Chapter 2

Prior Work

Existing volume rendering algorithms fall into four classes: ray casting algorithms, splatting algorithms, cell projection algorithms and multi-pass resampling algorithms. The two characteristics that distinguish each class are the order in which an algorithm traverses the volume and the method an algorithm uses to project voxels to the image. Each class of algorithms has performance advantages and disadvantages, and furthermore, some classes of algorithms are more amenable than others to particular acceleration techniques.

The volume rendering literature is too large to summarize here, so in this chapter we will focus on some representative algorithms and acceleration techniques to illustrate the differences. In Chapter 3 we will describe a method for combining the advantages of ray casting and splatting algorithms: volume rendering with the shear-warp factorization.

2.1 Volume Rendering Algorithms

We begin by describing the four classes of existing volume rendering algorithms: ray casting, splatting, cell projection and multi-pass resampling.

We will use a taxonomy based on loop orderings. Most volume rendering algorithms consist of six nested loops: three loops that iterate over the volume and three loops that iterate over the resampling filter kernel. The loops can be interchanged (subject to ordering constraints imposed by the volume rendering equation) to produce different algorithms. The loop orderings are important for our purposes because they impact the performance characteristics of each algorithm.

2.1.1 Ray Casting

Ray casting algorithms produce an image by casting a ray through the volume for each image pixel and integrating the color and opacity along the ray, as already described in Section 1.1.2 [Levoy 1988, Sabella 1988, Upson & Keeler 1988]. Ray casters are called *image order* algorithms since their outer loops iterate over the pixels in the image:

```
for y_i = 1 to ImageHeight
  for x_i = 1 to ImageWidth
    for z_i = 1 to RayLength
        foreach x_o in ResamplingFilter(x_i, y_i, z_i)
          foreach y_o in ResamplingFilter(x_i, y_i, z_i)
            foreach z_o in ResamplingFilter(x_i, y_i, z_i)
                add contribution of Voxel[x_o, y_o, z_o] to ImagePixel[x_i, y_i]
```

The outer two loops iterate over the image pixels. The next loop iterates over sample points along a viewing ray in image space. Finally, the inner three loops iterate over the voxels required by the resampling filter to reconstruct one image-space sample. The body of the loop multiplies the value of a voxel by a resampling filter weight and adds the result to an image pixel.

Ray casters are also called *backward projection* algorithms since they calculate the mapping of voxels to image pixels by projecting the image pixels along viewing rays into the volume. Light rays flow forward from the volume to the image whereas viewing rays flow backward from the image into the volume.

The main disadvantage of ray casters is that they do not access the volume in storage order since the viewing rays may traverse the volume in an arbitrary direction. As a result, ray casters spend more time calculating the location of sample points (e.g. the voxel indices in the innermost loop body) and performing addressing arithmetic than other classes of volume rendering algorithms. Acceleration techniques based on spatial data structures further aggravate this problem. For example, when used with a ray caster some acceleration techniques require the calculation of intersections between rays and bounding boxes, as we will see below.

A second problem is that ray casters may have higher memory overhead because they do not have good spatial locality. They do not access the volume in storage order, so the caches in modern processors are less effective at hiding memory latency. However, this problem is usually of secondary importance for software implementations because the overhead due to memory latency is insignificant compared to the amount of computation.

2.1.2 Splatting

In contrast to ray casting algorithms, splatting algorithms operate by iterating over the voxels [Westover 1990]. This class of algorithms computes the contribution of a voxel to the image by convolving the voxel with a filter that distributes the voxel's value to a neighborhood of pixels, a process that is called *splatting* [Westover 1989]. Algorithms of this type are called *object order* algorithms since the outer loop iterates over voxels in the object being rendered:

```
for z_o = 1 to VolumeDepth
  for y_o = 1 to VolumeHeight
    for x_o = 1 to VolumeWidth
        foreach z_i in ResamplingFilter(x_o, y_o, z_o)
          foreach y_i in ResamplingFilter(x_o, y_o, z_o)
            foreach x_i in ResamplingFilter(x_o, y_o, z_o)
                add contribution of Voxel[x_o, y_o, z_o] to ImagePixel[x_i, y_i]
```

Compared to the ray caster, the inner and outer loops have been interchanged. The three outer loops iterate over the voxels. The next loop iterates over the extent of the resampling filter kernel in the image-space depth dimension. Finally, the two inner loops iterate over the pixels affected by a single voxel.

Splatters are also called *forward projection* algorithms since voxels are projected directly into the image, in the same direction as the light rays.

