990]. Perspective projections present a problem because the viewing rays diverge so it is difficult to sample the volume correctly. Two types of solutions have been proposed for perspective volume rendering using ray-casters: as the distance along a ray increases the ray can be split into multiple rays [Novins et al. 1990], or each sample point can sample a larger portion of the volume using a mip-map [Levoy & Whitaker 1990, Sakas & Gerth 1991]. A mip-map is a hierarchical data structure that contains prefiltered copies of sampled data (the volume in this case) at several resolutions, so it is possible to compute one sample covering an arbitrarily large portion of the volume in constant time. The application of a mip-map to 3D arrays is a straightforward generalization of the 2D mip-maps introduced for texture filtering applications by Williams [1983]. The object-order splatting algorithm can also handle perspective, but the resampling filter

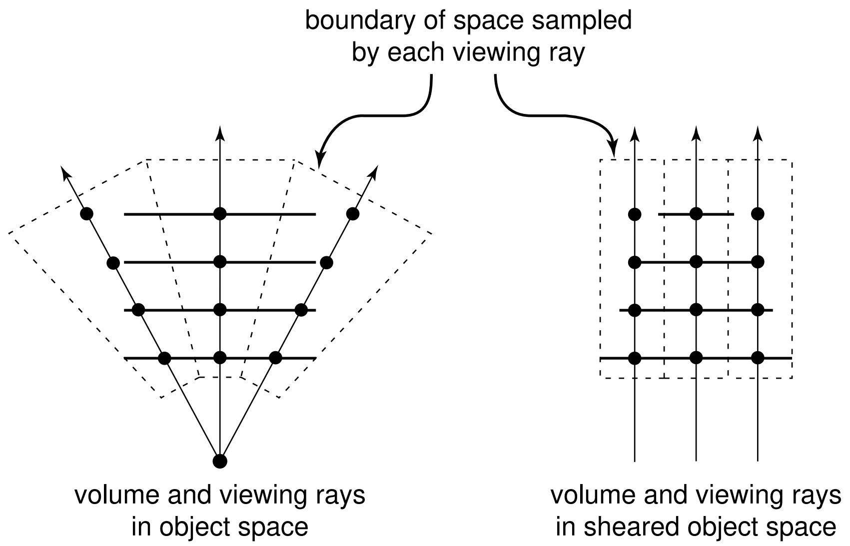

Figure 4.9: Sampling the volume in the perspective projection algorithm: The left diagram shows the sample locations and filter footprint boundaries in object space. The right diagram shows the sample locations in sheared object space.

changes for every voxel [Westover 1990]. This section describes a new algorithm for perspective volume rendering based on the shear-warp factorization.

4.2.1 Overview of the Perspective Projection Algorithm

The shear-warp factorization provides a simple and efficient solution to the sampling problem for perspective projections, as we saw in Section 3.1. The rendering algorithm transforms each slice of the volume to sheared object space by a translation and a uniform scale, and the algorithm then resamples and composites the slices together. These steps are equivalent to a ray-casting algorithm in which rays are cast to uniformly sample the first slice of volume data, and as each ray hits subsequent (more distant) slices a larger portion of the slice is sampled (Figure 4.9). The key point is that within each slice the sampling rate is uniform (Property 2 of the factorization), so the filter is relatively easy to implement.

The perspective algorithm is nearly identical to the parallel projection algorithm. The only substantial difference is that each voxel must be scaled as well as translated during resampling. We always choose a factorization of the viewing transformation in which the slice closest to the viewer is scaled by a factor of one and more distant slices are decimated, so no slice is ever enlarged.

4.2.2 Resampling the Volume

To resample the voxel slices we use a box reconstruction filter and a box low-pass filter, an appropriate combination for both decimation and unity scaling. In the case of unity scaling the two filter widths are identical and their convolution reduces to the bilinear interpolation filter used in the parallel projection algorithm.

The implementation of the resampling loop differs from the parallel projection algorithm because the filter footprint may cover more than two voxel scanlines, so a variable number of voxel scanlines must be traversed to produce a given intermediate scanline. Furthermore, the voxel scanlines cannot be traversed at the same rate as the image scanlines and the resampling weights must be recomputed for each sample point.

4.2.3 Opacity Correction

Opacity correction is more difficult for perspective projections than for parallel projections because not every voxel requires the same correction and it is expensive to compute the corrections individually (an exponentiation would be required for each voxel). However, two observations allow the calculation to be simplified. First, every sample point along a viewing ray requires the same correction since the sample spacing is constant along the ray (Figure 4.9). Second, the sample spacing for adjacent rays is nearly constant. It is therefore possible to use a set of opacity correction lookup tables precomputed for a subset of the rays and to interpolate between tables for the remaining rays.

An algorithm to correct the voxel opacities works as follows. First, we precompute several opacity correction tables using the formula derived in Section 4.1.6 with different values of Δx (the sample spacing along the viewing ray). Each table maps the opacity stored in a voxel to an opacity corrected for the particular value of Δx. We organize the tables as a 2D array indexed by the sample spacing and the uncorrected opacity:

$$\alpha_{\text{corrected}} = \text{OpacityTable}[\Delta x][\alpha_{\text{stored}}]$$

Once the viewing transformation has been fixed but before rendering begins our algorithm calculates Δx for the viewing ray emanating from each intermediate image pixel and stores the value with the pixel. The algorithm computes Δx exactly for a subset of the pixels, and the values for the remaining pixels are interpolated from exact values at nearby pixels. Then during rendering when a voxel is about to be projected onto an intermediate image pixel the voxel sampling routine (*Sample*) uses the value of Δx from the pixel to select a precomputed opacity correction table. Suppose Δx is in between two of the values used for the precomputed tables, Δx_1 and Δx_2. Then the algorithm computes the corrected opacity ($\alpha_{\mathrm{corrected}}$) from the opacity stored with the voxel (α_{stored}) and the opacity correction tables by indexing the two tables with the stored opacity and linearly interpolating the results:

$$
\begin{aligned}
\alpha_1 &= \text{OpacityTable}[\Delta x_1][\alpha_{\mathrm{stored}}] \\
\alpha_2 &= \text{OpacityTable}[\Delta x_2][\alpha_{\mathrm{stored}}] \\
W &= \frac{\Delta x_2 - \Delta x}{\Delta x_2 - \Delta x_1} \\
\alpha_{\mathrm{corrected}} &= \alpha_1 \cdot W + \alpha_2 \cdot (1 - W)
\end{aligned}
$$

This algorithm is fast and allows us to trade off memory (to store additional tables) for accuracy if necessary.

