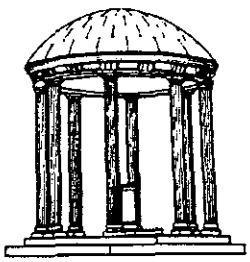
SPLATTING:
A Parallel, Feed-Forward Volume Rendering Algorithm

TR91-029

July, 1991

Lee Alan Westover



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

SPLATTING:
A Parallel, Feed-Forward Volume Rendering Algorithm

by
Lee Alan Westover

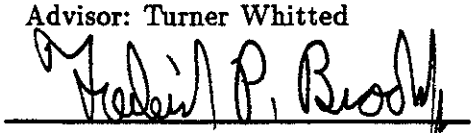
A dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

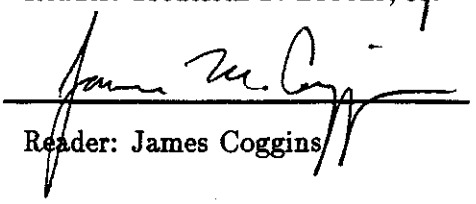
1991

Approved by:

Advisor: Turner Whitted



Reader: Frederick P. Brooks, Jr.



Reader: James Coggins

LEE ALAN WESTOVER. SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm (Under the direction of TURNER WHITTED.)

Abstract

Volume rendering is the generation of images from discrete samples of volume data. The volume data is sampled in at least three dimensions and comes in three basic classes: the rectilinear mesh—for example, a stack of computed tomography scans; the curvilinear mesh—for example, computational fluid dynamic data sets of the flow of air over an airplane wing; and the unstructured mesh—for example, a collection of ozone density readings at multiple elevations from a set of collection stations in the United States.

Previous methods coerced the volumetric data into line and surface primitives that were viewed on conventional computer graphics displays. This coercion process has two fundamental flaws: viewers are never sure whether they are viewing a feature of the data or an artifact of the coercion process; and the insertion of a geometric modeling procedure into the middle of the display pipeline hampers interactive viewing.

New direct rendering approaches that operate on the original data are replacing coercion approaches. These new methods, which avoid the artifacts introduced by conventional graphics primitives, fall into two basic categories: feed-backward methods that attempt to map the image plane onto the data, and feed-forward methods that attempt to map each volume element onto the image plane.

This thesis presents a feed-forward algorithm, called splatting, that directly renders rectilinear volume meshes. The method achieves interactive speed through parallel execution, successive refinement, table-driven shading, and table-driven filtering. The method achieves high image quality by paying careful attention to signal processing principles during the process of reconstructing a continuous volume from the sampled input.

This thesis' major contribution to computer graphics is the splatting algorithm. It is a naturally parallel algorithm that adheres well to the requirements imposed by signal processing theory. The algorithm has uncommon features. First, it can render volumes as either clouds or surfaces by changing the shading functions. Second, it can smoothly trade rendering time for image quality at several stages of the rendering pipeline. In addition this thesis presents a theoretical framework for volume rendering.

Acknowledgements

Thanks to

Turner Whitted for being my advisor, my boss and my friend.

Turner Whitted, Frederick P. Brooks Jr., James Coggins, Henry Fuchs, and Stephen Pizer for serving as my committee.

Apple Computer Incorporated, GRIP molecular modeling project, Numerical Design Limited, Schlumberger-Doll Research, and Sun Microsystems Incorporated for supporting portions of this research.

Leonard McMillan for his careful proofreading of this thesis.

Robert Whitton for helping me understand some of the more obscure mathematics.

Gary Bishop, John Zimmerman, and Mark Harris for their sometimes not so gentle pushes for me to finish.

Melvin Billik, who introduced me to computers and started this quest.

Guinness, BooBoo, Vanna, and Remote for blindly putting up with my late hours and my mood swings.

Mother and Father for their love, for answering my oh-so-many questions throughout growing up and for their belief that I could actually finish.

Rebekah, my wife and best friend, who knew when to ask how things were going and when not to ask.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Description of the Problem

A common question in scientific research is how does one interpret the large amounts of data generated by experiments and computations. A key to interpretation is the ability to visualize and explore the data, which often take the form of scalar and vector quantities sampled at discrete points in a space. Volume data sets are three- or more dimensional data sets, where the dimensions can be three spatial dimensions or another combination such as two spatial dimensions and one frequency dimension. These data typically fall into one of three general classes. Rectilinear meshes have samples with topologically and geometrically regular intervals such as a stack of computed tomography scans. Curvilinear meshes have samples with topologically regular but geometrically irregular intervals such as those commonly used in simulations of velocity and pressure of air flow over an airplane wing. Unstructured meshes have samples with topologically and geometrically irregular intervals such as a collection of ozone density readings at multiple elevations from a set of collection stations in the United States.

Researchers commonly view volumetric data with conventional line and surface rendering methods. They coerce the sampled data into lines or surfaces, and then view these primitives on conventional computer graphics displays [Levinthal 66], [Wright 72], [Fuchs 77], [Ganapathy 82], [Williams 82]. Engines for the interactive display of large collections of display primitives are readily available and widely used in scientific applications. This approach has two fundamental problems. First, the coercion process is prone to errors in the form of geometric artifacts. In the coercion process, the system must decide whether each sample is part of a display primitive or the system must fit a display primitive to each sample. There are many situations for which this decision may not yield satisfactory results as in the case of an isovalue surface that both enters and exits a single face of a volume element. Once the coercion process finishes, the renderer displays the primitives instead of the original data. As a result, the viewers often do not know if they are looking at a feature of the data or false features introduced by the coercion process.

Second, the insertion of a geometric modeling procedure into the middle of the display pipeline hampers interactive viewing. Since many of these surface generation processes are compute-intensive and some methods require human intervention during a set of trial and error attempts to accurately fit the data [Williams 82], the coercion process is usually run as a preprocess. If the coercion process did not select the proper surface

or the user selects a different isovalue surface to view, the preprocessor must be rerun. Alternatively, the preprocessor can generate a set of contour surfaces for the data set and the user can interactively change which subset of the contours he wishes to see. In order for the user to arbitrarily view any contour or set of contours, the preprocess must generate all possible contours, which slows the coercion process further. In addition, many of the resulting contour surfaces may never be viewed, as the user can only view a few at a time or the image becomes too busy.

This thesis addresses the problem of direct display of volumetric rectilinear meshes. My original investigations attempted to display an artificially generated cloud model. The cloud generator produced a density function sampled regularly along the three spatial dimensions, generating a rectilinear mesh. This data format is similar to that used in medical imaging, seismology, computational fluid dynamics, and molecular modeling. Data from these disciplines are frequently rectilinear meshes. This similarity illustrates the importance of finding a technique to directly display the three-dimensional data with minimal artifacts, at interactive rates, and without a significant loss of information.

The result of this effort is an algorithm for volume rendering that has many features not commonly found in a single algorithm. First, it is a feed-forward algorithm that maps the data elements into the image plane rather than mapping the image plane into the data set. Because it uses a feed-forward approach, the algorithm is easily parallelizable to run on a multiprocessor without having to replicate the entire data set at each node. Second, it applies signal processing principles to the reconstruction of the discrete input samples into a continuous volume before the the renderer samples the volume, yielding an accurate image relatively free of sampling artifacts. Third, it introduces ways of using table-driven processes for both the shading and the reconstruction processes to reduce the computational requirements of the renderer. Lastly, it uses non-binary classifiers to soften boundaries between classified regions (as do most direct volume renderers).

1.2 Thesis

My thesis is:

> **"Volume rendering can be described within the framework of linear systems theory."**
>
> **"Splatting: a feed-forward volume rendering algorithm can be made to conform to this theory."**
>
> **"Parallel implementations of splatting need not replicate the entire volumetric data set at each processing node."**

1.3 Definition of Terms

Splatting: *Splatting* is the name of the feed-forward volume rendering algorithm. The name is derived from a non-technical description of the feed-forward volume-rendering process. Consider the input volume to be a stack of snowballs and the image plane to be a brick wall. Image generation is the process of throwing each snowball, one-by-one,

at the wall. As each snowball hits the wall, the snowball flattens, makes a noise that sounds like "splat" and spreads its contribution across the wall. Snowballs thrown later obscure snowballs thrown earlier. This mental picture of the feed-forward rendering process inspired the name splatting. In the actual algorithm, the term *splat* refers to the process of determining a sample's image-space footprint on the image plane, and adding the sample's effect over that footprint to the image.

Classification, Reflection, Illumination, Opacity — CRIO: Conceptually, the the feed-forward renderer treats a sample as a reflective, light-emitting, semi-transparent cloud. *Classification*, in the context of this thesis, is the process of determining the discrete values for the primary properties that represent the sample. This choice may be based on the density value of the sample, the gradient magnitude near the sample, or any other feature of the sample. Emitted color, reflected color and opacity are currently the only three primary properties. Once a sample has been classified, it is illuminated using an illumination model. This can be a simple identity function, where the sample's color is merely the primary color, or this model may be more complicated as a result of taking into account directional lighting effects similar to Phong shading [Phong 75]. For the remainder of this thesis, the process of classifying a sample to determine its primary properties and applying an illumination model to the sample to calculate the illumination effects is called the *CRIO* process.

Opacity and Color: As stated above, opacity and color are primary properties of a data sample. The samples are treated as being semi-transparent and the percentage of background that does not show through the the semi-transparent foreground is called the sample's *opacity*. An opacity of 0.0 indicates the foreground is completely transparent and blocks none of the light from behind it. An opacity of 1.0 indicates the foreground is completely opaque and blocks all of the light from behind it. An opacity of 0.5 indicates the foreground is 50% opaque and blocks one-half of the light from behind it. The color of a fully opaque sample is only dependent on that sample and the light illuminating it. My feed-forward renderer operates in the $< red, green, blue >$ color space. This is not a requirement of the splatting method. It was chosen simply because it is a common way to represent color images. Monochrome or spectral color could be substituted for the current color model without significant changes to the method described in the thesis.

Compositing: *Compositing* is linear interpolation between a background color and a semi-transparent foreground color [Porter 84]. Let $C_{background}$ be the background color, $C_{foreground}$ be the foreground color, and $O_{foreground}$ be the foreground opacity. Compositing is defined mathematically by the composite operation

$$C_{result} = O_{foreground} \times C_{foreground} + (1 - O_{foreground}) \times C_{background}.$$

Feed-forward vs. Feed-backward: The difference between a feed-forward process and a feed-backward process is the process's mapping direction. If the output of the process asks for information regarding the input of the process—such as rays coming from pixels into the data in the case of a ray-casting algorithm, the process is called *feed-backward*. In

computer graphics, these processes are often called *image-order processes.* If the input of the process spreads its information to the output—such as triangles in a polygon z-buffer algorithm changing the pixel values, the process is called *feed-forward.* In computer graphics, these processes are often called *object-order processes.*

Binary vs. non-binary classifiers: The classification process, as described above, selects a color and an opacity for each input sample value.

If the result of the classification is a two-valued function in which some of the input values belong to an opaque object and all other values do not belong to the opaque object and are fully transparent, the classification process is a *binary classifier.* Binary classifiers have been used since the first images were generated of volume data. They were originally used in volume rendering because the value of each data element could be represented as a single bit which significantly reduced volume storage requirements. *Non-binary classifiers* are classifiers that permit a range of output from the classification process. These come in two types. The first type, called *hard classifiers*, have sharp transitions in the output for input values that are close together. The sharp transitions may cause artifacts in the rendered image as discussed in section 2.4. The second type, called *soft classifiers*, do not have sharp transitions in the output for input values that are close together. These classifiers allow the output to smoothly move from one level to another without sharp jumps. The distinction between hard and soft classifiers is not well defined, but merely differentiates groups within the group of non-binary classifiers.

Figure 1.01 shows an example of each of the three classifier types. The left represents a binary classifier. Notice how the function has only two values, 0.0 and 1.0. The middle represents a hard classifier. Notice that even though the function takes on a range of values between 0.0 and 1.0, there are C^1 discontinuities in the function. The right represents a soft classifier. Like the hard classifier, this classifier takes on values between 0.0 and 1.0. However, there are no abrupt changes in function value for nearby input values.

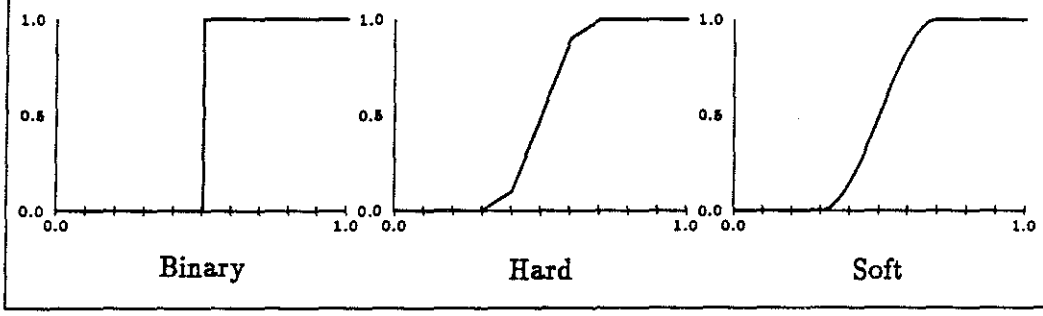


Figure 1.01: Example Classifiers.

1.4 Other Work

Figure 1.02 represents present volume-rendering methods divided along three axes. The first axis indicates whether the method directly renders the volume data or whether it coerces the data into other primitives before display. In the coercion methods, the input data is modeled with either line or surface primitives and then these primitives are rendered, using standard line- or surface-rendering algorithms, such as a z-buffer. This was an early, commonly used method for rendering volume data sets.

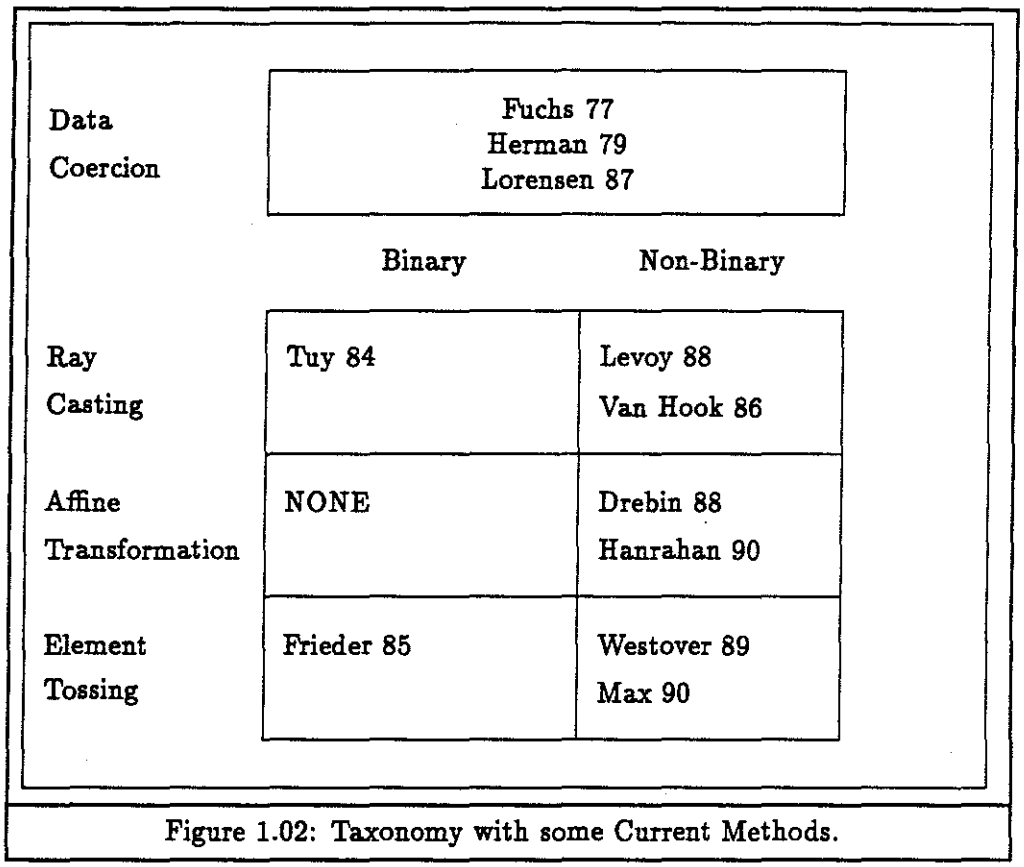


Figure 1.02: Taxonomy with some Current Methods.

According to Levoy [Kaufman 91, page 89], direct methods may be further divided along two axes. The first axis represents whether the method uses binary or non-binary classification. Binary methods dictate that each sample is either a part of or not a part of the desired object. Non-binary methods may have samples that are fractionally part of the desired object.

The second axis represents the type of projection method. There are three basic types of projection in direct volume-rendering algorithms. The first method is used in the feed-backward methods. The methods cast rays through the image plane into the data set to determine the color for each pixel. The second method is used in the hybrid methods. Affine transformation methods, an example of a hybrid method, use well-known feed-forward image warping methods to transform the data so it is pixel aligned.