Because splatting algorithms are object-order algorithms they can stream through the volume in storage order, an advantage compared to ray casters. However, accurately computing the filter footprint and the resampling weights is expensive because the filter is view-dependent. In a forward-projection algorithm the filter footprint must be scaled, rotated and transformed arbitrarily depending on the viewing transformation. Moreover, in a perspective view this footprint changes from voxel to voxel. The filter kernel must also be chosen carefully to avoid gaps or excessive overlap between adjacent voxels after projection into the image. Thus it is difficult to implement a filter that is both efficient and that produces high quality results.

In theory, the splatter can produce exactly the same image as a ray caster provided both algorithms use the same filter weights. In practice, since it is difficult to compute the filter weights in the splatting algorithm, approximations must be used. Westover proposes a technique based on precomputed lookup tables for rotation-invariant Gaussian resampling filters. The method can be adapted for either fast execution or high image quality, but not both simultaneously [Westover 1990].

An alternative splatting algorithm uses 2D image warping techniques. Many volume renderers use this algorithm, including the commercial VoxelView package [Vital Images 1994] and methods based on texture mapping hardware [Cabral et al. 1994]. This algorithm partitions the volume into 2D slices and transforms each slice into image space by applying a 2D warp. The warped slices are then composited together to form the final image. This algorithm uses the same loop ordering as the standard splatting algorithm above except that the loop over the third dimension of the resampling filter kernel (z_i) is omitted since the resampling filter is two-dimensional. The 2D resampling filter is an approximation, but high-quality images can still be produced if the input volume has a high resolution. By using a 2D filter the algorithm reduces the complicated 3D resampling in Westover's algorithm to the well-understood problem of 2D image warping. However, the filter footprint and resampling weights are still view-dependent, so it is difficult to achieve a fast software implementation that does not compromise on image quality.

2.1.3 Cell Projection

A third class of algorithms consists of the cell projection techniques [Upson & Keeler 1988, Max et al. 1990, Shirley & Tuchman 1990, Wilhelms & Van Gelder 1991]. These methods are often used for volumes sampled on a non-regular grid. The first step is to decompose the volume into polyhedra whose vertices are the sample points (using a Delauney triangulation for instance). The algorithm then sorts the polyhedra in depth order and scan converts the polygonal faces into image space. Finally, the algorithm evaluates the volume rendering integral between the front and back faces of each polyhedron to compute a color and an opacity at each pixel, and these values are composited into the image.

Cell projection algorithms are similar to splatting algorithms except that the former use polygon scan conversion to perform the projection. Software implementations of this algorithm are computationally expensive, but the polygon scan conversion can be accelerated with special-purpose graphics hardware.

In this thesis we concentrate on efficient software-only algorithms, which have the advantages of portability and flexibility, so we will not consider cell projection algorithms further.

2.1.4 Multipass Resampling

The fourth class of volume rendering algorithms operates by resampling the entire volume to the image coordinate system so that the resampled voxels line up behind each other on the viewing axis in image space. The voxels can then be composited together along the viewing axis as in a ray caster, except that in the resampled volume the viewing rays are always axis-aligned.

This class of algorithms uses multipass methods to resample the volume: the viewing transformation is factored into a sequence of simple shears and scales which are then applied to the volume in separate passes. Each shear or scale of the volume can be implemented with a scanline-order algorithm and a 1D resampling filter. For example, an affine transformation can be implemented using three passes. The first pass resamples scanlines in the x dimension of the volume. The new volume becomes the input to the second pass which resamples scanlines in the y dimension. The result then feeds into the third pass which resamples scanlines in the z dimension. In terms of the loop-order taxonomy, multipass methods perform a series of object-order traversals of the volume each of the form:

OutputWidth = VolumeWidth $\times$ x_{scale}
for $z = 1$ **to** VolumeDepth
  for $y = 1$ **to** VolumeHeight
    for $x = 1$ **to** OutputWidth
      for $t = t_{\min}$ **to** $t_{\max}$
        add contribution of Voxel[$x/x_{\text{scale}} + t$, y, z] to Output[x, y, z]

The three outer loops iterate over the volume and the inner loop iterates over a 1D filter kernel.

If the size of the filter kernel is m voxels then an n-pass algorithm requires nm operations per voxel whereas a single-pass resampling algorithm with a 3D filter requires m^3 operations per voxel. Thus with a filter kernel size greater than one, a three-pass resampling algorithm requires fewer operations than a single-pass algorithm.

Catmull & Smith [1980] introduced multipass resampling for warping 2D images, and the technique was first applied to volume rendering at Pixar [Drebin et al. 1988]. Hanrahan [1990] derived an efficient three-pass algorithm for volume rendering with affine viewing transformations and Vézina et al. [1992] extended it to a four-pass algorithm for perspective viewing transformations.