4.2.4 Implementation of the Perspective Projection Algorithm

Figure 4.10 is a pseudo-code description of the shear-warp volume rendering algorithm for perspective projections. The figure only includes routines that differ from the parallel projection algorithm.

The perspective projection algorithm uses a modified version of the *CompositeSlice* procedure that can decimate voxel slices before compositing them. The *voxel_ptr* array contains pointers to all of the voxel scanlines that contribute to a single intermediate image scanline. The maximum number of scanlines that can contribute equals the size of the filter footprint (*footprint*). In the parallel projection algorithm the size of the footprint is always two.

The outer loop of *CompositeSlice* iterates over the intermediate image scanlines onto

```
procedure CompositeSlice(k)
    footprint = 1 + ceiling(1 / ScaleFactor(k));
    voxel_ptr[1..footprint] = FindVoxelRuns(k, footprint);
    for (v = 1 to VolumeSize[Y] × ScaleFactor(k))
        (im_x, im_y) = (XTranslate(k), YTranslate(k) + v);
        while (not EndOfScanline(voxel_ptr))
            if (IsOpaque(TmpImage[im_x][im_y]))
                SkipPixelRun(im_x, im_y, voxel_ptr);
            else
                (color, alpha) = DecimateArea(im_x, im_y, voxel_ptr, footprint);
                CompositeVoxel(im_x, im_y, color, alpha);
                im_x = im_x + 1;
            endif
        end
        AdvanceVoxelPtr(voxel_ptr, (footprint–2) × VolumeSize[X]);
    end
end

procedure DecimateArea(im_x, im_y, voxel_ptr, footprint)
    (color, alpha) = (0, 0);
    for (j = 1 to footprint)
        voxels_left = footprint;
        while (voxels_left > 0)
            run_length = min(voxel_ptr[j].run_length, voxels_left);
            if (IsTransparent(voxel_ptr[j]))
                AdvanceVoxelPtr(voxel_ptr[j], run_length);
            else
                AccumulateVoxelRun(im_x, im_y, voxel_ptr[j], run_length, color, alpha);
            voxels_left = voxels_left – run_length;
        end
    end
    return(color, alpha);
end
```

Figure 4.10: Pseudo-code for the perspective projection algorithm. This figure only includes changes to the parallel projection algorithm in Figure 4.8 *(continued on next page)*.

```
   procedure AccumulateVoxelRun(im_x, im_y, voxel_ptr, run_length, color, alpha)
       for (c = 1 to run_length)
35         (vox_color, vox_alpha) = LoadVoxel(voxel_ptr);
           weight = ComputeWeight(im_x, im_y, voxel_ptr);
           (color, alpha) = (color, alpha) + (vox_color, vox_alpha) × weight;
           AdvanceVoxelPtr(voxel_ptr, 1);
       end
40 end
```

Figure 4.10 (cont.): Pseudo-code for the perspective projection algorithm.

which the voxel slice projects. The inner loop (line 6) iterates over intermediate image pixels until the ends of the voxel scanlines are reached. The algorithm skips runs of opaque pixels using *SkipPixelRun* just like the parallel projection algorithm. When the loop finds a non-opaque pixel it calls *DecimateArea* to filter the voxels that project onto the pixel. It then composites the resampled voxel value into the pixel and proceeds to the next pixel. When the loop terminates, the call to *AdvanceVoxelPtr* advances all of the voxel pointers to the beginnings of the scanlines required for the next intermediate image scanline. If the reciprocal of the scale factor is not an integer then the filter footprint may decrease by one for some iterations of the loops, but the details are not shown here.

DecimateArea loops over the set of voxels that project onto one intermediate image pixel and sums their contributions. The outer loop iterates over voxel scanlines. For each scanline, the inner loop skips over transparent voxel runs and calls *AccumulateVoxelRun* to composite the non-transparent runs. *AccumulateVoxelRun* steps through each voxel in the run, first calling *LoadVoxel* to perform opacity correction and shading, then calling *ComputeWeight* to calculate the filter weight for a particular voxel and image pixel, and finally adding the contribution of the voxel to the color and opacity accumulators.

Compared to the parallel projection algorithm the perspective projection algorithm has somewhat more control overhead and spends more time calculating resampling weights, but nevertheless the algorithm is efficient compared to other methods. The properties of the shear-warp factorization still allow the algorithm to traverse the volume and the intermediate image scanline-by-scanline while performing work only for visible, non-transparent

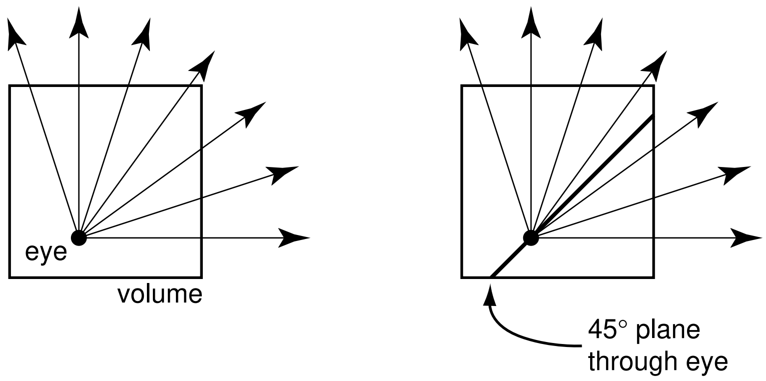

Figure 4.11: For the situation shown on the left there is no single principal viewing axis that is consistent with all viewing rays. The volume must be split along a plane through the eyepoint (the center of projection) at a 45 degree angle to the voxel slices so the subvolumes can be rendered separately, as shown on the right.

voxels, and the resampling filter can be implemented efficiently despite the diverging ray problem.