Once the data is pixel aligned, they composite the transformed data to calculate visibility. The third method is used in the feed-forward methods. These methods map each data element from the data set onto the image plane. They determine how the sample affects the image, and add the sample's contribution to the image. These methods are called *element-tossing* methods because they send each data element from the data set onto the image plane. The following sections briefly describe representative methods for each part of the taxonomy.

At SIGGRAPH 1986, researchers from Pixar displayed the first high quality volume-rendered images. They exhibited images of a computed tomography study of a female torso in a movie informally known as "The Spinning Fat Lady". The astonishing quality of these images encouraged researchers to investigate alternatives to surface rendering long before the group published details of this work at SIGGRAPH 1988 [Drebin 88] and in U.S. Patent 4,835,712. Subsequent researchers explored volume rendering [Van Hook 86] and related techniques [Levoy 85] in greater detail.

Research on volume rendering has progressed along four lines: data-coercion, ray-casting, compositing using affine transformation, and compositing by element tossing.

1.4.1 Data-Coercion Method

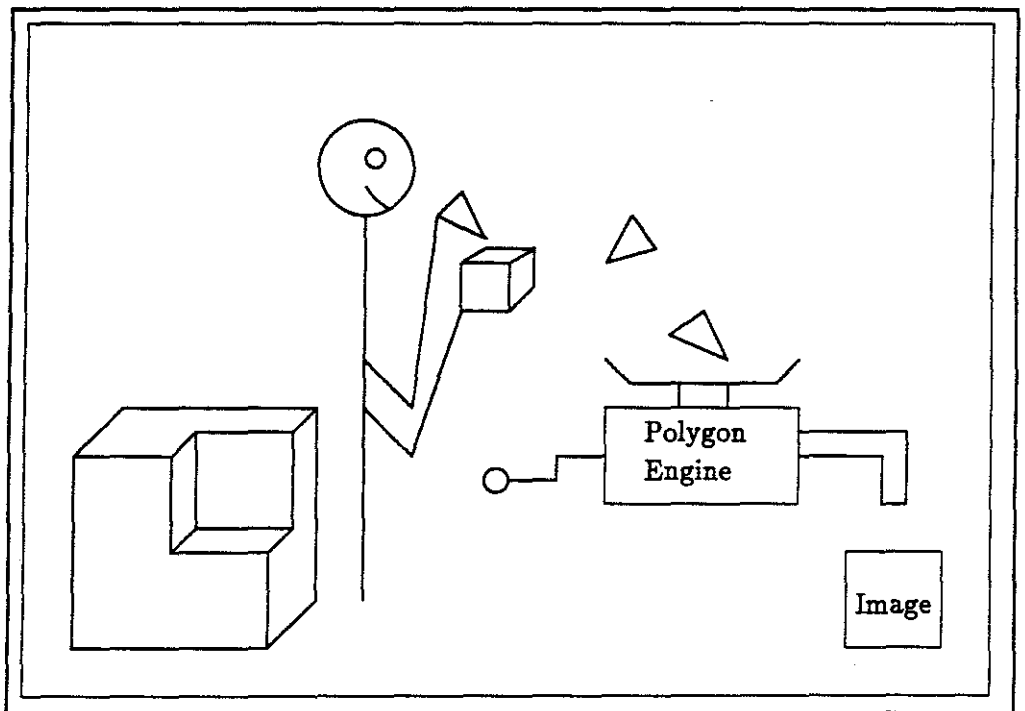


Figure 1.03: Data-Coercion Method.

[Fuchs 77], [Herman 79], [Ganapathy 82], [Meagher 84], [Lorensen 87], [Cline 88], [Upson 88] and [Gallagher 89], have investigated various methods of fitting surfaces in the data and then rendering the surfaces. In Figure 1.03, the renderer is fitting a

triangle to a volume element. Triangles from other elements are already on their way to the conventional polygon engine that is generating the image.

Fuchs, Kedem, and Uselton [Fuchs 77] introduced the first widely used coercion method. First, a preprocess generates contours within single slices of the data. The method forms inter-slice triangles from these intra-slice contours by solving the problem of finding minimum-cost cycles in a directed-toroidal graph. Their optimal solution generates the minimum-area surface. They bound the maximum cost of finding the optimal solution for a set of contours with m points in one contour and n points in the other contour as

$$\tau(m,n) < (\lceil log_2 m \rceil + 2) \times (2mn + m) \qquad \text{for} \qquad (m < n)$$

Conventional polygon rendering displays render the triangles to generate the image.

Ganapathy and Dennehy [Ganapathy 82] introduced a heuristic method based on inter-contour coherence. They theorize that if the method can obtain some knowledge of the nature of the object, these methods never need more than $m + n$ steps to form the triangles between the two contours, where m and n are the number of points in the two contours. This bound is significantly less than the above bound [Fuchs 77]. These heuristic methods are valuable when computational speed is more important than optimal results.

Herman and Liu [Herman 79] proposed the cuberille method. This method thresholds the data using a binary classifier to generate a binary array. The method renders ones in this array as opaque cubes by rendering each of the six faces into a z-buffer. Because the data comes presorted in the volume array, the method can determine a back-to-front traversal and can use a "Painter's" algorithm instead of the z-buffer.

Meagher [Meagher 84] proposed storing the above binary array as an octree. The binary array is often sparsely occupied and the areas that are occupied are grouped together. This coherence allows the octree to greatly increase the method's traversal speed.

Lorensen and Cline [Lorensen 87] introduced a simple and efficient method for generating surface data. The "marching cubes" method creates a triangle model of an isovalue surface by a divide and conquer approach. The method fits a triangle at each isovalue surface value for each volume element using a look-up table and calculates the triangle vertices using linear interpolation. The renderer renders this surface using standard polygon rendering algorithms and uses local gradients for the shading normal.

Cline, Lorensen, Ludke, Crawford, and Teeter [Cline 88] enhanced this method to generate individual points instead of triangles. They noticed that as the resolution of their data sets increases, the number of triangles generated is greater than the number the pixels in the image. In the "dividing cubes" algorithm, the method generates points at each isovalue surface instead of triangles. The relative difference in the pixel and volume element sampling rates controls the density of points.

Upson and Keller [Upson 88], introduced two methods: a ray-casting method, similar to those described below, and a polygon-based method. In the polygon method, the

renderer intersects scan planes, the z extension of scan lines, with each volume element forming intersection polygons. The method breaks these intersection polygons into spans and integrates the pixels of each span front-to-back to generate opacity. The renderer operates on the volume elements in a predetermined front-to-back order to calculate visibility.

Gallagher and Nagtegaal [Gallagher 89] describe an algorithm that builds upon the "marching cubes" method with an algorithm that works on unstructured as well as rectilinear meshes. It uses visually continuous bicubic polynomials instead of triangles. An advantage of this method is that it processes each volume element in isolation, thus minimizing the amount of information accessed in the inner loop. The authors contend the above is desirable for efficient microcoding of the inner-loop on a single workstation. In addition, the fact that the elements are treated independently lends the algorithm to parallel implementations.

1.4.2 Ray-Casting Method

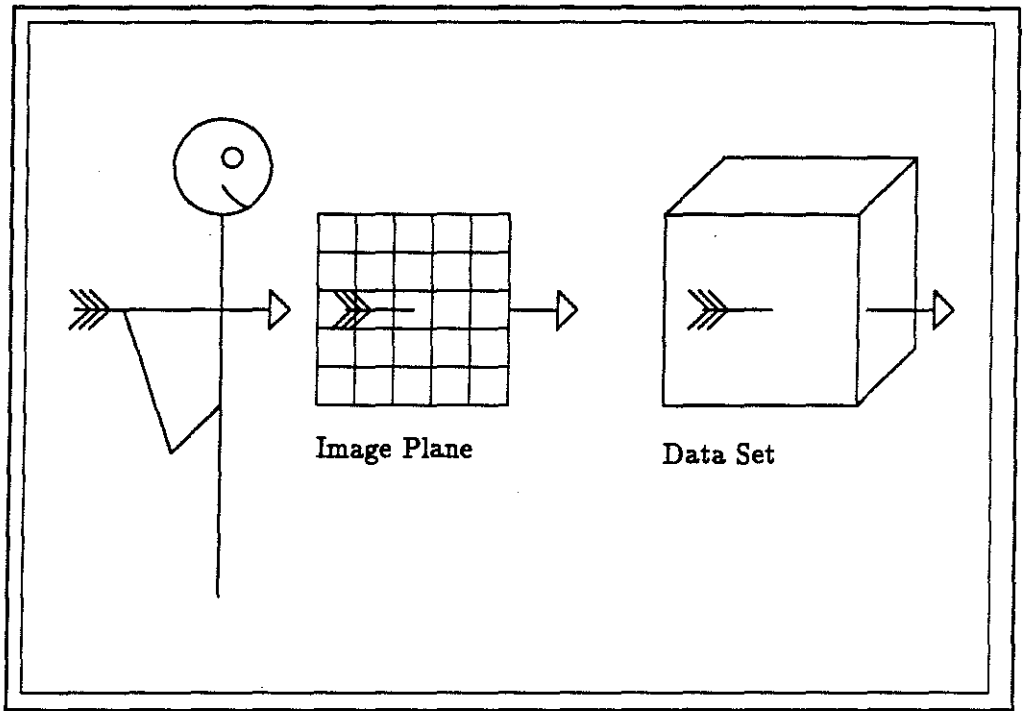


Figure 1.04: Ray-Casting Method.

[Blinn 82], [Kajiya 84], [Tuy 84], [Van Hook 86], [Levoy 88], and [Sabella 88], describe methods of ray-casting volume densities with algorithms that use projections of lines through the image plane to map pixels into the data set. In Figure 1.04, the renderer is throwing a ray through the image plane and into the data. The ray continues through the data until it has accumulated enough density to become opaque. The ray's final color is determined when the ray either becomes opaque or exits the volume.

Blinn [Blinn 82] first began investigating the use of scattering models to simulate how light interacts with density functions to render the rings of Saturn. Kajiya [Kajiya 84] investigated ray-casting volume densities as a way to render clouds and other phenomena defined on rectilinear meshes. In each case the goal was to produce a visually accurate depiction of natural phenomena. Their methods of shading were adopted as the preferred shading method in volume rendering. As seen in results from [Drebin 88] and [Levoy 88], conventional surface shading methods [Phong 75] work well with binary classifiers. However, the use of scattering models extends the utility of volume renderers to non-binary classifiers where there are no true surfaces and the desired image has cloud-like properties.

Tuy and Tuy [Tuy 84] introduced a binary classifying ray-casting method. The method first thresholded the data set to generate a binary data set. Then for each pixel, they cast rays into data set which stopped when the ray hit an opaque sample. They modeled each sample as a rectilinear solid.

Van Hook [Van Hook 86] and Levoy [Levoy 88] explored the basic ray-casting model for volume rendering. This method casts rays from the image plane into the data set, in a feed-backward algorithm. Each ray travels through the data set accumulating shade and opacity according to a rule that accounts for light transmission and reflection. At points along the ray, the renderer interpolates the data set to generate sample points, shades each sample point by a shading model, and adds the results of the shading model to the current ray. The process terminates when the ray exits the volume or the accumulated opacity along the ray equals one. When the opacity is one, no other data points can affect the ray's color. This early termination optimization is unique to the feed-backward method.

Sabella [Sabella 88] introduced a ray-casting method that uses an accurate light-scattering model to model the scattering effects of light in a cloud. Based on [Blinn 82], [Kajiya 84], and [Max 86], Sabella's model uses the view rays as the basis for computing line integrals through the data to generate the color for each ray. His approach generates cloud-like images that illustrate internal structure, instead of contour surfaces of the data.

1.4.3 Affine-Transformations Method

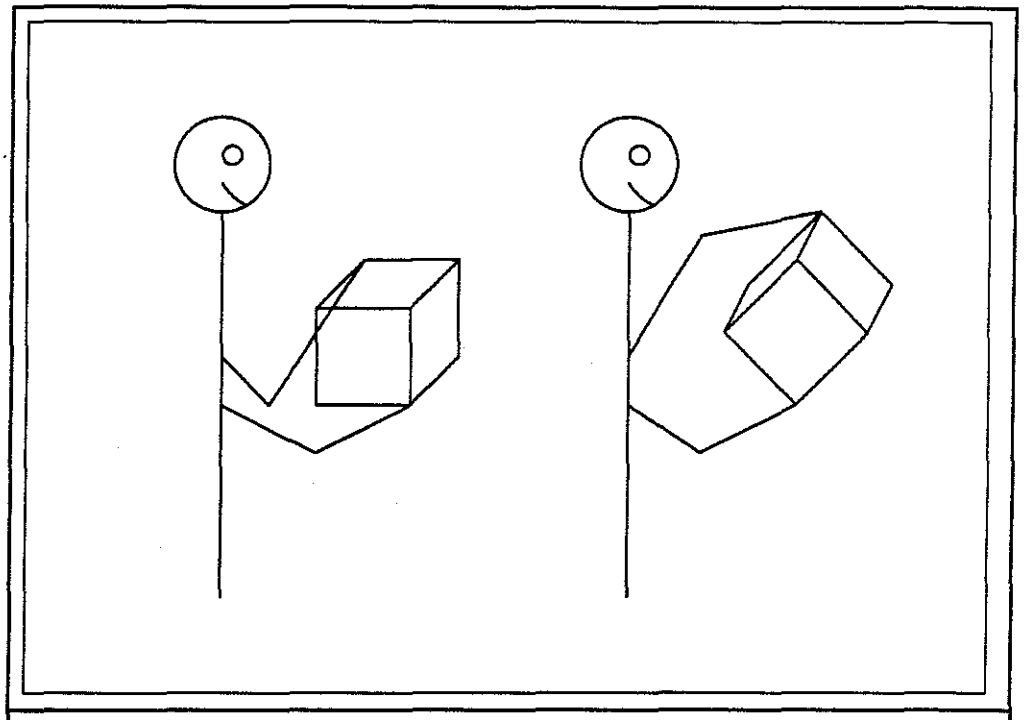


Figure 1.05: Affine-Transformations Method.

[Drebin 88] and [Hanrahan 90] use compositing techniques that warp slices or volumes into the image plane using affine transformations. In Figure 1.05, the renderer is warping the data until the data samples become pixel-aligned. Once the data samples are pixel-aligned, the renderer will scan each depth column of pixels and determine the pixel's color.

In 1988, Drebin, Carpenter, and Hanrahan [Drebin 88] presented the earlier Pixar work. Their feed-forward algorithm is an image warping method. The renderer uses conventional two-pass image warping [Catmull 80] to resample the volume elements in each slice so the resulting slices are pixel-aligned. It then converts the warped data set into a substance data set, an opacity data set, a gradient data set, and possibly a mask data set. The renderer uses these multiple data sets to generate shaded slices that it composites together to form the final image.

Hanrahan [Hanrahan 90] extended the above method to a three-pass algorithm. He developed the affine-mapping equations and discussed a way to avoid resampling artifacts by instructing the renderer to make all the magnification passes before the minification passes. If a minification step precedes a magnification step, much of the information in the signal would be lost, since multiple samples are collapsed into a single sample during minification and they cannot be separated during a subsequent pass. This ordering allows the renderer to retain the maximum amount of the signal's information during the transformation.

1.4.4 Element-Tossing Method

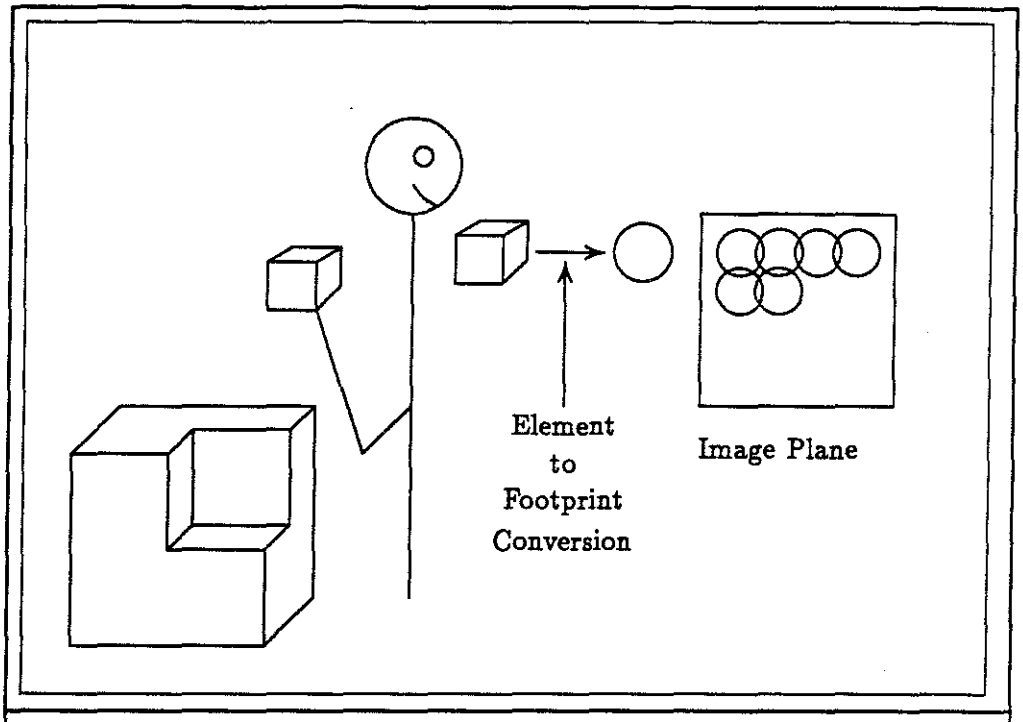


Figure 1.06: Element-Tossing Method.