Multipass resampling methods are appropriate for certain types of specialized parallel hardware that take advantage of the regular communication patterns of a scanline-oriented algorithm, such as the Pixar Image Computer [Drebin et al. 1988] and SIMD array processors [Vézina et al. 1992]. However, this class of algorithms is not optimal for use on a workstation. The filter footprint of the trilinear interpolation filter used in most software volume renderers is two voxels wide, so multipass resampling has little performance advantage ($2 \cdot 3$ operations versus 2^3 operations). Furthermore, multipass algorithms must use more expensive, higher-quality resampling filters to avoid image degradation (such as blurring) caused by the three or more resampling steps. Finally, this class of algorithms requires visiting every voxel multiple times and the volume must be transposed as the viewpoint changes (to avoid "bottleneck" problems [Catmull & Smith 1980]), leading to high costs for looping and memory overhead.

2.2 Acceleration Techniques

Many algorithmic optimizations have been proposed to speed up volume rendering. We will focus on acceleration techniques for ray casting and splatting since we saw in the previous section that they are the most appropriate methods for rendering regularly-sampled volumes on general-purpose hardware. In this thesis we also concentrate on acceleration techniques that do not trade off image quality for speed. For example, we do not consider subsampling the volume [Laur & Hanrahan 1991] or adaptive sampling [Levoy 1990a, Danskin & Hanrahan 1992]. Both of these techniques can miss or blur small features, although Laur & Hanrahan [1991] and Danskin & Hanrahan [1992] provide error bounds so that error can be controlled. In the rest of this section we consider two important optimizations that do not affect image quality: coherence acceleration using spatial data structures, and early ray termination.

2.2.1 Spatial Data Structures

An established acceleration technique for volume rendering is to exploit coherence in the volume by using a *spatial data structure*. For a given visualization of a data set, typically there are clusters of voxels that contribute useful information to the image and other clusters that are irrelevant. The purpose of a spatial data structure is to encode this type of coherence in such a way that the irrelevant voxels can be culled efficiently.

Spatial data structures are often used to encode the location of high-opacity voxels in a classified volume. Data structures that have been used for this purpose include octrees and pyramids [Meagher 1982, Levoy 1990a], k-d trees [Subramanian & Fussell 1990], run-length encoding [Reynolds et al. 1987, Montani & Scopigno 1990], shells [Udupa & Odhner 1993] and distance transforms [Zuiderveld et al. 1992]. Rendering algorithms use these data structures to skip transparent voxels rapidly.

Spatial data structures have also been used to encode other types of coherence. They have been used to accelerate isosurface renderings by building an octree containing min-max bounds [Wilhelms & Van Gelder 1992] or Lipschitz bounds [Stander & Hart 1994] for the scalar values within a volume, and to encode statistical information about the voxel opacities so the sampling frequency on a viewing ray can be reduced in homogeneous regions of the volume [Danskin & Hanrahan 1992].

Algorithms that rely on spatial data structures require a preprocessing step in order to compute the data structure. Often the cost of preprocessing is acceptable if it can be amortized over many renderings. However, none of the existing coherence optimizations can be used for interactive classification of a volume since they all require preprocessing every time the voxel opacities change. Wilhelms & Van Gelder [1992] and Stander & Hart [1994] propose methods that allow the user to change a threshold value interactively during isosurface rendering but neither technique applies to general classification functions. While the preprocessing step is a disadvantage of previous algorithms, in Chapter 4 we present a new algorithm that eliminates the problem by supporting general classification functions with a classification-independent octree.

The speedup achievable with coherence optimizations is data-dependent, but typical classified volumes have a high degree of coherence. Furthermore the percentage of voxels that are transparent is typically 70-95% [Levoy 1990a, Subramanian & Fussell 1990], so the coherence can be used to eliminate a substantial fraction of work. Levoy reports that for medical data sets a ray caster with an octree to exploit coherence achieves a 3-5x speedup over a brute-force ray caster [Levoy 1990a].

These speedups are substantial, but we will see in Chapter 5 that most spatial data structures are more effective when used with an object order rendering algorithm rather than an image order algorithm. In the case of a ray caster with an octree-encoded volume, every ray must independently descend the octree, resulting in redundant computation. The costs of intersecting rays with octree nodes and switching between levels of the octree are significant overheads and limit the speedup, as we will see in Section 5.2. In contrast, a splatting algorithm with an octree only traverses the volume once and requires less-complicated addressing arithmetic since it sweeps through each octree node in storage order. Efficient use of spatial data structures is an important advantage of object-order algorithms.