4.2.5 Limitations of the Perspective Projection Algorithm

A potential problem with the perspective projection algorithm is that for some viewing transformations there is no single principal viewing axis that is consistent with all viewing rays. Figure 4.11 shows an example. While it is possible to use the rendering algorithm even if some of the viewing rays intersect the voxel slices at greater than a 45 degree angle, as the rays approach 90 degrees image quality degrades severely. The problem is most likely to occur during a fly-through in which the viewpoint is inside the volume and the field of view approaches 90 degrees or greater.

The solution to the problem is to subdivide the volume by splitting it along planes oriented at 45 degree angles relative to the slices. At most six subvolumes are necessary to render a field of view covering the entire sphere of possible directions (i.e. the volume can be decomposed into six pyramids with their apexes located at the eyepoint). Each subvolume must be rendered separately with the algorithm described above and then the subimages must be tiled to form the complete image. To prevent discontinuities at the seams between tiles each subvolume should contain a few voxels of overlap with the adjacent subvolumes.

The subimages will then overlap slightly, and the overlapping pixels can be averaged to produce a smooth transition. This modification to the rendering algorithm introduces some complexity and has not yet been implemented.

A second problem with the perspective rendering algorithm is that visibility is not correctly computed. Visibility along each viewing ray must be determined before averaging values from the rays together to compute a pixel value, but our algorithm interchanges these two operations. The response $R(i, j)$ of a pinhole-camera sensor spanning a pixel on the image plane is proportional to the incident radiance L integrated over the solid angle Ω subtended by the pixel:

$$R(i, j) = \int_{\Omega(i,j)} L(i, j, \vec{\omega}) \cos\theta \, d\omega$$

(where θ is the angle between $\vec{\omega}$ and the image plane normal). The radiance in the integrand can be found by evaluating the volume rendering integral:

$$R(i, j) = \int_{\Omega(i,j)} \int_{x} e^{-\int \phi_t(x') dx'} \epsilon(x) \cos\theta \, dx \, d\omega$$

Thus the correct way to compute the response is to first integrate along each viewing ray and then integrate over the solid angle. The perspective shear-warp algorithm filters the voxel slices before compositing them together, which is equivalent to integrating over the solid angle before integrating along the rays:

$$R'(i, j) = \int_{x} e^{-\int \phi_t(x') dx'} \int_{\Omega(i,j)} \epsilon(x) \cos\theta \, d\omega \, dx$$

Thus the order of the integrals has been incorrectly interchanged.

Figure 4.12 illustrates the problem: objects A and B should fully occlude object C. However, the rendering algorithm down-samples the voxel slices before compositing them together. The area of the voxel slice containing object A and its neighborhood are averaged into a single, partially opaque sample. Similarly, object B is averaged into a partially opaque sample. When the two samples are composited together the result is not fully opaque, so object C incorrectly appears partially visible.

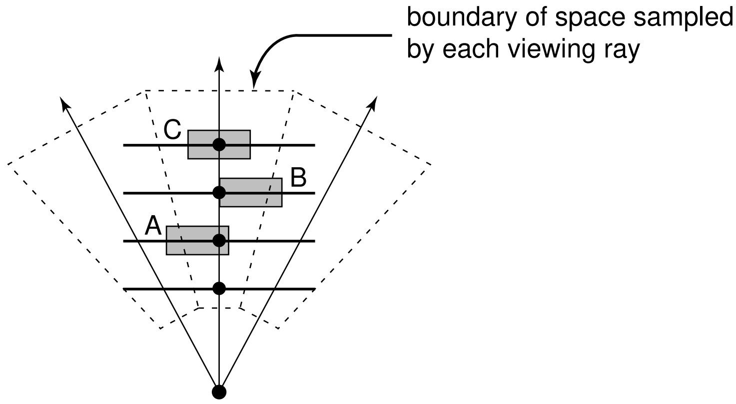

Figure 4.12: Visibility errors can occur in the perspective rendering algorithm because it filters the volume before compositing. In this case object C should be occluded by objects A and B (assuming A and B are fully opaque). However, the rendering algorithm averages objects A and B with the surrounding empty space before compositing the voxel slices together so object C shows through.

This problem can occur in both parallel and perspective renderings, but its likelihood is exacerbated in the perspective case by the divergence of the rays. The problem is fundamental to the shear-warp volume rendering algorithm and any other algorithm that decimates the volume before compositing, such as algorithms employing mip-mapping [Levoy & Whitaker 1990, Sakas & Gerth 1991] and splatting algorithms employing a sheet cache [Westover 1990]. In contrast, the algorithm proposed by Novins et al. [1990] computes visibility more accurately because it casts rays that split into multiple rays as the distance from the viewer increases, so the filter kernel size remains constant along each ray.

In practice the artifacts caused by this problem are small except in cases of very severe perspective distortion.

4.3 Fast Classification Algorithm

The run-length encoded data structure used by the previous two algorithms must be recomputed whenever the opacity transfer function changes. This preprocessing time is insignificant if the user wishes to generate many images from a single classified volume, but if the user wishes to interactively adjust the transfer function then the preprocessing step is unacceptably slow. In this section we present a third variation of the shear-warp algorithm that