[Frieder 85], and [Max 90] use compositing techniques that map each volume element directly into the image plane. Element tossing approaches incrementally add each sample to the image by first determining an area of effect and then adding the sample's contribution over this area. In Figure 1.06, the renderer is throwing the data samples at the image plane. The renderer calculates the image-plane footprint of the element as the element heads toward the plane, and incrementally adds the element's affect to the image.

Frieder, Gordon, and Reynolds [Frieder 85] presented a back-to-front method that traverses a rectilinear mesh in a back-to-front order. Volume elements that have a density that lies within a given range are coded as part of an object and these elements are mapped to a pixel and added to the image. Since the elements are either in the object or not in the object, the renderer simply overwrites the pixel value with the element's color. The method shades the samples based on distance from the image plane, an imaginary light source, and binary classification. This method depends on the density of input samples being higher than the density of image samples to prevent gaps in the displayed image.

If neighboring volume samples are spanned by polygons, rendering the polygons allows volume data to be rendered at any scale without gaps. Max, Hanrahan, and Crawfis [Max 90] have developed a method of rendering unstructured meshes by rendering each

polyhedron individually. It scan-converts the front faces and the back faces of the polyhedron into separate buffers and then integrates the density between these faces using a scattering model. It then composites these partial images together in a back-to-front order.

1.5 Thesis Overview

This thesis presents a element-tossing feed-forward volume-rendering algorithm. It operates on rectilinear meshes that are uniformly sampled in each mesh direction. However, the sampling rate does not have to be the same in each of the mesh directions. The method uses non-binary classifiers to reduce rendering artifacts. In addition, a small local neighborhood surrounding a sample contains all the information a renderer needs for that sample. This allows the renderer to treat each sample independently, so that when the renderer runs in parallel, there is little replication of the input data set. Both the reconstruction and the CRIO methods use tables extensively to reduce computation.

Chapter 2 presents a theoretical framework for this and other volume-rendering algorithms.

Chapter 3 describes the rendering pipeline, including transformation, CRIO, reconstruction, and visibility. Chapter 3 also describes methods for table-driven CRIO and reconstruction processes.

Chapter 4 discusses two ways to speed up rendering: successive refinement and parallel execution.

Chapter 5 describes some compromises of my implementation of the splatting algorithm.

Chapter 6 outlines the major areas of future work.

Chapter 2

Volume Data Sampling and Reconstruction

2.1 Introduction

Signal processing forms the basis of volume rendering because volume rendering involves the reconstruction of the input data and a resampling to generate a discrete image. The input data set is a collection of samples taken from an original input signal or generated by a computation. The sampling theorem states [Shannon 49], if the volumetric sampling rate is at least twice the original source's maximum spatial frequency, it is possible to fully reconstruct the original signal from the sampled signal. The *Nyquist rate* of a signal is defined as the frequency twice as high as the highest frequency in a signal. If the original signal contains higher frequencies than one half the volumetric sampling rate, which is normally the case, the sampled signal does not contain the information required to properly reconstruct the input signal, and aliasing artifacts will be introduced. These samples may not represent the original source, but they do represent a continuous function. Volume renderers assume the original signal was properly filtered before sampling and this continuous filtered signal is the signal they are trying to display, since the volume renderer does not have access to the original source and can only work from the samples.

The volume-rendering pipeline is a series of geometric mapping and filtering steps. For a given view, the renderer reconstructs, transforms, shades, filters, and resamples the input data to generate the image. The ordering of these operations differs among volume-rendering methods. This section will describe the ideal sequence of operations.

From a signal-processing point of view, the ideal process would take the following steps.

(1) Reconstruct the continuous volume function from the input samples by convolving the samples with a properly chosen reconstruction filter kernel.

(2) Transform this continuous function into image space.

(3) Shade the continuous function.

(4) Filter the continuous shaded function with a low-pass filter to lower the signal's maximum frequency so that it is below one half the image resampling rate.

(5) Sample the function at a rate corresponding to image resolution and calculate the visibility function to generate the image.

Figure 2.01 is a block diagram of the idealized volume-rendering process.

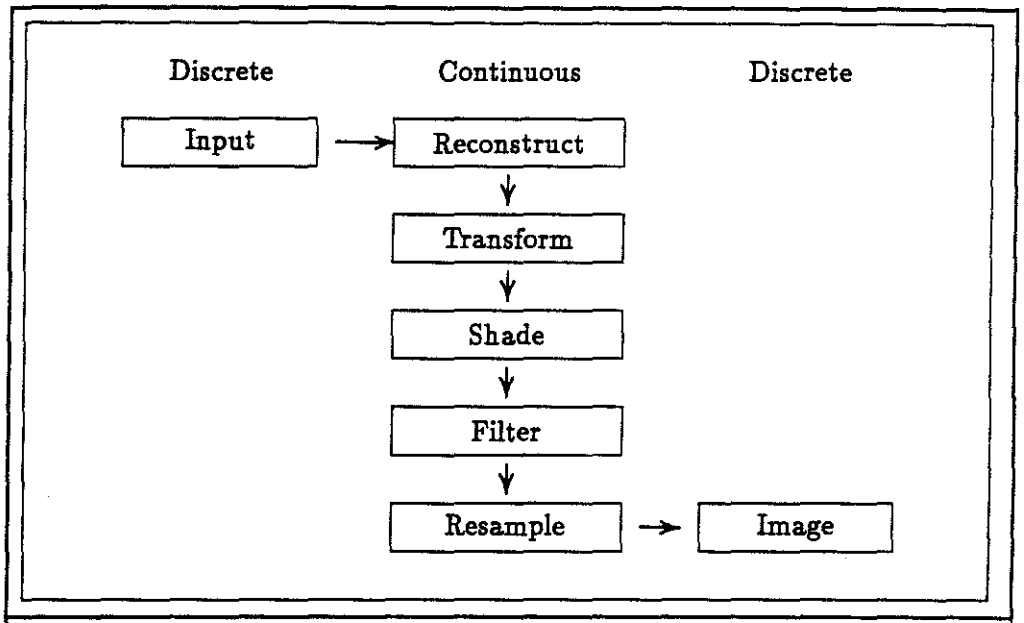


Figure 2.01: Idealized Volume-Rendering Process.

The renderer first transforms the discrete input into a continuous signal and then transforms the continuous function into image space in preparation for the eventual image-space sampling. The shading process must occur before the resampling process, because the nonlinearities of the shading process may introduce high frequencies into the signal's spectrum and increase the signal's Nyquist rate so that it is above the resampling rate. Once the renderer shades the continuous signal, the renderer must filter it to band-limit the signal to one half image resampling rate. After the renderer has filtered the signal, the renderer can sample the signal at image locations (typically pixels) without aliasing.

This chapter will discuss sampling theory and the assumptions volume-rendering researchers make to apply this theory. It will describe convolution, the mechanism of reconstruction, in detail to explore efficient convolution methods. In addition, it will discuss the connection between reconstruction and interpolation and describe the frequency response of common interpolation functions. Finally, the chapter describes the four basic volume-rendering methods in terms of their interpolation functions.

2.2 Sampling and Reconstruction

Signal processing and digital signal processing have their roots in 17th and 18th century mathematics. These techniques are indispensable in the electronics and computer fields. Today's digital computers store signals, images, and other representations of continuous functions as a collection of samples that are discrete in both space and representation. Algorithms designed without regard for signal processing principles, to minimize the

number and size of errors when working with these discrete functions, are *ad-hoc* at best.

2.2.1 Sampling

Sampling is the process of generating a regular collection of discrete data values from a continuous signal, illustrated in Figure 2.02. In one dimension, a continuous function $f_{input}(x)$ denotes the continuous signal. The comb function is a train of evenly spaced delta functions

$$c(x) = \sum_{n=-\infty}^{\infty} \delta(x - n).$$

Sampling is expressed mathematically as multiplication of the original signal, $f_{input}(x)$, by the comb function, $c(x)$,

$$\hat{f}_{sampled}(x) = f_{input}(x) \times c(x).$$

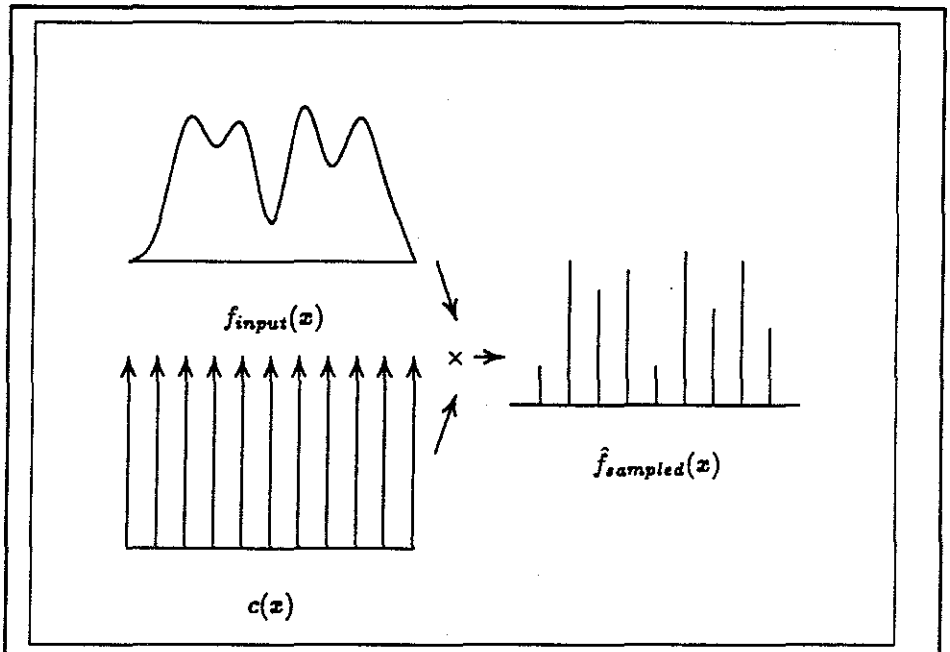


Figure 2.02: Sampling in the Spatial Domain.

By the convolution theorem, multiplication in the spatial domain corresponds to convolution, as defined in section 2.5, in the frequency domain [Castleman 79, page 169]. Therefore, sampling, which is multiplication in the spatial domain, can also be expressed as convolution in the frequency domain, as shown in Figure 2.03. Processes are often easier to visualize and interpret in one domain than the other and the above property allows researchers to alternate from one domain to the other and work with the more convenient representation. Let $F(\nu)$ be the Fourier transform of $f(x)$ and $C(\nu)$ be the Fourier transform of $c(x)$, then

$$\hat{F}_{sampled}(\nu) = F_{input}(\nu) \star C(\nu).$$

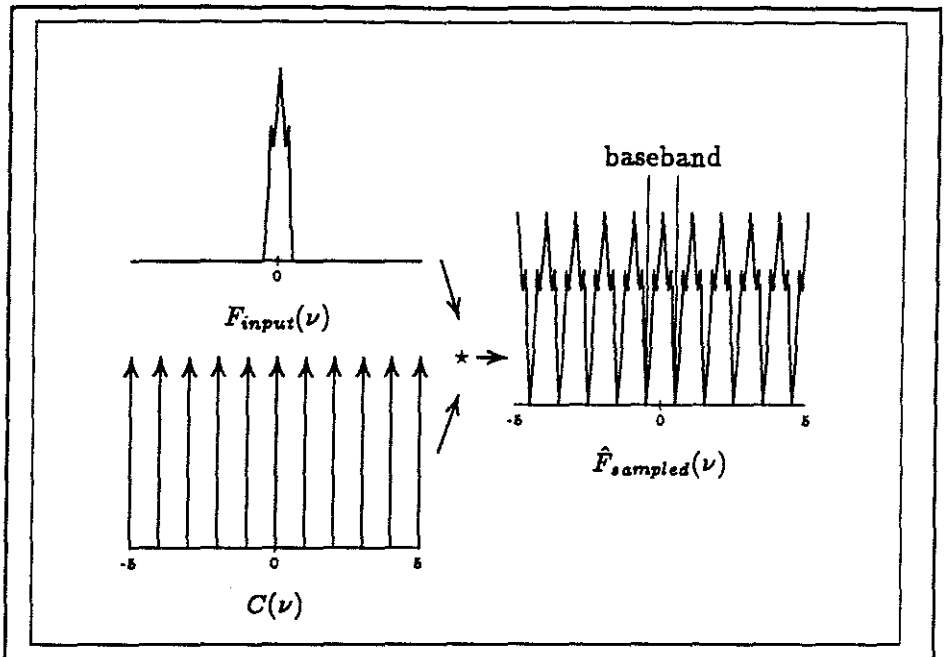


Figure 2.03: Sampling in the Frequency Domain.

Convolution of F_{input} with a comb function replicates F_{input} at each harmonic of the comb pulse rate, as seen in Figure 2.03.

$\hat{F}_{sampled}(\nu)$ contains two parts: the baseband spectrum, which consists of an exact copy of the original spectrum, and the harmonic components which make up the rest of the replicated spectra, Figure 2.04.

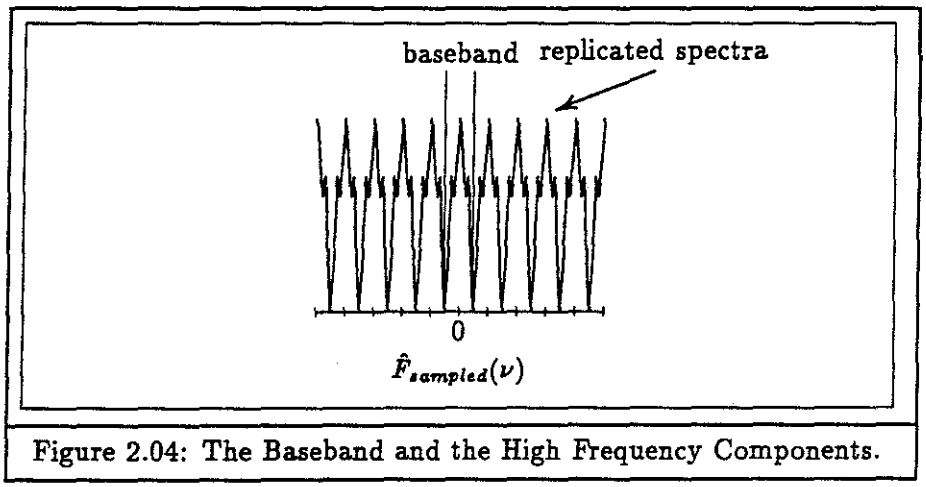


Figure 2.04: The Baseband and the High Frequency Components.

The Fourier transform of the comb function is also a comb function with reciprocal spacing of pulses. Thus, if function $c()$ has a period of τ, its Fourier transform, $C()$, will have a spacing of $\frac{1}{\tau}$. The closer the pulses lie in the spatial domain the further apart

the pulses will lie in the frequency domain—for example, for an original sampling rate of one, $c()$ is shown in Figure 2.05.

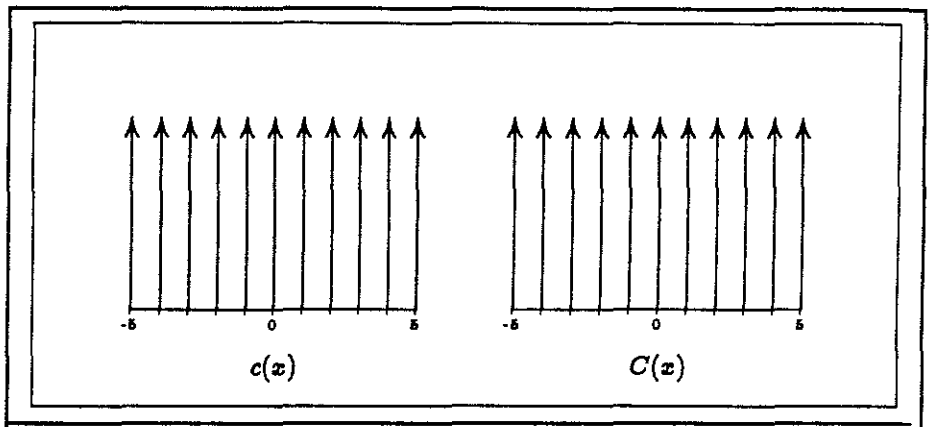


Figure 2.05: Initial Sampling Rate.

If $c'()$ has one-half the period of $c()$, then $C'()$ will have twice the spacing of $C()$, as shown in Figure 2.06.