2.2.2 Early Ray Termination

A second common acceleration technique for volume rendering is a technique called *early ray termination*. This optimization is most easily implemented in a ray casting algorithm: the algorithm traces each ray in front-to-back order and terminates the ray as soon as the accumulated ray opacity reaches a threshold close to full opacity. Any additional voxels that would have been reached by the ray must be occluded, so they need not be rendered. Two generalizations of early ray termination are Russian roulette [Arvo & Kirk 1990, Danskin & Hanrahan 1992], which terminates rays according to a probability that increases with accumulated ray opacity, and β-acceleration [Danskin & Hanrahan 1992], which decreases the sampling rate along the ray as optical distance from the viewer increases.

The goal of these optimizations is to reduce or eliminate samples in occluded regions of the volume. Levoy reports that a ray caster rendering medical data sets with early ray termination using a threshold of 95% opaque achieves speedups of 1.6-2.2x, and combined with an octree the overall speedup is 5-11x [Levoy 1990a].

Efficient implementation of early ray termination is trivial for a ray caster but is difficult for an object-order algorithm. The naive implementation of early ray termination in a splatting algorithm is to check each pixel's opacity before compositing a voxel into it. However, the algorithm would still have to visit every voxel in the data set. This optimization reduces the number of shading, resampling and compositing operations, but does not reduce the time spent traversing occluded portions of the volume.

Meagher [1982] proposes a more sophisticated algorithm that exploits coherence in the image. The algorithm uses a quadtree data structure to encode regions of the image with high accumulated opacity, in addition to an octree to encode coherence in the volume. To render a volume the algorithm traverses the octree nodes in front-to-back order and splats the voxels in each node into the image. Before splatting, the algorithm computes the visibility of a given octree node by scan-converting the silhouette of the node into image space and checking the accumulated opacity in the quadtree nodes that overlap the silhouette. If the quadtree nodes are opaque then the voxels in the octree node can be culled.

The octree and quadtree data structures significantly reduce the complexity of the algorithm. Instead of individually examining each voxel as in the naive algorithm, Meagher's algorithm can cull an occluded octree node of arbitrary size by checking a small number of quadtree nodes. Greene et al. [1993] use Meagher's image-space quadtree technique in a rendering algorithm for very large polygonal models. However, the method is less appropriate for interactive volume rendering because the overhead required to scan convert octree nodes and traverse the quadtree data structure is likely to be large compared to the rendering time. This overhead eliminates much of the gain achieved by culling the occluded octree nodes.

Reynolds et al. [1987] propose a more efficient technique, called a "dynamic screen," for implementing early ray termination in an object-order volume rendering algorithm. Instead of using an octree and a quadtree, the technique uses run length encoding to encode coherence in both the volume and the image. Furthermore, the authors make the following observation: if the viewing transformation is a parallel projection then it transforms parallel lines in object space into parallel lines in image space. As a result, for some rotation of the image every voxel scanline projects onto a line that is parallel to the rotated image scanlines.

This observation leads to an efficient rendering algorithm: for each voxel scanline (in front-to-back order) the algorithm simultaneously traverses the voxel scanline and the image scanlines it projects onto. During the traversal, the algorithm uses the run length encoding of the voxel scanline to skip transparent voxels and the run length encoding of the image scanline to skip occluded voxels. The algorithm only composites voxels in runs containing non-transparent voxels that project onto non-opaque pixels. After the algorithm finishes traversing all of the voxel scanlines it rotates the 2D image to the correct orientation.

This algorithm never visits voxels that are occluded, which is the goal of early ray termination. It also has less overhead than Meagher's algorithm because it is less expensive to compute intersections between 1D runs than to scan-convert octree nodes and descend a quadtree. The implementation reported by Reynolds is fast compared to contemporary results but uses binary thresholding of the voxel data and point sampling, resulting in low-quality images. The algorithm is also limited to parallel projections. In Chapter 4 we will expand on this idea to support both high-quality images and perspective projections.

2.3 Chapter Summary

In this chapter we have seen that previous algorithms appropriate for volume rendering on general-purpose workstations fall into two classes, image-order ray casters and object-order splatting algorithms. We also observed that both types of algorithms have their own advantages: Object-order algorithms can efficiently traverse a spatial data structure to find the non-transparent voxels, but resampling is complicated because the filter is view dependent. Image-order algorithms must perform more work to traverse the spatial data structure but resampling is much simpler and these algorithms can take full advantage of early-ray termination. In the next chapter we describe a method for combining the advantages of these two classes of algorithms.