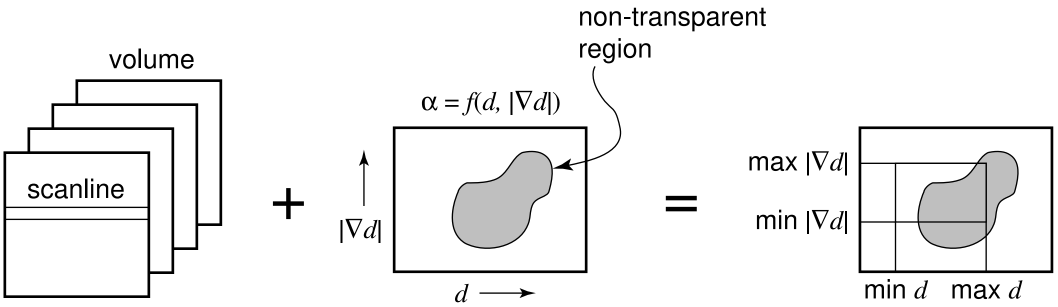

Figure 4.13: Goal of the fast classification algorithm: Given a scanline of voxels, shown on the left, we must determine which voxels are transparent. The classification function $f(d, |\nabla d|)$ shown in the middle of the diagram partitions a two-dimensional feature space into transparent and non-transparent regions. The voxels in the scanline are contained in some bounding box within the feature space, shown on the right. If the bounding box and the non-transparent regions do not intersect then the voxels must be transparent.

evaluates the opacity transfer function during rendering and is only moderately slower than the previous algorithms.

4.3.1 Overview of the Fast Classification Algorithm

A run-length encoding of the volume based upon opacity is not an appropriate data structure when the opacity transfer function is not fixed. Instead we apply the algorithms described in Sections 4.1 and 4.2 to unencoded voxel scanlines, but with a new method to determine which portions of each scanline are non-transparent.

We allow the opacity transfer function to be any scalar function of a multi-dimensional scalar domain:

$$\alpha = f(p, q, ...)$$

For example, the opacity might be a function of the scalar field and its gradient magnitude [Levoy 1988]:

$$\alpha = f(d, |\nabla d|)$$

The function f essentially partitions a multi-dimensional feature space into transparent and non-transparent regions, and our goal is to decide quickly which portions of a given scanline contain voxels in the non-transparent regions of the feature space (Figure 4.13).

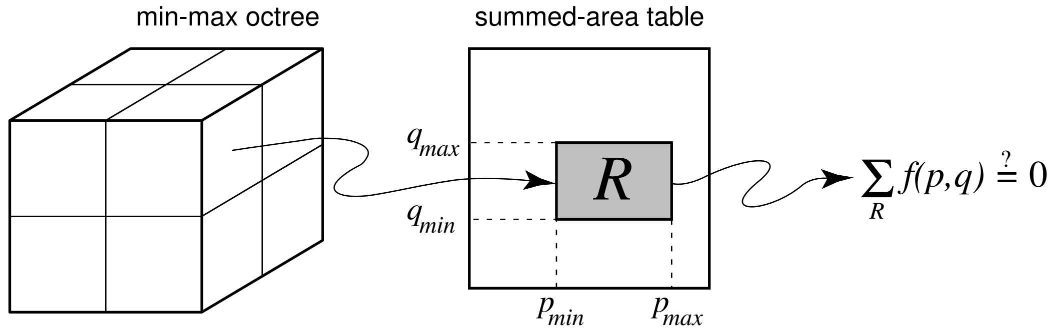

Figure 4.14: Data structures for the fast classification algorithm: A min-max octree is used to determine the range of the parameters p, q of the opacity transfer function $f(p, q)$ in a subcube of the volume. A summed-area table is used to integrate f over that range of p, q. If the integral is zero then the subcube contains only transparent voxels.

We solve this problem with the following recursive algorithm which takes advantage of coherence in both the opacity transfer function and the volume data:

Step 1: For some block of the volume that contains the current voxel scanline, find the extrema of the parameters of the opacity transfer function ($\min(p)$, $\max(p)$, $\min(q)$, $\max(q)$, $\ldots$). These extrema bound a rectangular region of the feature space defined by the domain of f.

Step 2: Determine if the region is transparent, i.e. f evaluated for all parameter points in the region yields only transparent opacities. If so, then discard the scanline since it must be transparent.

Step 3: Subdivide the scanline and repeat this algorithm recursively. If the size of the current scanline portion is below a threshold then render it instead of subdividing.

This algorithm relies on two data structures for efficiency (Figure 4.14). First, Step 1 uses a precomputed min-max octree [Wilhelms & Van Gelder 1990, Wilhelms & Van Gelder 1992]. Each octree node contains the extrema of the parameter values for a subcube of the volume. Second, to implement Step 2 of the algorithm we need to integrate the function f over the region of the feature space found in Step 1. Assume for now that the opacity threshold is zero, so only voxels with opacity equal to zero are transparent. If the integral of f over the region of feature space covered by the subcube is zero then all voxels in the subcube must be transparent. This integration can be performed in constant time using a multi-dimensional summed-area table [Crow 1984, Glassner 1990a]. The voxels themselves are stored in a third data structure, a simple 3D array.

The overall algorithm for rendering unclassified data sets proceeds as follows. When a volume is first acquired or loaded into memory the algorithm precomputes the min-max octree, which is independent of the opacity transfer function and the viewing parameters. Next, just before rendering begins the algorithm uses the opacity transfer function to compute the summed-area table. This computation is inexpensive provided that the domain of the opacity transfer function is not too large. We then use either the parallel projection or the perspective projection rendering algorithm to render voxels from an unencoded 3D voxel array. We traverse the array scanline-by-scanline as usual. But, for each scanline we use the octree and the summed-area table to determine which portions of the scanline are non-transparent. The algorithm classifies voxels in the non-transparent portions of the scanline using a lookup table and then renders them as in the previous algorithms. The algorithm skips opaque regions of the image just as before. Voxels that are either transparent or occluded are never classified, which reduces the amount of computation.

The main advantage of this algorithm is that the min-max octree can be precomputed before fixing the classification function. Along similar lines, two algorithms that use a spatial data structure for generating isosurfaces from volume data have been proposed: Wilhelms & Van Gelder [1992] use a min-max octree, and Stander & Hart [1994] use an octree containing a Lipschitz constant which bounds the variation of the scalar value. Both of these data structures can be precomputed before the user selects an isosurface threshold. However neither algorithm is appropriate for more general classification functions, which is the advantage of our algorithm.