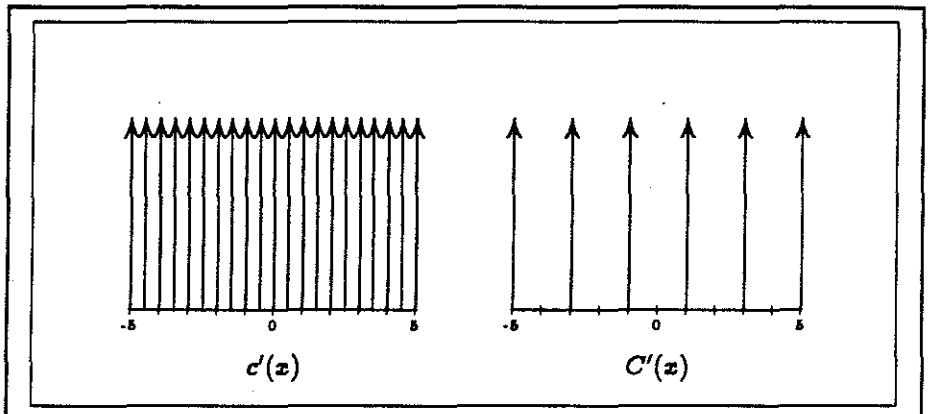


Figure 2.06: Double the Initial Sampling Rate.

2.2.2 Reconstruction

Reconstruction is the process of generating a continuous signal from a set of samples, as illustrated in Figure 2.07. Convolving the samples, $\hat{f}_{sampled}(x)$, with a reconstruction kernel $h(x)$, produces a function defined at all x

$$f_{reconstructed}(x) = \hat{f}_{sampled}(x) \star h(x).$$

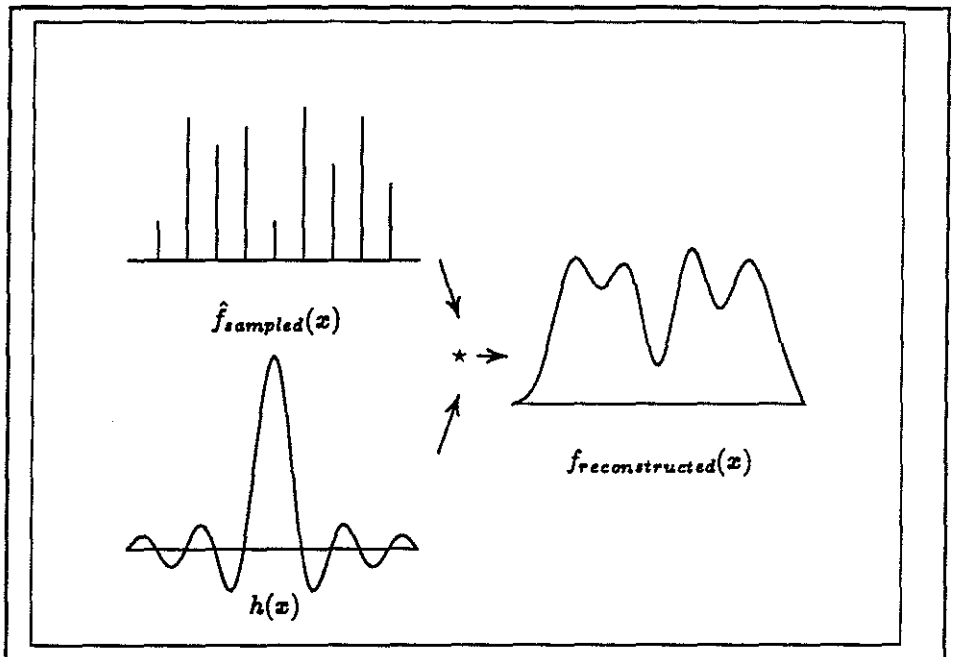


Figure 2.07: Reconstruction in the Spatial Domain.

In the same way that sampling may be expressed in either the spatial domain or the frequency domain, reconstruction, which is convolution in the spatial domain, may also be expressed as multiplication in the frequency domain, as shown in Figure 2.08. Interpreting the result of reconstruction by different kernels is significantly easier in the frequency domain than the spatial domain because multiplication is much easier to visualize than convolution [Castleman 79, page 169]. Let $F(\nu)$ be the Fourier transform of $f(x)$ and $H(\nu)$ be the Fourier transform of $h(x)$, then

$$F_{reconstructed}(\nu) = \hat{F}_{sampled}(\nu) \times H(\nu).$$

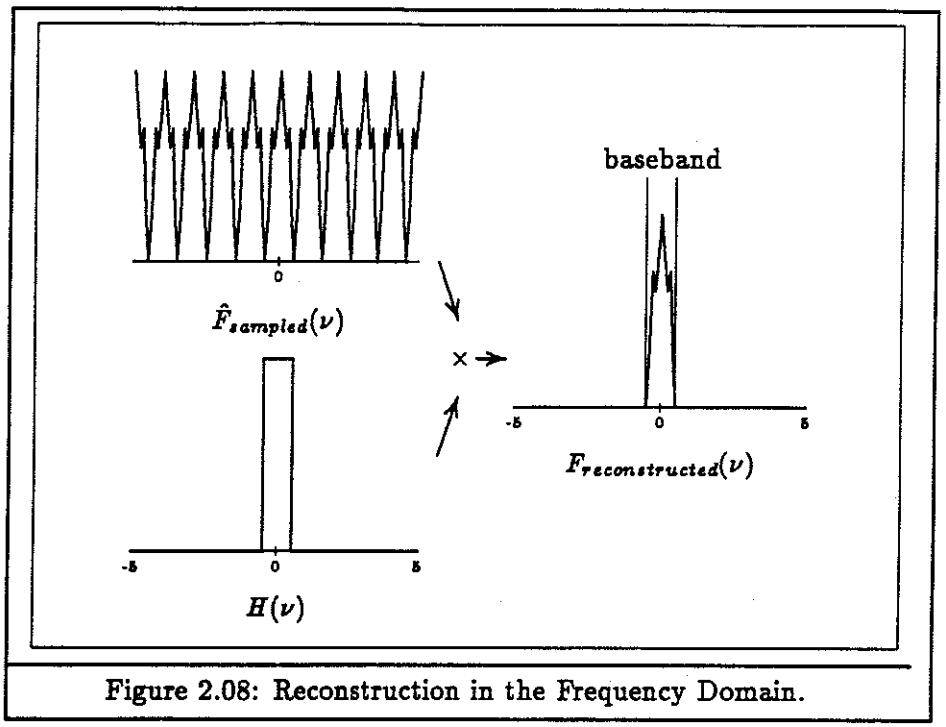


Figure 2.08: Reconstruction in the Frequency Domain.

2.2.3 Ideal Reconstruction

The goal of reconstruction is to isolate the baseband spectrum from the replicated spectra in $\hat{F}_{sampled}(\nu)$. The original sampling process must satisfy two criteria for the reconstruction process to be able to reconstruct the original input signal. First, the input must be band-limited. If the signal has a spectrum that is zero everywhere except in a specific finite range, then it is band-limited.

$$\exists \quad \nu_0 \quad \ni \quad \hat{F}_{sampled}(\nu) = 0 \quad \forall \quad |\nu| > \nu_0.$$

Second, the sampling frequency must be greater than the input's Nyquist rate. If $F_{input}(\nu)$ contains no frequencies greater than one half the sampling rate, it is theoretically possible to reconstruct $F_{input}(\nu)$ from $\hat{F}_{sampled}(\nu)$ exactly. If the sampling process fails to meet either of these conditions, the replicas of the input spectrum overlap in the frequency domain, as shown in Figure 2.10, causing the baseband and the replicated spectra to mix. The extraneous high-frequency information from the replicas is indistinguishable from the baseband information. This high-frequency energy is said to *alias* to low-frequency energy. When the input signal violates the first condition, the spectrum of the input signal has an infinite extent and cannot be sampled without aliasing. When the sampling process violates the second condition, the result is an undersampled signal.

If the sampling process satisfies the above two criteria, the reconstruction process can exactly reconstruct the original input. In the frequency domain, the ideal reconstruction process multiplies $\hat{F}_{sampled}(\nu)$ with $H(\nu)$ defined as

$$H(\nu) = \begin{cases} 1, & \text{if } |\nu| < \text{Nyquist rate;} \\ 0, & \text{if } |\nu| = \text{Nyquist rate;} \\ 0, & \text{otherwise.} \end{cases}$$

$H(\nu)$ is called the *rectangular* or *rect* filter and is illustrated in Figure 2.09. It is an ideal low-pass filter which will suppress all spectral replicas in $F_{sampled}(\nu)$ except the baseband spectrum. The passband is all frequencies whose absolute value is below the maximum frequency of a band-limited signal. The stopband is all frequencies whose absolute value is above the maximum frequency [Oppenheim 83, page 401]. The ideal low-pass filter will pass all frequencies in the passband without distortion and completely suppress all frequencies in the stopband. The spatial equivalent of the frequency domain rect filter is the Fourier transform of $H(\nu)$, which is the sinc function illustrated in Figure 2.09. The sinc function is defined as

$$sinc(x) = \frac{sin(\pi x)}{(\pi x)}.$$

Multiplication in the frequency domain by the rect function is equivalent to convolution in the spatial domain with the sinc function. The sinc function has an infinite extent, which is a problem for the practical use of the ideal low-pass filter in the spatial domain, as discussed in section 2.7.1. Convolution with a filter of infinite extent is troublesome for computers, since it requires an infinite amount of work, and truncating the filter to a finite extent has adverse effects on the filter's passband and stopband performance. Consequently, no practical implementation of a low pass filter in the spatial domain exhibits ideal behavior.

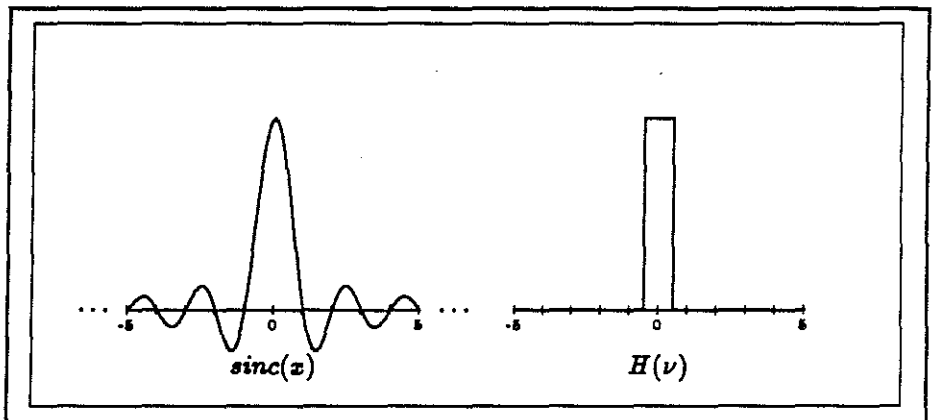


Figure 2.09: Ideal Low-Pass Filter.

2.3 Sampling and Reconstruction Errors

Deviation from the ideal in both the sampling and reconstruction processes causes errors in the result. Since sampling and reconstruction are central to the volume-rendering process, sources of error at each step must be identified so that these errors may be minimized. Choice of sampling rate and reconstruction kernels significantly affect the types and amounts of these errors.

2.3.1 Sampling Errors

Once the original signal has been sampled, frequency information above one half the sampling rate aliases as low-frequency information and the reconstruction process cannot distinguish it from the energy of the baseband spectrum. This extraneous energy causes

Moiré patterns and jagged edges [Mitchell 88]. The reconstruction process cannot remove these effects since the sampling process modified the original information. The system must address these effects during the original sampling process in one of two ways. It may sample at a higher rate, shifting apart the spectral replicas and thereby removing any overlap, or at least reducing the amount of overlap. Alternatively, the system may filter the original input before sampling to remove the high frequencies so there is no overlap of spectral replicas after sampling.

In Figure 2.10, the sampling process has undersampled the original signal and the input signal's spectral replicas overlap.

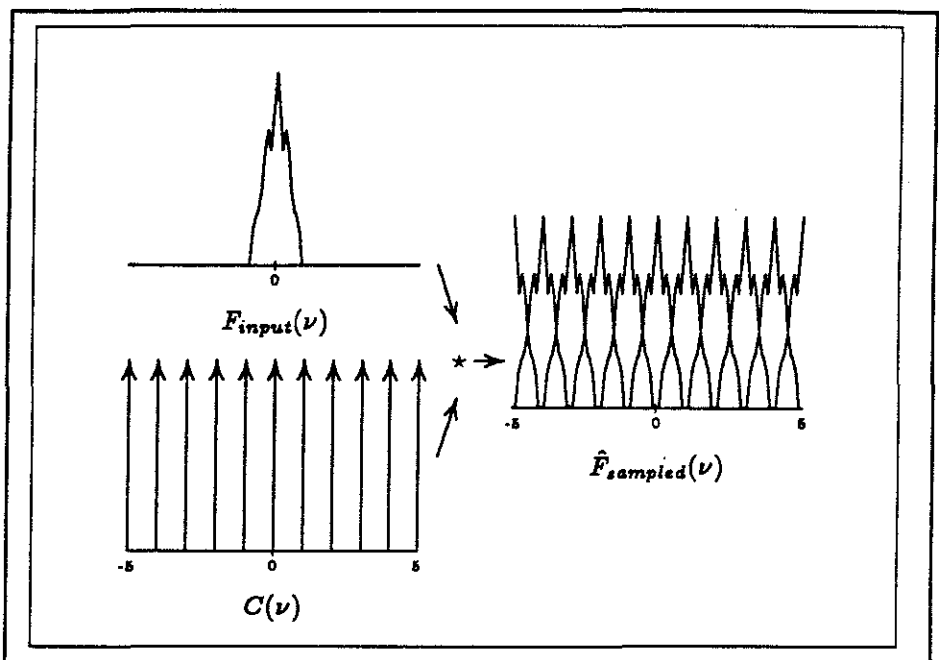


Figure 2.10: Initial Sampling Rate allows Replica Overlap.

In Figure 2.11, the spatial sampling is doubled which spreads out the comb function's pulses in its Fourier transform far enough so the replicas do not overlap.

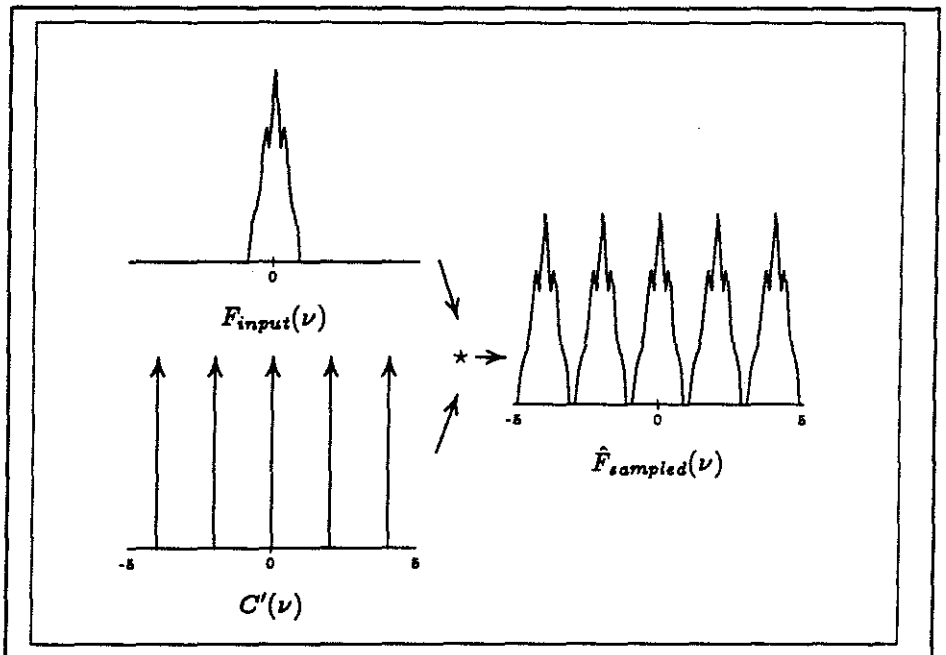


Figure 2.11: Doubling Sampling Rate removes Replica Overlap.

In Figure 2.10, the original signal has frequencies above one half the sampling rate and the replicas overlap after sampling. Instead of increasing the sampling rate, the input may be prefiltered to remove the high-frequency energy, as shown in Figure 2.12.. Figure 2.12 illustrates the result of prefiltering the input spectrum with a Bartlett window. Now the input spectrum does not have the high-frequency energy that causes overlap.