4.3.2 The Min-Max Octree

An octree is a recursive subdivision of space organized as a tree (Figure 4.15). Each node of an octree represents a cubic region of space and has eight children. The children represent a subdivision of the node's space into eight smaller octants. A min-max octree is an octree that

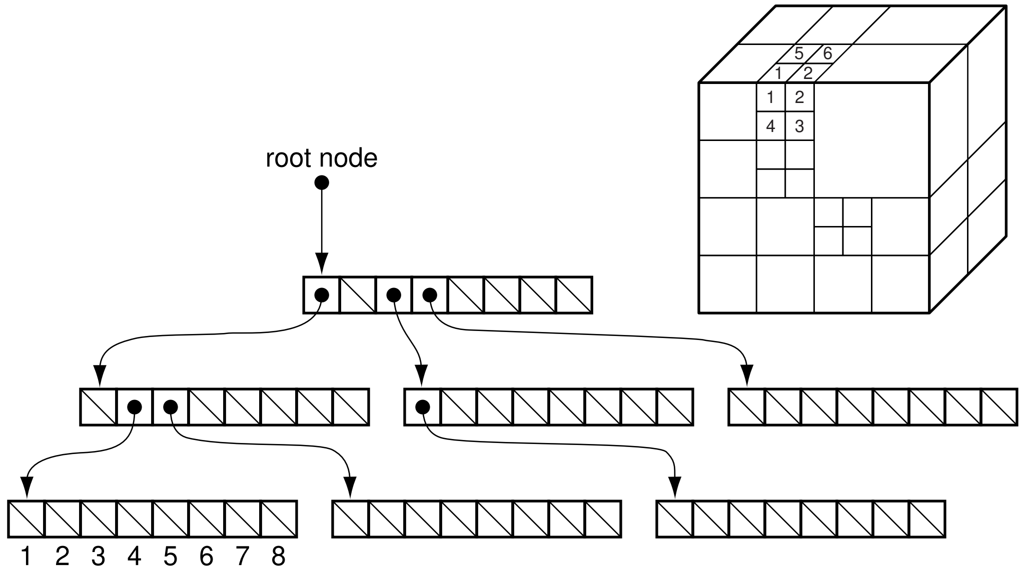

Figure 4.15: Diagram of a min-max octree. Each node contains the minimum and maximum values of each argument to the opacity transfer function over a subcube of the volume.

is used to store the minimum and maximum values of some parameter associated with the region of space represented by each node. The root node contains a minimum and maximum value of the parameter for the entire volume. Each additional level of the tree contains min-max values for smaller regions of the volume, resulting in a more detailed description of parameter variations inside the volume. The smallest octree node size can be chosen to trade off the maximum level of detail versus the amount of time it takes to traverse the octree during rendering. At some point the benefit of increased detail no longer outweighs the time required to traverse the octree nodes while extracting the information.

Each node in the min-max octree contains the following information:

- The minimum and maximum value of each classification parameter over all of the voxels represented by the octree node.

- Storage for caching a result after classifying the octree node. This field indicates whether the voxels represented by the node are all transparent, all non-transparent, or a combination. In the latter case the children of the octree node contain more detailed information.

- A pointer to the children of the octree node. All eight children are stored in a contiguous block of memory so each octree node contains only one pointer that points to the first of its children (as shown in Figure 4.15). Some octree nodes have no children, in which case the pointer is set to NIL. Nodes at the lowest level of the octree contain no pointer field at all (to conserve memory).

The octree does not contain the original volume data. The volume is stored as a separate 3D array of voxels.

An alternative data structure for storing the same information is a forest of min-max binary trees with one tree for each voxel scanline. Binary trees appear to be the natural choice given that we used run-length encoding rather than an octree for the fast rendering algorithm described earlier. We decided against this alternative because binary trees would require eight times as much memory for the same number of levels, and three copies of the data structure would be required (one for each principal viewing axis). The required storage can exceed the size of the volume even in practical cases, whereas the min-max octree is always significantly smaller than the volume.

The algorithm for computing the octree operates as follows. First, the algorithm builds a full octree (an octree in which every node has children except for the nodes in the most detailed level; this data structure is also known as a pyramid). The octree is constructed bottom-up: the algorithm computes the base of the octree by looping over the voxels in the volume to compute the min-max data for the most-detailed octree nodes, and then each node at the next higher level is computed by combining the values from its eight children. This process continues until the root node has been computed.

Second, the algorithm prunes octree nodes for which the difference between the minimum and maximum parameter values falls below a user-defined threshold. For example, if the minimum and maximum values are equal for all parameters in a particular octree node then no additional information is provided by the children of the node, so they can be pruned. Even if the minimum and maximum values are not equal but are nearly so it may be advantageous to remove the children: the more nodes in the tree the longer it takes to traverse the tree during rendering. The user-defined thresholds can be adjusted to optimize the tradeoff between level-of-detail and traversal time.

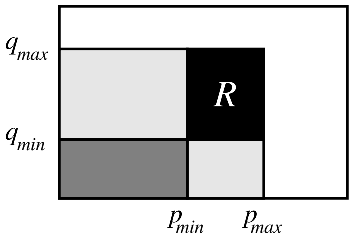

Figure 4.16: Diagram of a 2D summed-area table (after Crow [1984]). The table can be used to integrate a function of two variables over any rectangular region of its domain.

4.3.3 The Summed-Area Table

A summed-area table is a convenient data structure for integrating a discrete function over a rectangular region. It was proposed by Crow [1984] for down-sampling texture maps. Figure 4.16 illustrates an example of a 2D summed-area table used to integrate a two-parameter discrete function. Let $f(p, q)$ be a function defined at discrete lattice points on a rectangular domain. The summed-area table for this function is a 2D array of values $S(p, q)$ computed from f. Each entry in S is the integral of f over a rectangular region with its bottom-left corner at the origin:

$$S(p, q) = \sum_{p'=0}^{p} \sum_{q'=0}^{q} f(p', q')$$

The integral of f over an arbitrary rectangular region can be found from at most four entries in the table. Consider the region R shown in the figure. $S(p_{\max}, q_{\max})$ gives the integral of f over R plus the region to the left of R and below R. The integrals over these two "extra" regions must be subtracted. The integral over the region below R is $S(p_{\max}, q_{\min} - 1)$ and the integral over the region to the left of R is $S(p_{\min} - 1, q_{\max})$. If both of these are subtracted then the integral over the region that is both below and left of R is subtracted twice, so $S(p_{\min} - 1, q_{\min} - 1)$ must be added back. The integral over an arbitrary rectangular region can therefore be computed in constant time using the following formula:

$$\begin{aligned}\sum_{p=p_{\min}}^{p_{\max}} \sum_{q=q_{\min}}^{q_{\max}} f(p, q) \quad &= \quad S(p_{\max}, q_{\max}) - S(p_{\max}, q_{\min} - 1) - \\ &\quad\quad S(p_{\min} - 1, q_{\max}) + S(p_{\min} - 1, q_{\min} - 1)\end{aligned}$$

Similar formulae can be derived for summed-area tables with an arbitrary number of dimensions as described in Glassner [1990a]. An algorithm to compute the table operates as follows: first compute $S(0, 0)$, which is equal to $f(0, 0)$, and then compute $S(p, q)$ from $f(p, q)$ and entries $S(p-1, q)$, $S(p, q-1)$ and $S(p-1, q-1)$.

In the context of the fast classification volume rendering algorithm, the summed-area table is used to integrate the opacity transfer function over the voxels represented by an octree node. The result of the integral indicates whether or not the voxels are transparent. The algorithm operates as follows. When the user selects a new opacity transfer function $f(p, q, \ldots)$ the algorithm computes a transparency function $f_T(p, q, \ldots)$ by evaluating the opacity transfer function for each (p, q) and comparing the result to the minimum opacity threshold α_T:

$$f_T(p, q, \ldots) \equiv \begin{cases} 1 & \text{if } f(p, q, \ldots) > \alpha_T \\ 0 & \text{otherwise} \end{cases}$$

If f_T evaluates to one for a particular point in the parameter space then the function f maps that point to a non-transparent opacity. (Here we have removed the earlier restriction that the opacity threshold must equal zero.)

Next the algorithm computes a summed-area table $S(p, q, \ldots)$ for the transparency function f_T. If there are n parameters to the opacity transfer function then the table must have n dimensions. Each table entry contains the number of parameter tuples in a given region of parameter space that map to a non-transparent opacity. Computing the table requires time proportional to the number of entries in the table, which is the size of the domain of f.

To determine whether or not the voxels represented by an octree node are transparent, the algorithm uses the summed-area table to integrate the transparency function f_T over the voxels in an octree node. The minimum and maximum values of $p, q, \ldots$ stored in the octree node give the bounds of the region of integration. These bounds are used to index the summed-area table. 2^n table lookups are required to compute the integral, where n is the number of parameters to f_T. If the integral equals zero then all of the voxels represented by the octree node must be transparent and need not be rendered. Otherwise, a more detailed level of the octree can be used to refine the estimated classification. When the algorithm reaches the lowest level of the octree it classifies the voxels in the potentially-nontransparent octree nodes (using the original opacity transfer function f) and then renders them.

This algorithm never discards a non-transparent voxel, but it may cause transparent voxels to be rendered. The run-length encoding used in the previous two algorithms exactly represents the subset of the voxels that are transparent whereas the fast classification algorithm computes a conservative approximation. However, if the volume has a high degree of coherence then the approximation will be good enough to achieve a substantial speedup.

The algorithm also takes advantage of coherence in the opacity transfer function. This type of coherence arises because users often choose transfer functions that partition feature space into a small number of transparent and non-transparent regions, e.g. only the voxels within a certain range of scalar values become non-transparent. During rendering, any octree subcube with parameters lying entirely within one region of the partition can be tested for transparency without visiting more detailed levels of the octree. Thus the algorithm can exploit coherence in the opacity transfer function to classify octree nodes with widely-varying parameter values in constant time.

4.3.4 Implementation of the Fast Classification Algorithm

The parallel projection rendering algorithm and the perspective projection rendering algorithm can both be modified to use the data structures described in this section (a min-max octree, a summed-area table, and an unencoded volume array) instead of a run-length encoded volume. In the current implementation each voxel in the volume array contains an explicit value for each parameter to the opacity transfer function as well as any parameters required for shading. Figure 4.17 is a pseudo-code description of the routines that implement the fast classification algorithm. They must be combined with one of the two rendering algorithms described earlier.

The first step during rendering is to compute the summed-area table from the opacity transfer function and to use the table to determine which nodes of the min-max octree represent transparent voxels. The routine *ClassifyOctree* performs these operations. It should be called from the beginning of the *RenderVolume* procedure in either the parallel projection algorithm or the perspective projection algorithm.

ClassifyOctree first calls *ComputeSummedAreaTable* to build a summed-area table from the current classification function, as described in Section 4.3.3. Then *ClassifyOctree* calls

```
procedure ClassifyOctree
    ComputeSummedAreaTable();
    ClassifyOctreeNode(octree_root);
end

procedure ClassifyOctreeNode(node)
    integral = LookupSummedAreaTable(node.minima, node.maxima);
    if (integral = 0)
        node.status = TRANSPARENT;
    else if (node.children = NIL)
        node.status = NON_TRANSPARENT;
    else
        node.status = SUBDIVIDE;
        for (i = 1 to 8)
            ClassifyOctreeNode(node.children[i]);
    endif
end

procedure ComputeVoxelRuns(node, node_size, j, k)
    CurrentRunType = NIL;
    CurrentRunIndex = 0;
    if (node.status = SUBDIVIDE)
        child = 0;
        if ((j mod node_size) > node_size/2)
            child = child + NEXT_J;
        if ((k mod node_size) > node_size/2)
            child = child + NEXT_K;
        ComputeVoxelRuns(node.children[child], node_size/2, j, k);
        ComputeVoxelRuns(node.children[child + NEXT_I], node_size/2, j, k);
    else
        if (CurrentRunType = node.status)
            RunLengths[CurrentRunIndex] = RunLengths[CurrentRunIndex] + node_size;
        else
            CurrentRunIndex = CurrentRunIndex + 1;
            RunLengths[CurrentRunIndex] = node_size;
            CurrentRunType = node.status;
        endif
    endif
    return(RunLengths);
end
```

Figure 4.17: Pseudo-code for the fast classification algorithm.