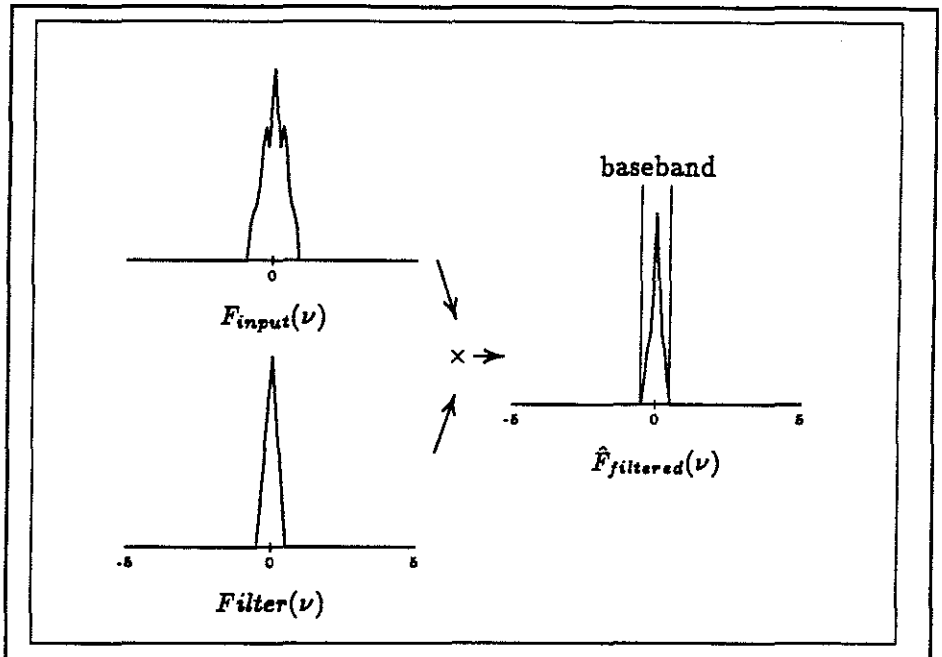


Figure 2.12: Bartlett Window as a Low-Pass Filter.

Convolving the filtered signal with the comb function produces a result with no overlap, but with a distorted baseband, as shown in Figure 2.13.

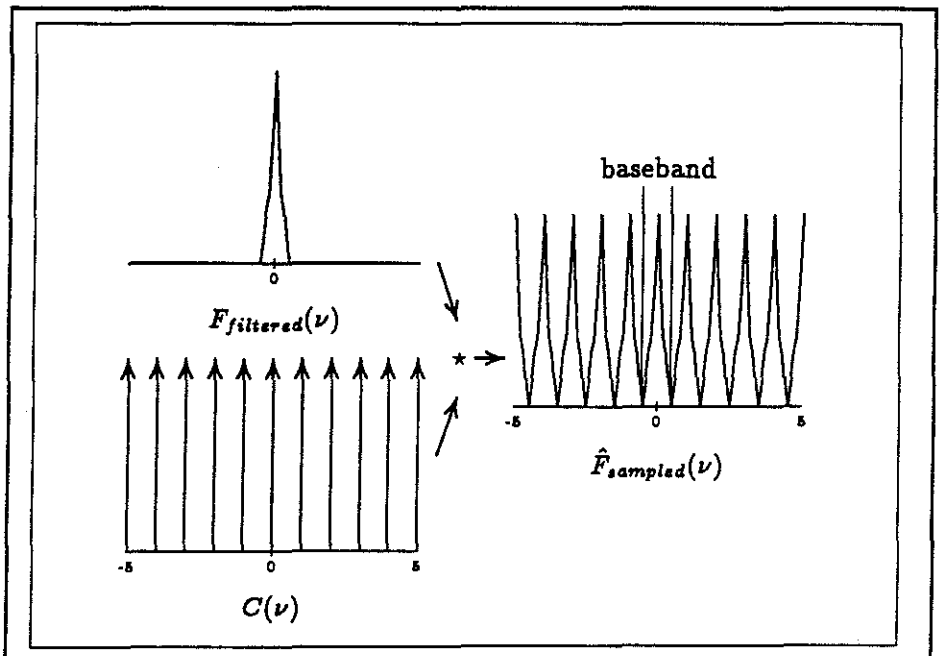


Figure 2.13: Spectrum has No Replica Overlap, but Baseband is Distorted.

2.3.2 Reconstruction Errors

Poor selection of the reconstruction kernel may cause two other artifacts. The first artifact arises when the reconstruction kernel distorts the baseband. The second artifact arises when the reconstruction kernel passes frequencies in the stopband. If the kernel distorts the baseband, it modifies the reconstructed signal, as shown in Figure 2.14. If the kernel passes frequencies in the stopband, it may collect energy from other replicas even when these replicas do not overlap, as show in Figure 2.15. This introduces high frequencies in the reconstructed signal that were not originally present.

In Figure 2.14, the reconstruction process multiplies the sampled signal's spectrum by the frequency domain triangle pulse. The multiplication deforms the resulting spectrum in the baseband.

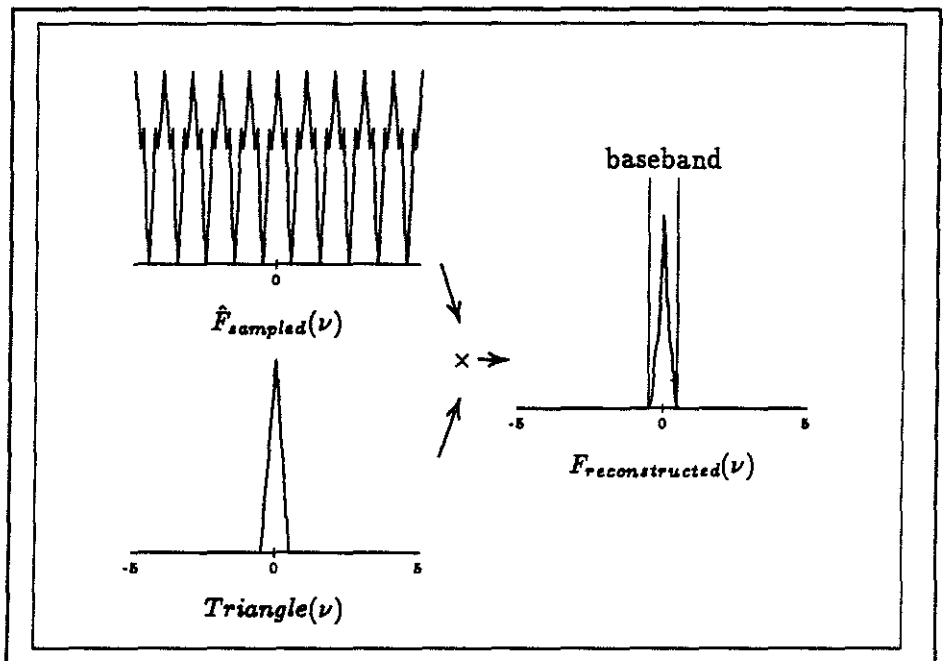


Figure 2.14: Distortion due to a Triangle Reconstruction Kernel.

In Figure 2.15, the reconstruction process multiplies the sampled signal's spectrum by the frequency domain equivalent to the spatial triangle filter. Since the triangle filter is the result of convolving two rect functions, the Fourier transform of the triangle is the product of two *sinc*() functions, which is $sinc^2()$. The lobes of the kernel which extend beyond the baseband allow energy in the stopband to pass to the resulting spectrum. As we shall see in section 2.7.3, reconstruction with the spatial triangle filter is of particular interest in volume rendering since it corresponds to the linear interpolation steps found in several conventional display algorithms.

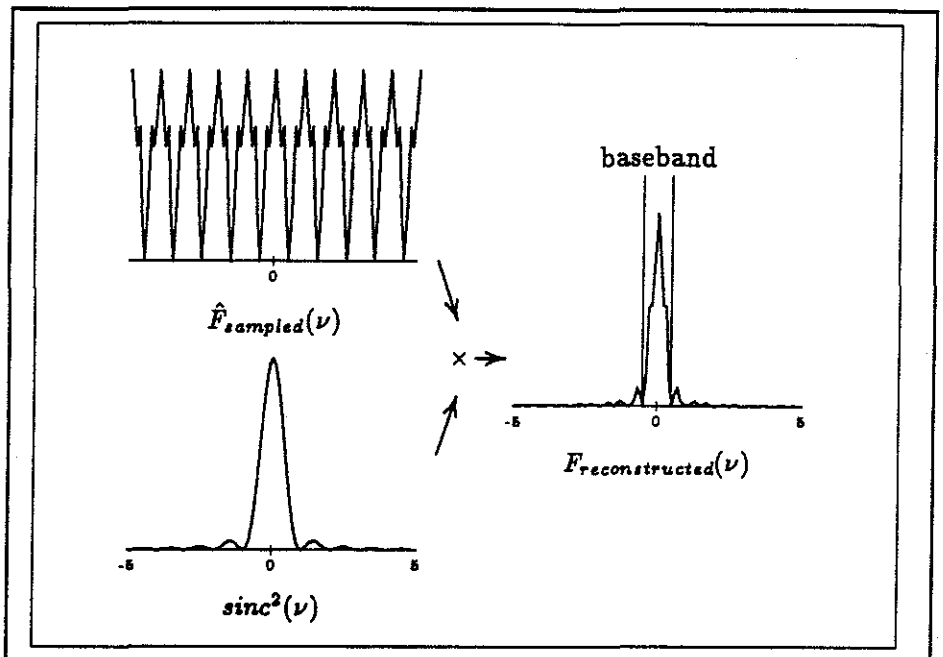


Figure 2.15: Distortion due to a Bartlett Reconstruction Kernel.

2.4 Volume Rendering Assumptions and Artifacts

Researchers must understand signal processing techniques to develop effective volume-rendering methods because reconstruction of a continuous volume from the sampled volume is an essential step before the data can be resampled as an image. Even if the step is omitted, it is still implicitly a part of the operation because the theory demands it. Furthermore, to use these techniques, researchers make assumptions about their data.

Most volume-rendering algorithms assume that the input signal is band-limited and that the original signal was properly sampled. The signal was either known to be band-limited or it was correctly filtered to lower the Nyquist rate of the input so that it is below the sampling rate. If this is not true, the sampled data is corrupt and the volume-rendering process cannot correct the situation.

Although the original samples may accurately represent the original signal, many volume-rendering methods classify and shade the samples first and then try to reconstruct these samples, a second source of errors in volume rendering. The CRIO process is often a nonlinear process which may introduce high frequencies into the sampled data as shown in Figure 2.20 and Figure 2.24.

Since the CRIO process may be any arbitrary function, illustrated in Figure 2.16, the result of the CRIO process on a sample's spectrum is hard to characterize. Instead we will look at two example CRIO processes. The first example compares the results of a binary and a non-binary classifier on a simple input signal. The difference in classifiers can have a dramatic effect on the output signal's spectrum, as shown in Figure 2.20 and Figure 2.21. The second example illustrates the effect on an output sample's spectrum of using the

local gradient in the CRIO process. The Fourier transform of the gradient operator is $j\omega$, a high-pass filter, which damps low frequencies and enhances high frequencies. These enhanced high frequencies in the output will cause aliasing during the resampling process if they are not reduced or removed during a filtering step.

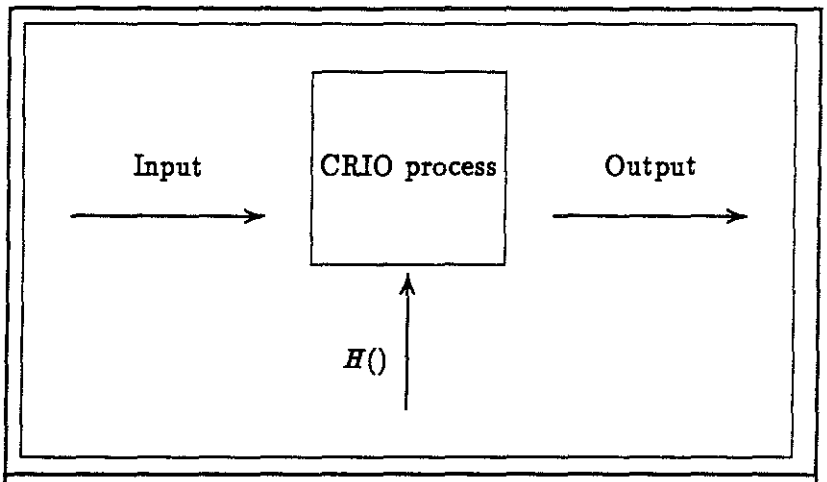


Figure 2.16: Classifying Process.

In Figure 2.17 through Figure 2.21, both a binary classifier process and a non-binary classifier process classify an input signal. Both select the region where the input lies between 0.3 and 0.7. The binary classifier decides "yes" or "no" whether to include that point in the output at every sample. The binary classifier introduces sharp transitions in the output function and introduces high frequencies in the output spectrum. Images of binary classified data typically have many rendering artifacts because of these high frequencies [Frieder 85]. The non-binary classifier does not make "yes" or "no" decisions, but smoothly transitions from a "no" decision to a "yes" decision with varying degrees of acceptance in-between. This classifier generates soft edges, not sharp transitions, in the output and introduces significantly fewer high frequencies.

Figure 2.17 shows the example input signal. This signal is

$$f_{input}(x) = \begin{cases} 0.5 + \frac{cos(\frac{x\pi}{5.0})}{2.0}, & \text{if } -5 < x < 5; \\ 0, & \text{otherwise.} \end{cases}$$

The signal does contains high frequencies, but they have near zero amplitude.

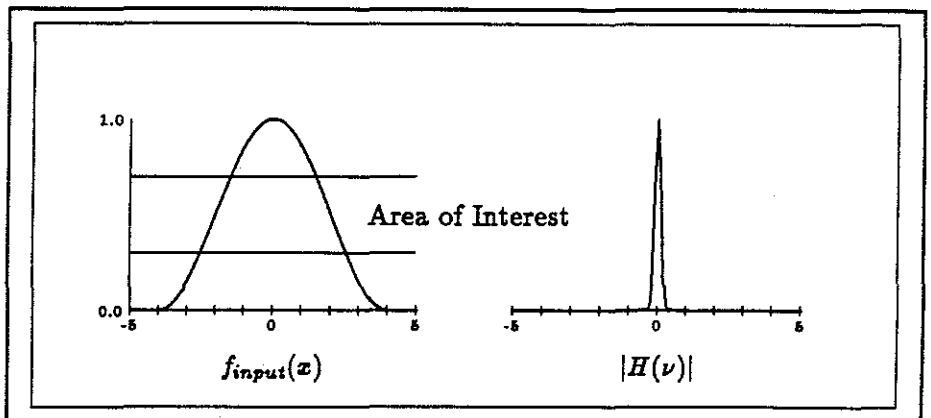


Figure 2.17: Band-limited Example Signal.

Figure 2.18 is an example of a binary classifier. It classifies input in the range 0.3 to 0.7 as part of the object of interest. This classifiers is almost the rect function, but is actually trapezoidal in shape as the sides are not vertical. Notice the sharp corners in the classifier and the resulting high frequencies in its spectrum.

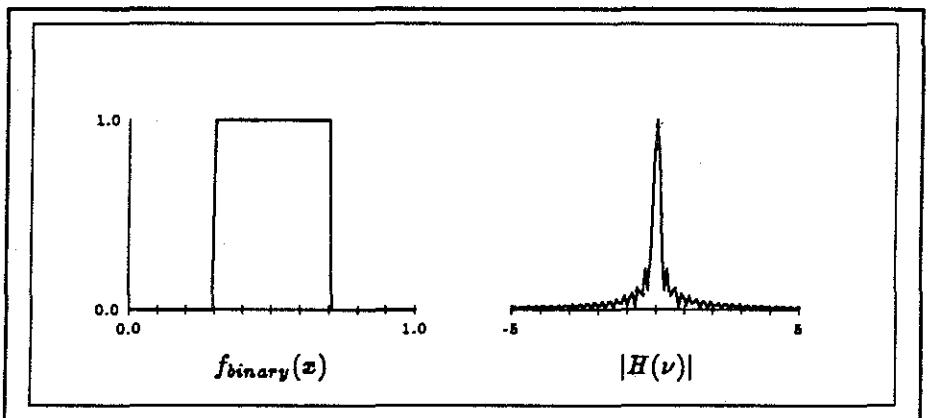


Figure 2.18: Binary Classifier.

Figure 2.19 is an example of a non-binary classifier. It classifies input in the range 0.3 to 0.7 as part of the object of of interest, but unlike Figure 2.18, this classifier as smooth transitions from the region of interest to the region of no interest. Notice the lack of sharp corners in the classifier and the resulting absence of high frequencies in its spectrum.

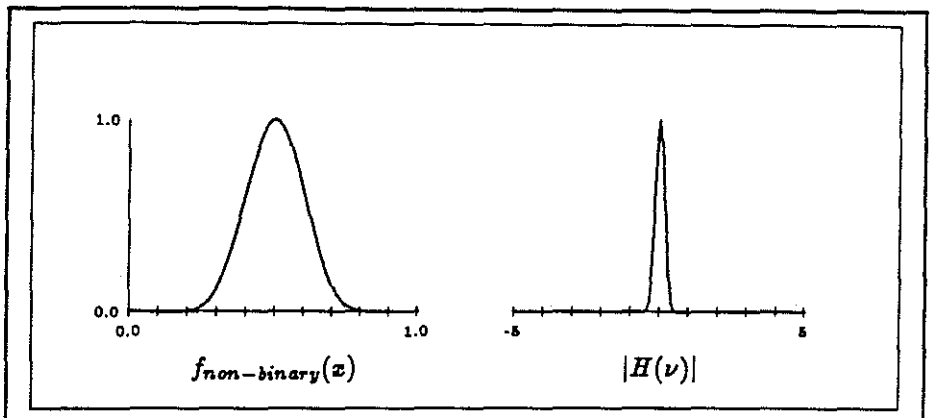


Figure 2.19: Non-binary Classifier.

Figure 2.20 illustrates the result of the binary classifier of Figure 2.18 operating on the input from Figure 2.17. Notice the sharp corners in the output and the resulting high frequencies in the output spectrum.

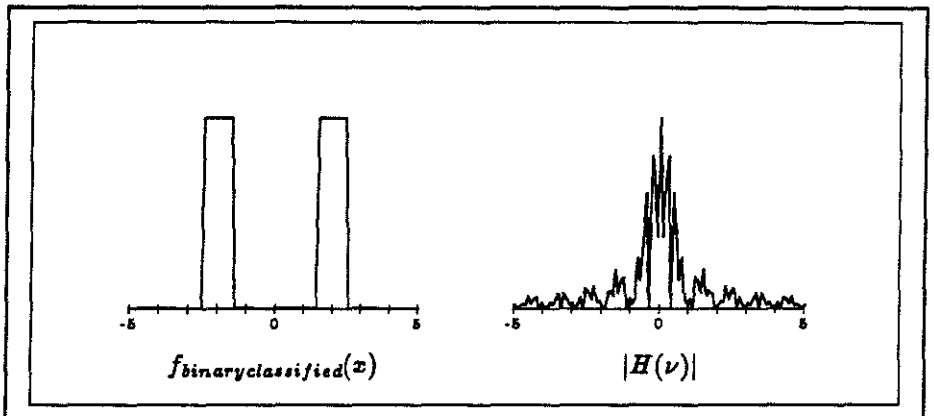


Figure 2.20: Result of Binary Classification.

Figure 2.21 shows the result of the non-binary classifier of Figure 2.19 operating on the input from Figure 2.17. Notice the absence of sharp corners in the output and the resulting absence high frequencies in the output spectrum. While some high frequencies remain, there are not nearly so many as in Figure 2.20.

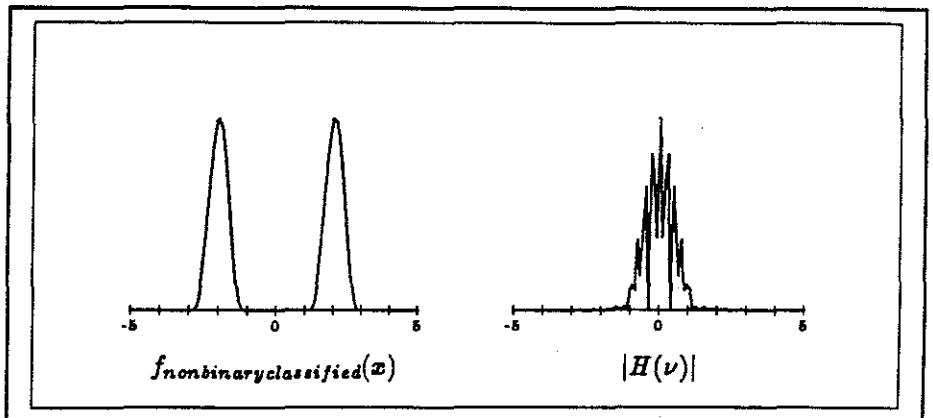


Figure 2.21: Result of Non-binary Classification.

In Figure 2.22 through Figure 2.24, a gradient shader process shades the example input signal of Figure 2.17. The system modulates the intensity of the output by the strength of the gradient raised to a power, much like Phong shading [Phong 75]. Phong shading is a common technique used in volume rendering to give the user more three-dimensional shape cues [Levoy 88]. The output has many high-frequency components that are not present in the input. The exponent used in Phong shading controls the width of the highlight with higher powers narrowing the highlight, producing the impression of more mirror-like surfaces. Higher powers introduce more high frequencies than lower powers, because higher powers narrow the region of the highlight and cause sharper changes in the brightness of the image.

Consider the input from Figure 2.17. The absolute value of its derivative (one-dimensional gradient) is shown in Figure 2.22.

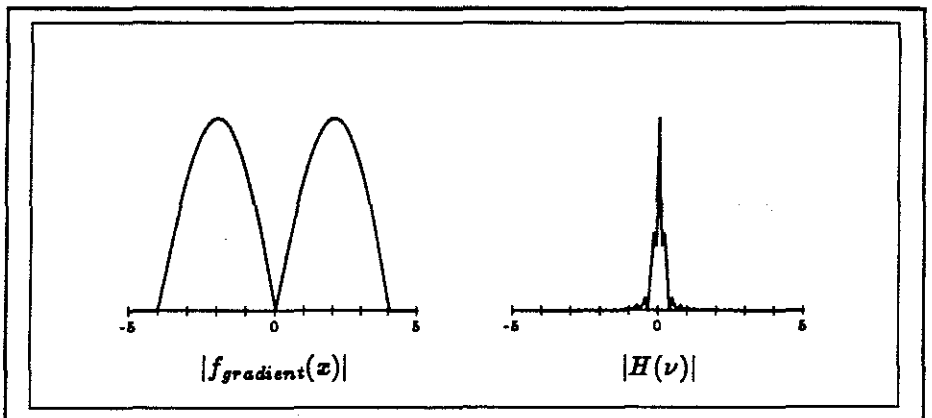


Figure 2.22: Gradient of Input.

Raising the result of the gradient operator to a power, accentuates the region where the input is changing most rapidly. The result of raising the signal in Figure 2.22 to the $32nd$ power is shown in Figure 2.23.

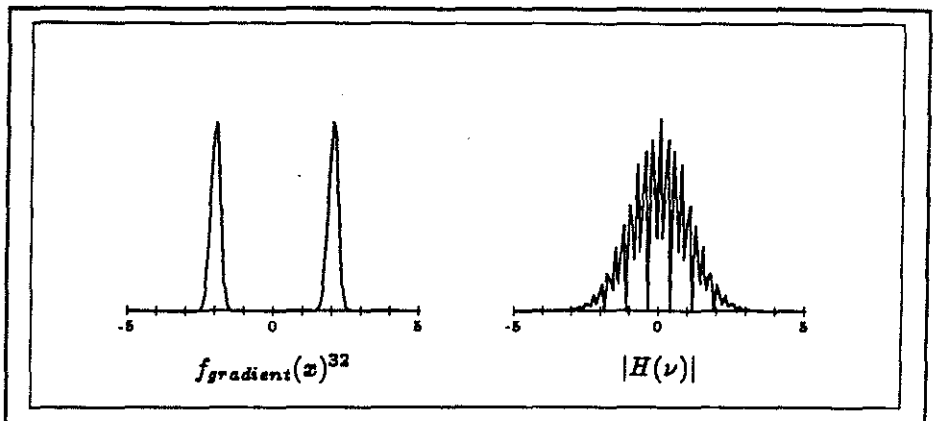


Figure 2.23: Gradient of Input raised to the 32*nd* Power.

Now the input signal of Figure 2.17 is multiplied by the raised gradient to select the region of the input changing most rapidly. This result is shown in Figure 2.24. Notice the high-frequency components introduced by the nonlinear gradient operator.

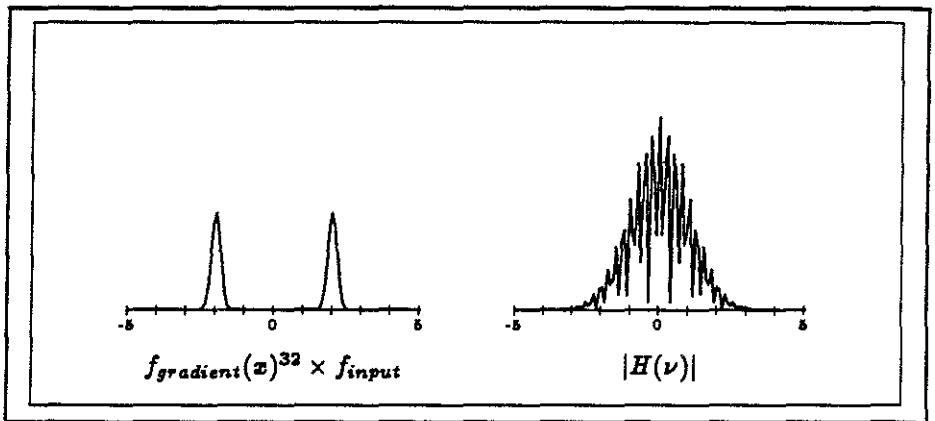


Figure 2.24: Result of Multipling Original by Gradient.

Another common volume-rendering assumption is that the image resampling rate is greater than the original sampling rate. There are more output samples than input samples, thus magnifying the input. For example, renderers often generate 512 x 512 images from 256 x 256 x 256 or smaller input data. This eliminates the need to filter the reconstructed signal, because a properly chosen reconstruction kernel will not introduce frequencies that are above one half the resampling rate. However, the shading process may introduce such frequencies.

2.5 Convolution

While visualizing reconstruction is easier in the frequency domain, since it is expressed as multiplication rather than convolution, most renderers operate in the spatial domain. For discrete kernels of limited extent, transforming from the spatial domain to the frequency domain, performing the multiplication, and transforming back to the spatial

domain may be more compute-intensive than just doing the discrete convolution in the spatial domain. For discrete kernels with a large extent, the convolution will be more compute-intensive. Volume-rendering algorithms typically convolve the samples and the reconstruction kernel in the spatial domain, since they commonly use small kernels.

Convolution is expressed mathematically by the convolution integral. Let $g()$ denote the output signal, $f()$ denote the input signal, and $h()$ denote the filter function.

In one dimension, the convolution integral is

$$g(t) = \int_{-\infty}^{\infty} f(u)h(t-u)\,du.$$

Generalizing to two dimensions, the convolution integral is

$$g(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(u,v)h(x-u,y-v)\,du\,dv$$

and in three dimensions, the convolution integral is

$$g(x,y,z) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(u,v,w)h(x-u,y-v,z-w)\,du\,dv\,dw.$$

For a discrete signal, $f(i,j,k)$, this becomes

$$g(x,y,z) = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty}\sum_{k=-\infty}^{\infty} f(i,j,k)h(x-i,y-j,z-k)$$

called the *convolution sum* [Oppenheim 75, page 32].

In the equations above, convolution is viewed as the gathering of contributions from neighboring samples for each output sample. Alternatively, it may be viewed as the distribution of contributions from a given sample to its neighboring samples. The former views an output sample being the weighted average of the set of input samples that lie under the reconstruction kernel centered at the output sample. The latter views an input sample distributing its energy to the set of output samples that lie under the reconstruction kernel centered at the input sample. Both methods will produce identical results. Feed-backward methods gather information from the input for each output. Feed-forward methods spread information from each input to the output.

Figure 2.25 demonstrates feed-backward reconstruction [Feibush 80] with the triangular pulse, which is equivalent to linear interpolation. The renderer calculates each output value individually by centering the reconstruction kernel at a output value and calculating a weighted average of each input sample that lies under the kernel.

In pseudo code this operation is

```
For each output sample I
        Output sample I = 0
        For each input sample J under Kernel centered at I
                Output sample I += Input sample J * Kernel value at J
        End for
End for
```

For example, the renderer centers the kernel at A_1 and calculates A_1. When the kernel is centered at A_1, it has a value of $\frac{2}{3}$ at X_1 and a value of $\frac{1}{3}$ at X_2. The renderer then centers the kernel at A_2 and calculates A_2. When the kernel is centered at A_2, it has a value of $\frac{1}{3}$ at X_1 and a value of $\frac{2}{3}$ at X_2. The four steps are

$$A_1 = 0, \quad A_2 = 0 \tag{1}$$

$$A_1 = \frac{2}{3}(X_1) + \frac{1}{3}(X_2) = \frac{2}{3}(4) + \frac{1}{3}(2) \tag{2}$$

$$A_2 = \frac{1}{3}(X_1) + \frac{2}{3}(X_2) = \frac{1}{3}(4) + \frac{2}{3}(2) \tag{3}$$

$$A_1 = \frac{10}{3}, \quad A_2 = \frac{8}{3} \tag{4}$$

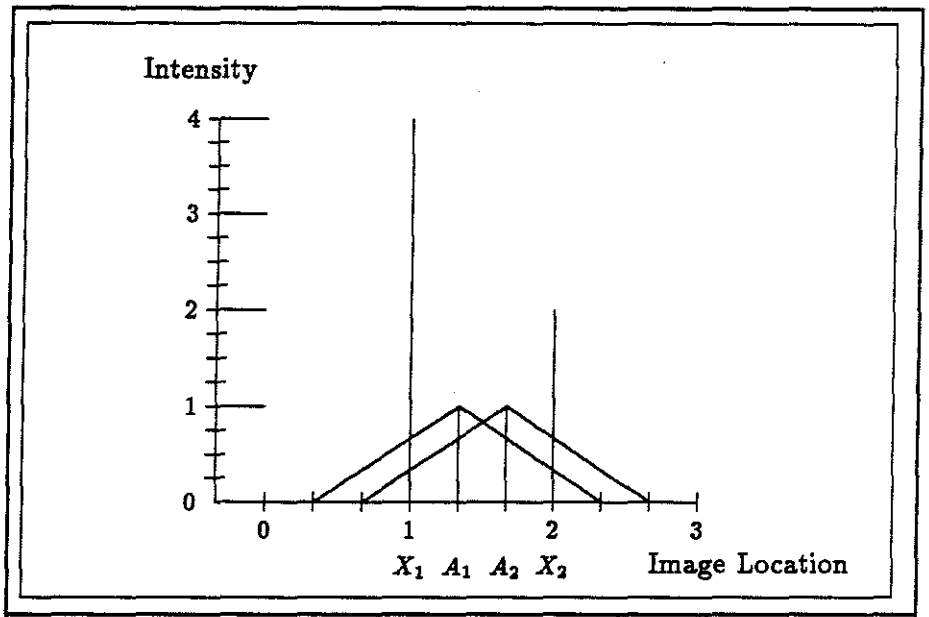


Figure 2.25: Feed-Backward Reconstruction.

Figure 2.26 demonstrates feed-forward reconstruction with the triangular pulse. The renderer incrementally calculates the outputs. It centers the reconstruction kernel at an input value and spreads the energy to each output that lies under the kernel.

In pseudo code this operation is

```
Zero all Output samples
For each input sample J
        For each output sample I under Kernel centered at J
                Output sample I += Input sample J * Kernel value at I
        End for
End for
```

For example, the renderer centers the kernel at X_1 and distributes its energy to A_1 and A_2. When the kernel is centered at X_1 it has a value of $\frac{2}{3}$ at A_1 and a value of $\frac{1}{3}$ at A_2. The renderer then centers the kernel at X_2 and distributes its energy to A_1 and A_2. When the kernel is centered at X_2 it has a value of $\frac{1}{3}$ at A_1 and a value of $\frac{2}{3}$ at A_2. When the renderer finishes processing all the input samples, the outputs have correct values. The four steps are

$$A_1 = 0, \quad A_2 = 0 \tag{1}$$

$$A_1 = A_1 + \frac{2}{3}(X_1) = 0 + \frac{2}{3}(4) \tag{2}$$

$$A_2 = A_2 + \frac{1}{3}(X_1) = 0 + \frac{1}{3}(4)$$

$$A_1 = A_1 + \frac{1}{3}(X_2) = \frac{8}{3} + \frac{1}{3}(2) \tag{3}$$

$$A_2 = A_2 + \frac{2}{3}(X_2) = \frac{4}{3} + \frac{2}{3}(2)$$

$$A_1 = \frac{10}{3}, \quad A_2 = \frac{8}{3} \tag{4}$$

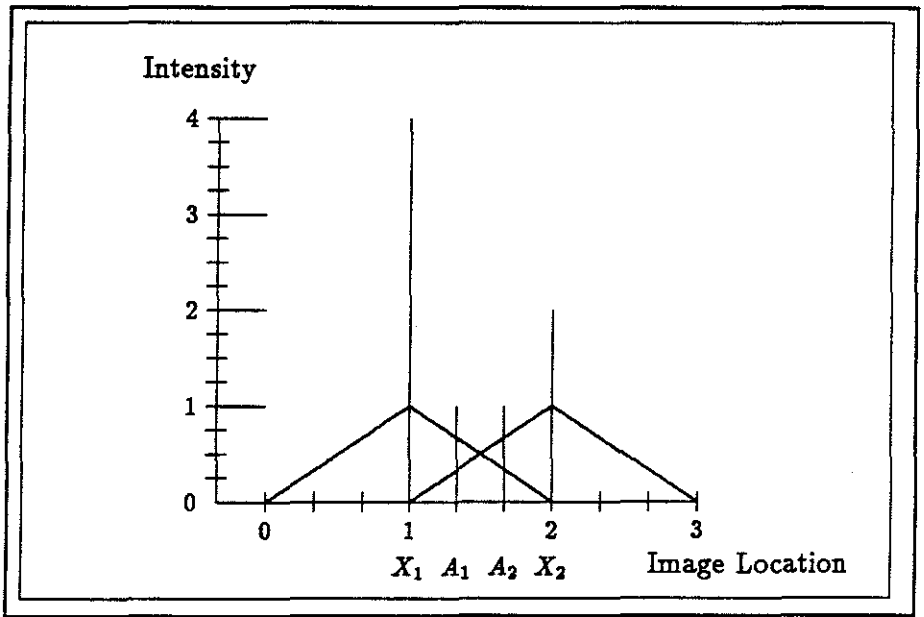


Figure 2.26: Feed-Forward Reconstruction.