ClassifyOctreeNode which performs a preorder depth-first traversal of the min-max octree. At each node during the traversal *ClassifyOctreeNode* calls *LookupSummedAreaTable* to integrate the transparency function (derived from the opacity transfer function) as described in Section 4.3.3. If the integral equals zero then the routine marks the node as transparent. If the integral is nonzero and the node has no children then the routine marks the node as potentially non-transparent, meaning that the node will be considered by the rendering routine. The remaining case is that the integral is nonzero but the node is not a leaf node, so *ClassifyOctreeNode* recursively visits the eight children. After the recursion unwinds, the *status* fields in the octree nodes indicate which nodes represent transparent voxels.

The last routine in the fast classification algorithm is *ComputeVoxelRuns* which uses the octree to compute a list of transparent and non-transparent runs for any voxel scanline. The list of runs is stored in an array just like the *RunLengths* array in the run-length encoded volume (Figure 4.3). The rendering routines call *ComputeVoxelRuns* to compute the run lengths on-the-fly and then they use the array of run lengths and a scanline of voxels from the unencoded voxel array instead of the precomputed run-length encoded volume. Otherwise, the rendering algorithms do not change. The parallel projection algorithm in Figure 4.8 can be modified to use the fast classification algorithm by removing the call to *FindVoxelRuns* on line 11 and inserting two calls to *ComputeVoxelRuns* just before line 14 (one for each voxel scanline). The perspective projection algorithm in Figure 4.10 can be modified by removing the call to *FindVoxelRuns* on line 3 and inserting calls to *ComputeVoxelRuns* for each voxel scanline just before line 6. The call to *AdvanceVoxelPtr* on line 15 must also be removed.

ComputeVoxelRuns operates as follows. It is a recursive routine that must be called from *CompositeSlice* with the root node of the octree, the number of voxels per side in the root node, and the coordinates of the current voxel scanline (k is the slice number and j is the scanline number). *ComputeVoxelRuns* first examines the *status* field of the current node. This field contains the result computed by *ClassifyOctree*. If the field indicates that the octree node should be subdivided then *ComputeVoxelRuns* computes which two of the eight children of the node intersect the scanline. Since scanlines are always aligned with an axis of the volume, exactly two of the children must intersect and the arithmetic to determine which two is simpler than in a ray caster (lines 21–25). Once the two children have been determined, *ComputeVoxelRuns* calls itself recursively on each one.

When *ComputeVoxelRuns* finds a node that should not be subdivided it adds a run to the array of run lengths for the current scanline. If the type of the current run and the node match (i.e. if both are transparent or both are non-transparent) then the length of the current run is extended by the size of one side of the octree node (line 30). Otherwise *ComputeVoxelRuns* creates a new run.

When the recursion unwinds the *RunLengths* array contains a list of run lengths indicating which voxels are transparent and which are potentially non-transparent. When the rendering loop in *CompositeSlice* encounters a potentially non-transparent voxel run it must retrieve the voxels from the unencoded voxel array and classify them individually just before they are resampled and composited.

Adjacent voxel scanlines often intersect the same set of octree nodes, so the array of run lengths can sometimes be reused for several adjacent scanlines. If the size of the smallest octree node visited by *ComputeVoxelRuns* is $n_{\min}$ then the next $n_{\min}$ voxel scanlines can reuse the same array of run lengths. *CompositeSlice* keeps track of when to call *ComputeVoxelRuns* and when to reuse the existing run lengths.

4.3.5 Limitations of the Fast Classification Algorithm

The fast classification algorithm places two restrictions on the opacity transfer function. First, it must be possible to precompute the parameters to the classification function for each voxel. The function itself need not be specified in advance, but the parameters must be fixed so that the min-max octree can be precomputed. For example, a transfer function that maps the scalar value and the gradient magnitude of the scalar value into an opacity is acceptable because both the scalar value and the gradient magnitude can be precomputed and stored with the voxel. An example of a classification function that does not satisfy this property is a segmentation algorithm based on a 3D flood fill. In order to compute the opacity of a particular voxel, a flood fill function must examine the values of many neighboring voxels. Thus the "parameters" of the function include a non-precomputable subset of the voxels in the volume.

The second restriction is that the cardinality of the domain of the opacity transfer function must not be too large. The cardinality of the domain depends on the number of parameters of the opacity transfer function and the number of possible discrete values for each parameter. The summed-area table must have one entry for every parameter tuple, so if the domain is too large then the table will take excessive time to compute and will occupy a large amount of memory. Classification functions with more than two or three parameters are not likely to be feasible.

Compared to the algorithms based on run-length encoding, the fast-classification algorithm has one primary disadvantage: it requires more memory, which can have an impact on the maximum feasible volume size and on rendering time. The fast classification algorithm uses an uncompressed 3D array of voxels, and each voxel contains precomputed values for each classification parameter and each shading parameter. For example, in the current implementation each voxel contains the scalar value (one byte), the gradient magnitude of the scalar value (one byte), and a precomputed surface normal vector for shading (two bytes). Thus a volume containing 512^3 voxels occupies 512 Mbytes of memory, too large to fit into main memory on most workstations. A run-length encoded classified volume often occupies much less space, so the previous two rendering algorithms are practical for larger volumes.

Furthermore, because of the size of the voxel array, it is not practical to build three transposed copies of the array. As a result, if the major viewing axis changes then the volume data must be accessed against the memory storage stride and therefore performance degrades. Other volume rendering algorithms are also prone to this problem as we will see in the next chapter. The performance degradation may be acceptably small depending on the speed of the memory system. If memory performance is a problem it can be combated by transposing the 3D array of voxels, but doing so results in a delay during interactive rotation sequences. It is better to use a small range of viewpoints that does not require transposing the volume while modifying the classification function, and then to switch to the run-length encoded representation for rendering with arbitrary viewpoints. The next section describes an efficient algorithm for switching between the two representations.