Both the feed-backward and the feed-forward methods produce identical results: $A_1 = \frac{10}{3}$ and $A_2 = \frac{8}{3}$.

The primary difference between the two methods is how often they access each output and input sample. Feed-backward methods access an output sample once and an input sample many times. Feed-forward methods access an input sample once and an output sample many times. The number of input samples and output samples the two methods access is directly related to the extent of the reconstruction kernel and the number of intermediate samples the reconstruction and resampling process is generating. Thus, the total number of input and output accesses differs in the two methods. A renderer using linear reconstruction, generating output samples at three times the rate of input samples, as illustrated in Figure 2.25 and Figure 2.26, and having four input samples would calculate 13 output samples. For feed-backward reconstruction, the renderer would access two input samples for each output sample and access that output sample once, so there would be a total of 39 sample accesses. For feed-forward reconstruction, the renderer would access 5 output samples for each input sample and access that input sample once, so there would be 24 sample accesses.

Viewing convolution as generating outputs as a weighted average of inputs or an input distributing its energy to outputs is the primary difference in the ray-casting and the element-tossing approaches to volume rendering.

2.6 Interpolation

Interpolation is the process of generating intermediate in-between points from a sampled signal [Wolberg 90, page 124]. Interpolation processes fit a continuous function through the input samples and evaluate the function at the intermediate points. The process of finding intermediate values may be thought of as convolving a sampled signal with a reconstruction kernel, called *reconstruction*, or alternatively as directly evaluating the interpolated continuous function at the desired points, called *interpolation*. While reconstruction methods typically use feed-forward methods and interpolation methods typically use feed-backward methods, both these methods generate identical results. Most volume renderers contain no explicit reconstruction phase. Instead, they use interpolation methods to map volume data to image samples. Reconstruction and interpolation are two names for the same process.

Given this view, we need to analyze interpolation functions as reconstruction kernels. For example, the reconstruction kernel for linear interpolation is a triangular pulse whose width is twice the spacing of the samples. Interpolation, implemented as convolution, attempts to select the baseband spectrum of a sampled signal. When the interpolation function is the sinc function and the signal was properly sampled, interpolation can reproduce the original signal without error. Other interpolating functions fail to completely remove all other spectral replicas and distort the shape of the baseband, causing aliasing. Different choices for interpolation or reconstruction kernel cause different amounts and types of aliasing and distortion. Researchers often examine the quality of the different interpolation functions by studying the function's frequency response in the stopband and the passband as is done in section 2.7.

2.7 Interpolation Methods

In the frequency domain, the multiplication of the sample's spectrum by the reconstruction kernel should exactly select a signal's baseband. This section will describe eight common interpolation methods. It will show the one-dimensional version of the function in the spatial domain and its Fourier transform. The Fourier Transform is plotted linearly for easy comparison with the rect function. It is also plotted logarithmically, since the stopband response of these filters is small compared to the passband response and these effects are important for understanding the filter's response as a whole. These sections also show the result of using the interpolation method to reconstruct the four two-dimensional signals described in Figure 2.27 and below.

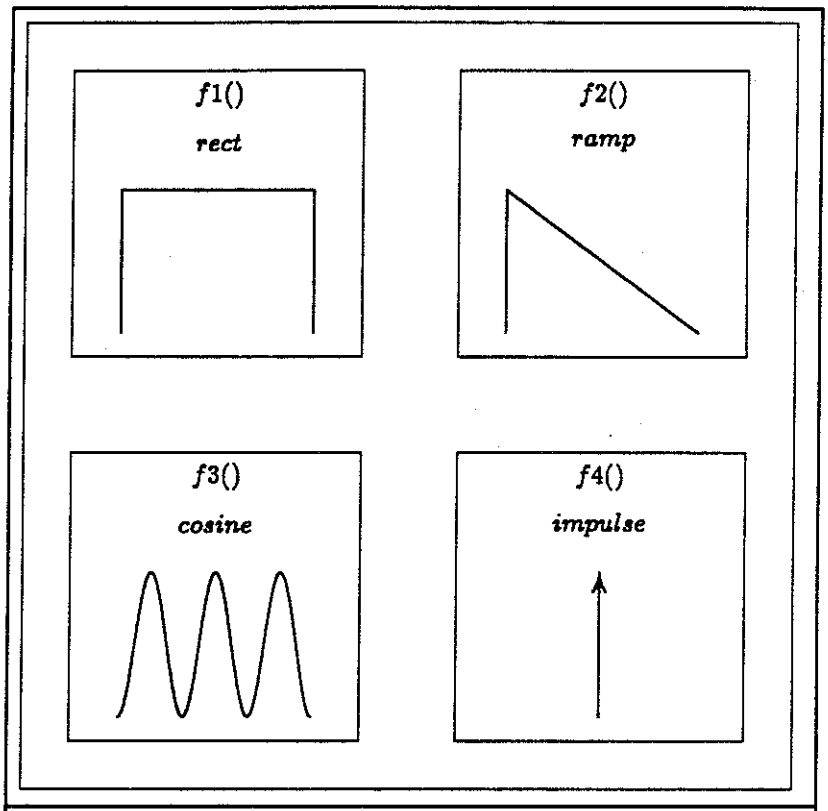


Figure 2.27: Four Example Reconstruction Inputs.

$f1()$ is a rect function defined as

$$f1(x) = \begin{cases} 1, & \text{if } |x| <= 10; \\ 0, & \text{otherwise.} \end{cases}$$

$f2()$ is a ramp function defined as

$$f2(x) = \begin{cases} 0, & \text{if } x < -10; \\ \frac{(11-x)}{21}, & \text{if } x >= -10 \ \& \ x <= 10; \\ 0, & \text{if } x > 10. \end{cases}$$

$f3()$ is a cosine defined as

$$f3(x) = \begin{cases} 0, & \text{if } |x| > 10; \\ \dfrac{1.0 - cos(2\pi \frac{(x+10)\times 3}{20})}{2.0} + 0.5, & \text{if } |x| <= 10. \end{cases}$$

$f4()$ is the impulse function defined as

$$f4(x) = \begin{cases} 1, & \text{if } x = 0; \\ 0, & \text{otherwise.} \end{cases}$$

The input signal is a mesh of 21 x 21 samples with x and y going from -10 to 10. The input is magnified by a factor of 16 to produce an image that is 345 x 345 pixels. Superimposed on each image is a cross section of the original function in green, a cross section of the reconstructed function in red, and the original sample points for a single row of samples in yellow.

2.7.1 Ideal Kernel

As noted in section 2.2.3, the ideal reconstruction kernel is a rect function in the frequency domain. The rect function may be a one-dimensional, two-dimensional, or three-dimensional pulse depending on the input sample's dimensionality.

The problem with the ideal filter is that the Fourier transform of the rect function is the sinc function with infinite extent, as illustrated in Figure 2.28. For the interpolator to use the ideal filter in the spatial domain, it must convolve the samples with the sinc function. The interpolator must truncate the sinc function in order to reduce the compute time. This truncation distorts the filter and it is no longer the ideal filter, as can be seen in Figure 2.30.

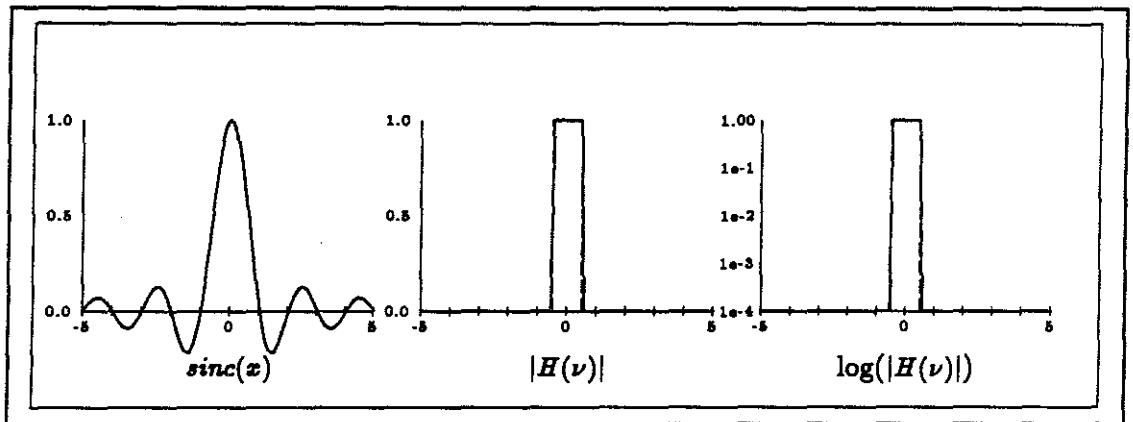


Figure 2.28: Spatial Domain Sinc Kernel in One Dimension.

Figure 2.29 illustrates sinc function reconstruction of the four example functions described above.

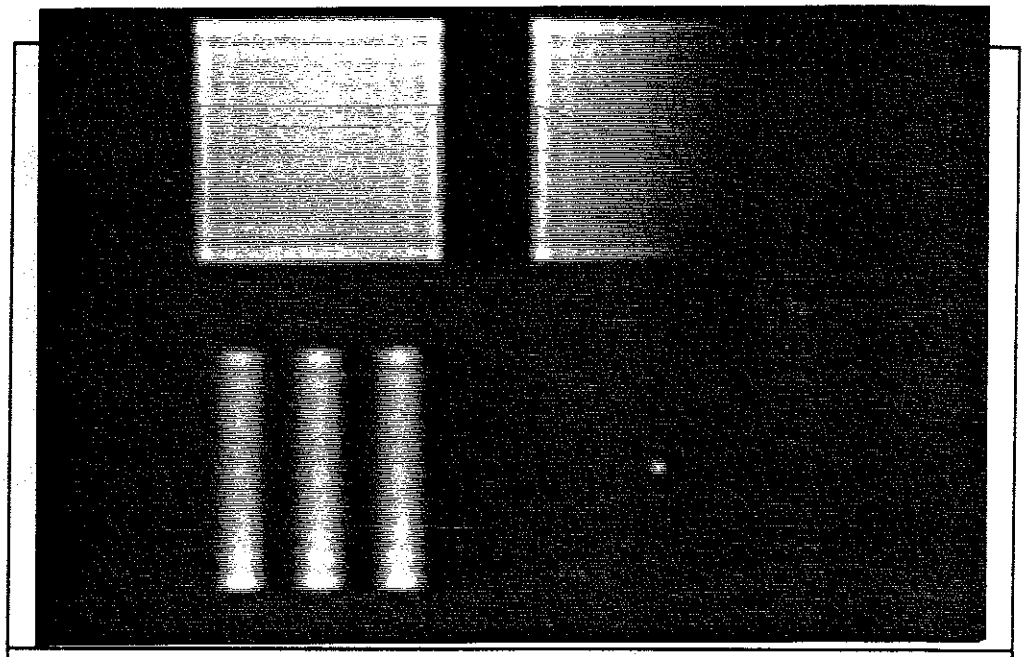


Figure 2.29: Spatial Domain Sinc Kernel Example.

Since the sinc function's bound asymptotically falls to zero, it is tempting to truncate the sinc function after a few cycles. A truncated sinc and its Fourier transform are illustrated in Figure 2.30. However, truncation in one domain produces ringing in the other domain, and the sharp cut-off in the spatial domain produces high-frequency leakage [Mitchell 88], as illustrated in Figure 2.30.

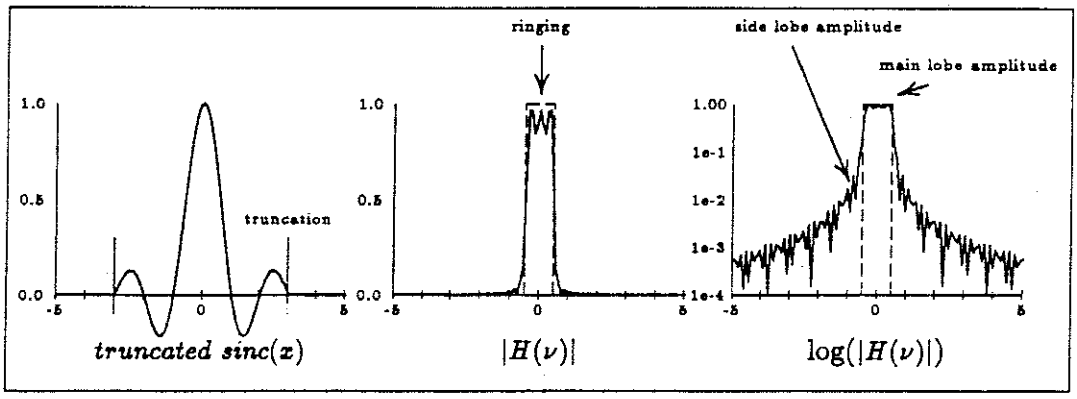


Figure 2.30: Spatial Domain Truncated Sinc Kernel in One Dimension.

Figure 2.31 illustrates truncated sinc reconstruction of the four example functions described above.

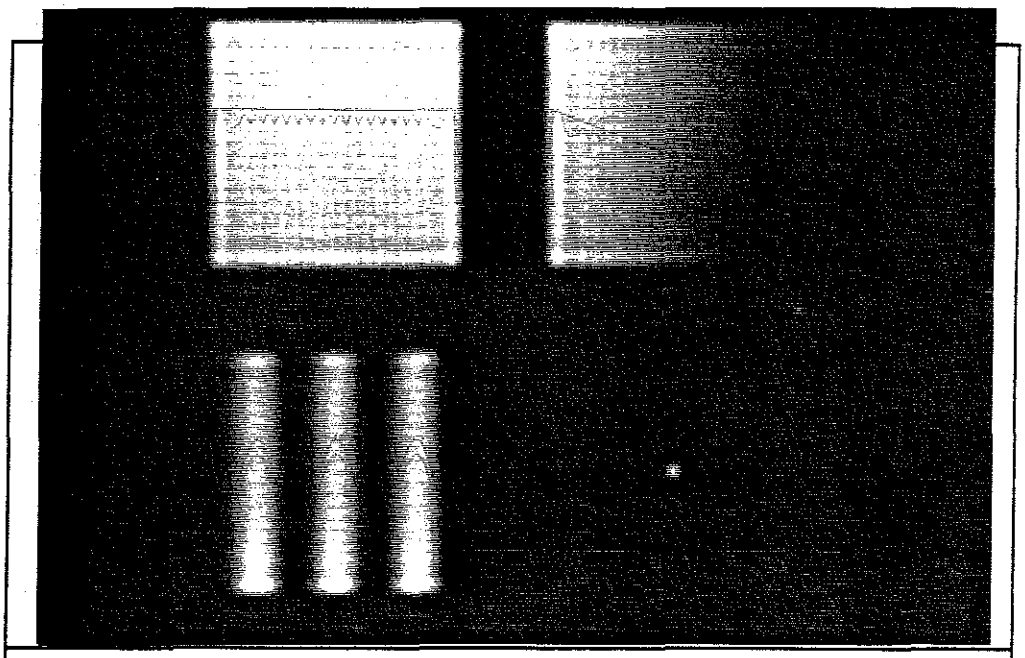


Figure 2.31: Spatial Domain Truncated Sinc Kernel Example.

While the spatial sinc filters is good in a theoretical sense, it is seldom used in practice. The early truncation of the sinc produces ringing, which causes the truncated sinc to have large ripple ratio, defined as the ratio of side lobe amplitude to main lobe amplitude [Mitchell 88], as illustrated in Figure 2.30. The interpolator truncates the sinc filter quickly because the width of the reconstruction kernel directly affects the amount of computation required to do convolution, yet this kernel require a large extent to avoid artifacts.

One way to address this problem is to window the sinc function with another function, by multiplying the sinc by a function that smoothly goes to zero at the desired truncation point [Oppenheim 75, page 239]. For example, the interpolator may multiply the sinc function by a Gaussian filter that goes near zero where it wants to truncates the sinc function. Oppenheim and Shafer [Oppenheim 75, page 242] discuss a family of nearly optimal windowing functions known as the *Kaiser* family of windowing function. They are optimal in the sense that they have the largest main lobe energy for a given peak side lobe amplitude.

Figure 2.32 shows the Gaussian windowed sinc function and its Fourier Transform.

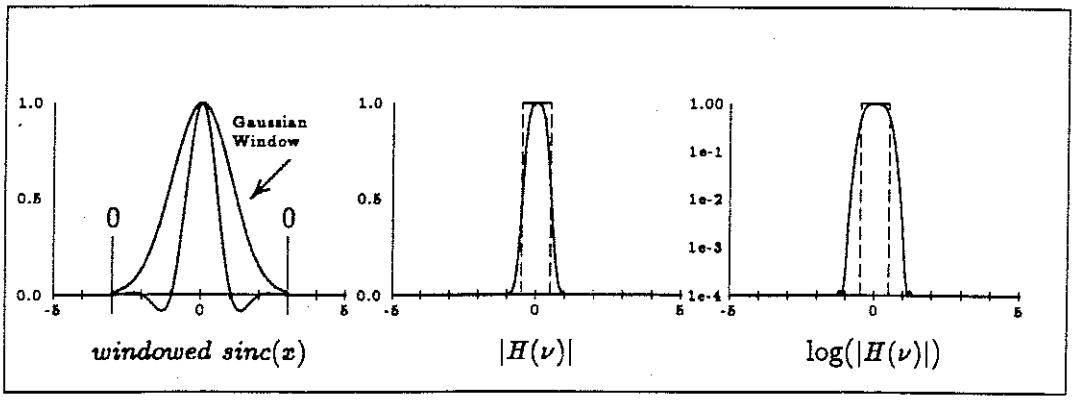


Figure 2.32: Spatial Domain Windowed Sinc Kernel in One Dimension.