Another solution is to first eliminate portions of the volume that will never be of interest by classifying the volume with a conservative opacity transfer function. This crudely-classified volume can be stored using the run-length encoded representation. The run-length encoded volume can then be used in conjunction with the min-max octree and the fast classification algorithm. For example, it is straightforward to choose a transfer function that maps air in a CT scan to transparent opacity, eliminating a large fraction of the volume in many medical data sets. The user could then use the fast classification algorithm in conjunction with a run-length encoded volume to refine the classification function. This hybrid algorithm is a potential area for future research.

4.4 Switching Between Modes

We have built a demonstration application based on the algorithms in this thesis that has two modes, one for classification and one for viewing and shading (Figure 4.18). The classification mode uses the min-max octree and summed-area table, and the viewing mode uses the run-length encoded volume. These two modes address different parts of the visualization process outlined in Section 1.1.4.

However, users of the application will want to switch between the two modes at will. Since the min-max octree is independent of all user-selectable classification and rendering options it can be precomputed when the user loads a volume and therefore the data required for the fast classification mode is always available. But when the user wishes to switch from the classification mode to the viewing mode the run-length encoded volume must be recomputed for the current classification function. A fast algorithm to create the data structure is necessary so that the user can switch modes quickly.

The data structures and pseudo-code fragments developed in the previous section can be used to construct such an algorithm. A brute-force algorithm to compute a run-length encoded volume operates by streaming through the unencoded volume data in scanline order, classifying each voxel and collecting the classified voxels into runs. To optimize this algorithm we use the min-max octree and the summed-area table to determine which voxels in a voxel scanline must be transparent before traversing the scanline. Then the transparent runs can be skipped without classifying individual voxels. The routine *ComputeVoxelRuns* from Section 4.3 computes exactly the required information to enable the run-length encoding algorithm (Section 4.1.2) to skip transparent regions of the volume.

One way the rendering algorithms in this chapter can be integrated under a unified user

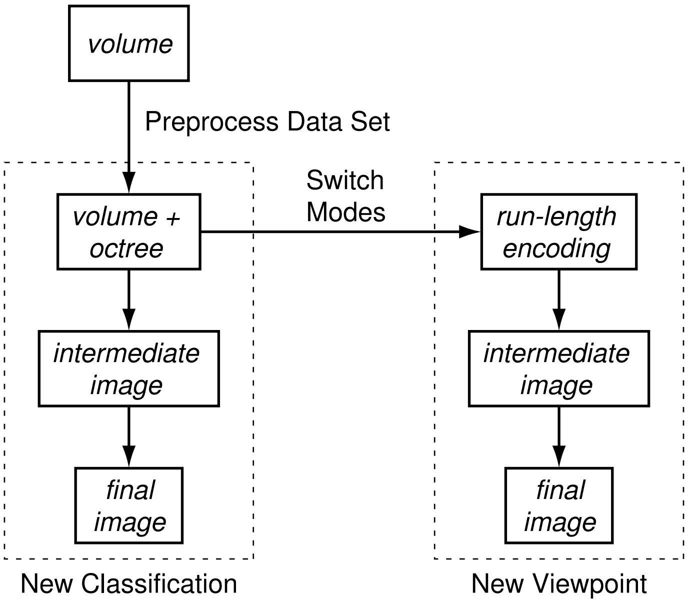

Figure 4.18: A family of volume rendering algorithms: When the user loads a new volume we preprocess the data to produce a min-max octree. The user can then interactively adjust classification parameters using the fast classification algorithm (left side). When the user settles on a classification function we use the min-max octree to compute a run-length encoded volume and the user can interactively adjust the viewing and shading parameters (right side).

interface is to automatically choose the appropriate rendering mode based on the rendering parameters the user adjusts. When the user adjusts a classification parameter the application uses the fast classification algorithm to render the volume and then starts a background process to compute the corresponding run-length encoded volume. If the user continues to adjust the classification parameters then the application interrupts the background process, rerenders the volume using the fast classification algorithm, and restarts the background process with the new parameters. If the user decides to adjust the viewing or shading parameters, the application pauses momentarily until the background process finishes computing the run-length encoded volume and then the application switches to viewing mode.

For very large volumes the fast classification mode is less efficient and the rendering time may be unacceptably slow. A solution is to use the fast classification algorithm to render a small subcube of the volume while using the most-recently computed run-length encoding to render the rest of the volume. The user can then select a subcube of the volume (say, 128^3 voxels) to focus on during classification, but the rest of the volume would still be visible to provide context. The application can provide faster feedback using this hybrid algorithm than it can if it rerenders the entire volume using the fast classification algorithm. If the user pauses for a long enough time then the background process can finish recomputing the run-length encoded volume and the main application can rerender the entire volume with the new classification.

4.5 Chapter Summary

In this chapter we developed three new volume rendering algorithms based on the shear-warp factorization: a rendering algorithm for parallel projections, a rendering algorithm for perspective projections, and a classification algorithm that can be incorporated into either rendering algorithm.

Unlike previous uses of the shear-warp factorization, the new algorithms exploit the properties of the factorization to combine the advantages of image-order and object-order algorithms. As in other object-order algorithms the new algorithms make efficient use of coherence data structures because they traverse the volume only once and they traverse the data structures in storage order, so there is very little overhead due to addressing arithmetic or inefficient memory access patterns. As in image-order algorithms the resampling filter is accurate and simple to implement, and the new algorithms include the early ray termination optimization.

Furthermore, we introduced coherence data structures for the fast classification algorithm that allow it to determine which portions of the volume are transparent without evaluating the classification function for each voxel. Unlike previous volume rendering algorithms based on spatial data structures the new algorithm allows a user to interactively adjust classification functions and see the results immediately.

This family of algorithms covers all phases of the visualization process outlined in Section 1.1.4. The next chapter examines the performance of the new algorithms and compares them to existing algorithms.