Figure 2.33 illustrates Gaussian windowed sinc reconstruction of the four example functions described above.

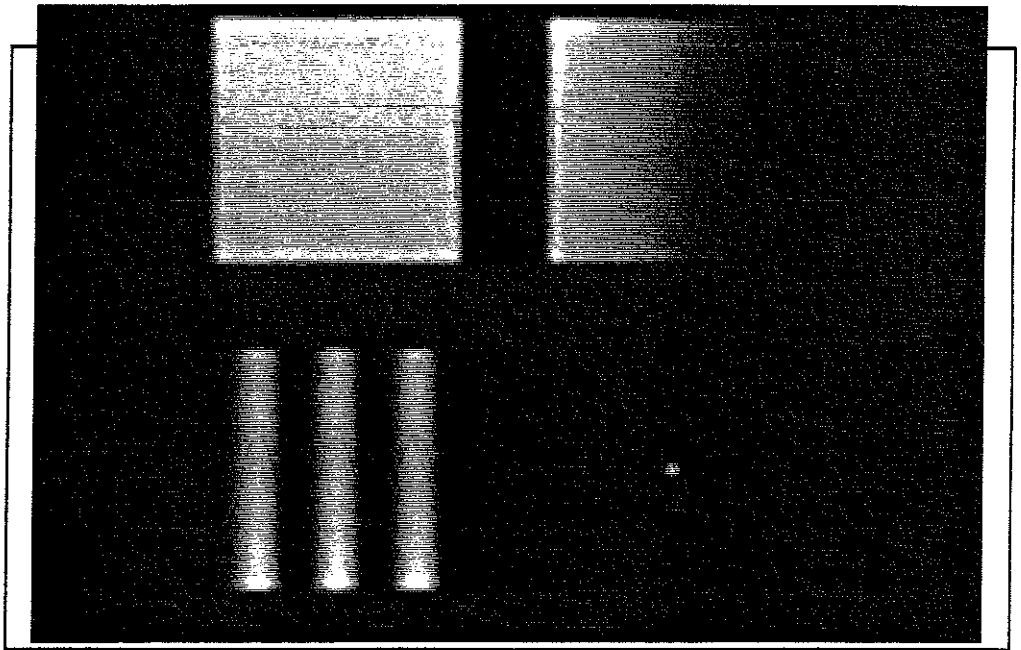


Figure 2.33: Spatial Domain Windowed Sinc Kernel Example.

Volume-rendering methods use a variety of interpolation functions, including nearest-neighbor interpolation, linear interpolation, cubic interpolation, and Gaussian interpolation. The computation-time vs. image-quality trade-off is the primary reason to chose different methods. Low-quality filters have limited spatial extent and are therefore quick to evaluate, while the higher quality filters have larger spatial extents and require proportionally more computation.

2.7.2 Nearest-Neighbor Interpolation

Computationally, the simplest method for interpolation is the nearest-neighbor function, illustrated in Figure 2.34. It is a zeroth-order interpolation function. The interpolator assigns each output sample the value of the nearest input sample. This is equivalent to convolving the input signal with a one-sample-wide rect function in the spatial domain. The rect function, also known as the sample-and-hold function or the Fourier window, is

$$h(x) = \begin{cases} 1, & \text{if } |x| < 0.5; \\ 0, & \text{if } |x| = 0.5; \\ 0, & \text{otherwise.} \end{cases}$$

Convolution with the rect function in the spatial domain is equivalent to multiplication by the sinc function in the frequency domain. The spatial rect function is a poor low-pass filter for two reasons: the central lobe distorts the passband and the other lobes allow stopband energy to get through to the result. These are exactly the two problems a good low-pass filter is trying to avoid. For these quality reasons, the nearest-neighbor method is usually used to obtain quick-preview images.

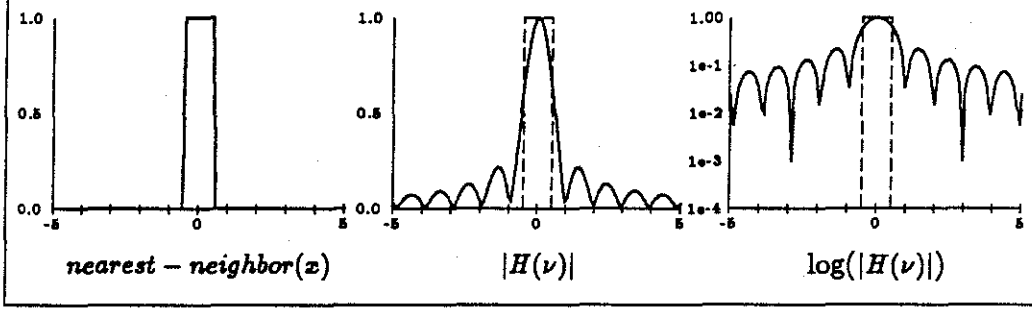


Figure 2.34: Nearest-Neighbor Interpolation in One Dimension.

Figure 2.35 illustrates nearest-neighbor reconstruction of the four example functions described above.

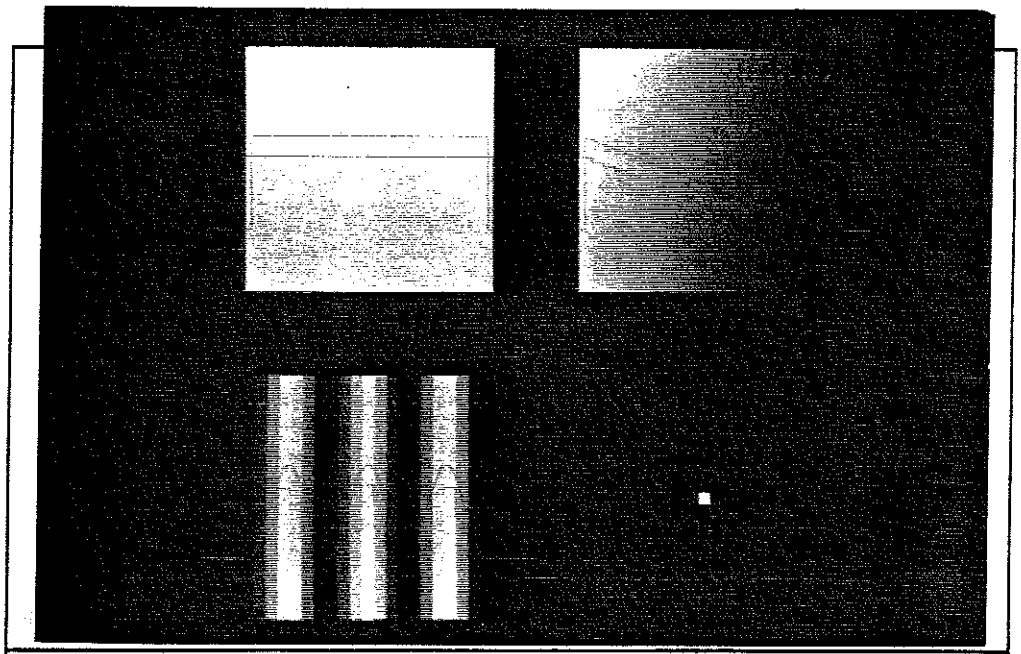


Figure 2.35: Nearest-Neighbor Interpolation Example.

2.7.3 Linear Interpolation

The first-order interpolation function passes a line between each consecutive pair of input samples and evaluates points on the line. This amounts to using a triangular pulse, as illustrated in Figure 2.36, as a reconstruction kernel. The triangle pulse's Fourier transform, $sinc^2(x)$, has an infinite extent and large positive lobes far from zero. These lobes pass energy from the stopband that causes C^1 discontinuities (false edges) in the output as can be seen in the lower left image of Figure 2.37. Volume-rendering algorithms use linear interpolation because the filter's extent in the spatial domain is only 2^N samples, where N is the dimension of the input (i.e. 2, 4, or 8 for the one-, two-, or three-dimensional cases). This high-frequency energy may manifest itself as false edges in the images giving the result an apparent sharpness that is not present in the data.

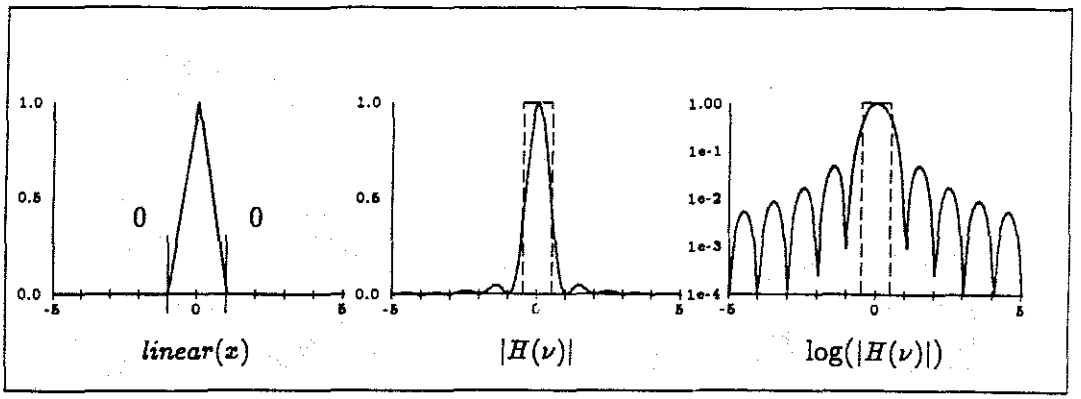


Figure 2.36: Linear Interpolation in One Dimension.

Figure 2.37 illustrates linear reconstruction of the four example functions described above.

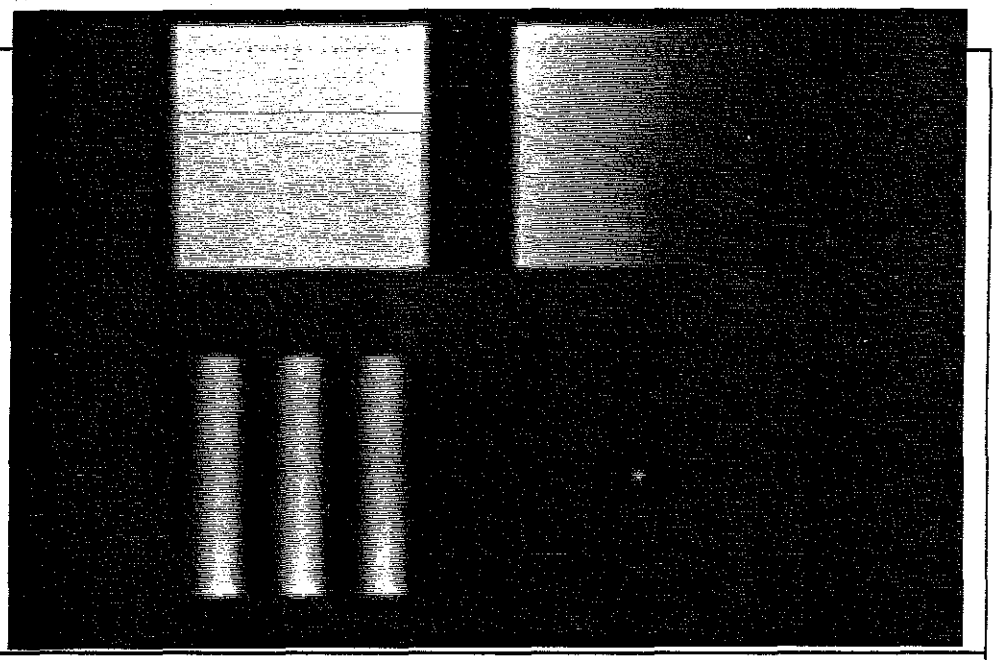


Figure 2.37: Linear Interpolation Example.

2.7.4 Quadratic and Even-Order Interpolation

Researchers seldom use an even-order interpolation function because the number of sample points on either side of the output point differs by one [Schafer 73], [Wolberg 90, page 128].

2.7.5 Cubic Interpolation

Mitchell defines a large family of cubic filters [Mitchell 88] that are both C^0 and C^1 continuous. Discontinuities in C^0 or C^1 cause high-frequency leakage that may lead to reconstruction artifacts, as seen in section 2.7.3. These filters also satisfy the constraint that for all x

$$\sum_{n=-\infty}^{\infty} h(x-n) = 1.$$

This prevents sample frequency ripple, as illustrated in Figure 2.38, which results when a comb function convolved with the kernel does not generate a constant.

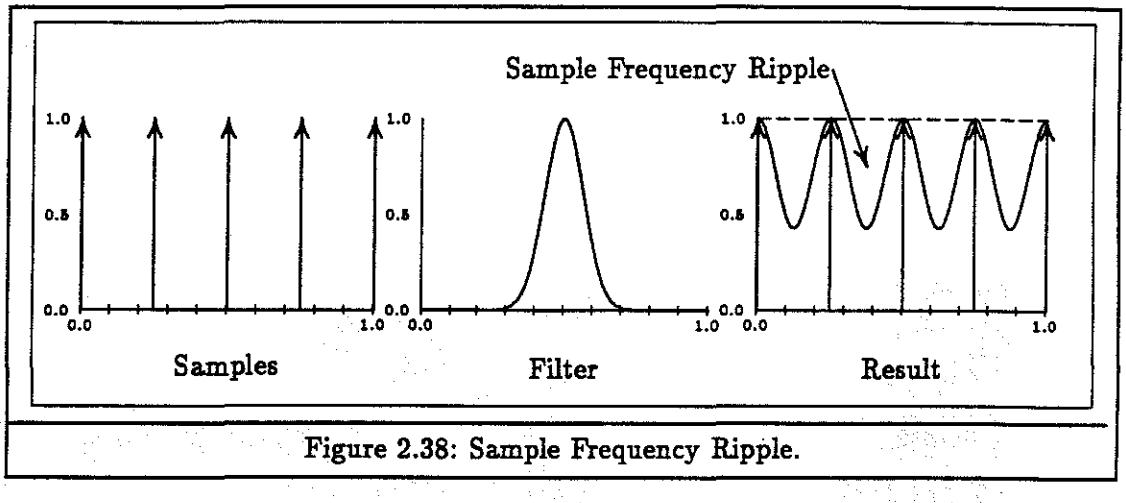


Figure 2.38: Sample Frequency Ripple.

The family of filters is defined by parameters B and C and is defined as

$$k(x) = \frac{1}{6}\begin{cases} (12-9B-6C)|x^3| + (-18+12B+6C)|x^2| + (6-2B), & \text{if } |x|<1; \\ (-B-6C)|x^3| + (6B+30C)|x^2| + (-12B-48C)|x| + (8B+24C), & \text{if } |x|<2; \\ 0, & \text{otherwise.} \end{cases}$$

General cubic interpolation can produce values that lie outside the range of the inputs. The interpolator must clamp these values. Tuning B and C can generate filters that trade the aliasing effects of blurring, ringing, and anisotropy [Mitchell 88]. A parameter setting of $B = 1.0$ and $C = 0.0$ generates the cubic B-splines, as shown in Figure 2.39. A parameter setting of $B = 0.0$ and $C = 0.5$ generates the Catmull-Rom splines, as shown in Figure 2.41. This family of filters has the further advantage that the filter kernel has a limited extent. This filter's extent in the spatial domain is only 4^N samples, where N is the dimension of the input (i.e. 4, 16, or 64 for the one-, two-, or three-dimensional cases).

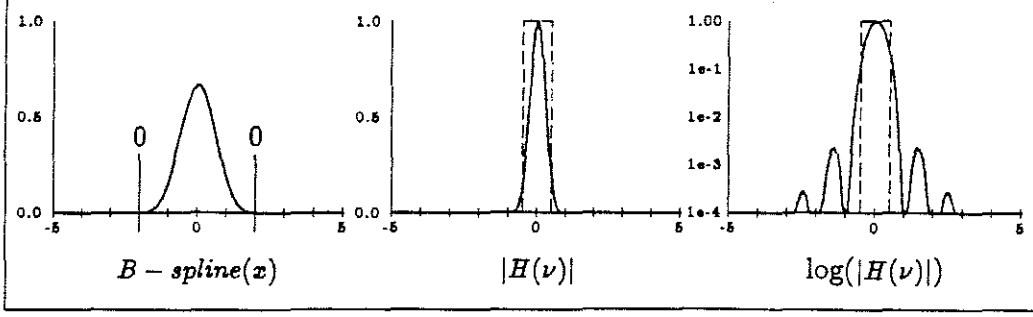


Figure 2.39: Cubic B-spline Interpolation in One Dimension.

Figure 2.40 illustrates the B-spline version of cubic reconstruction of the four example functions described above.

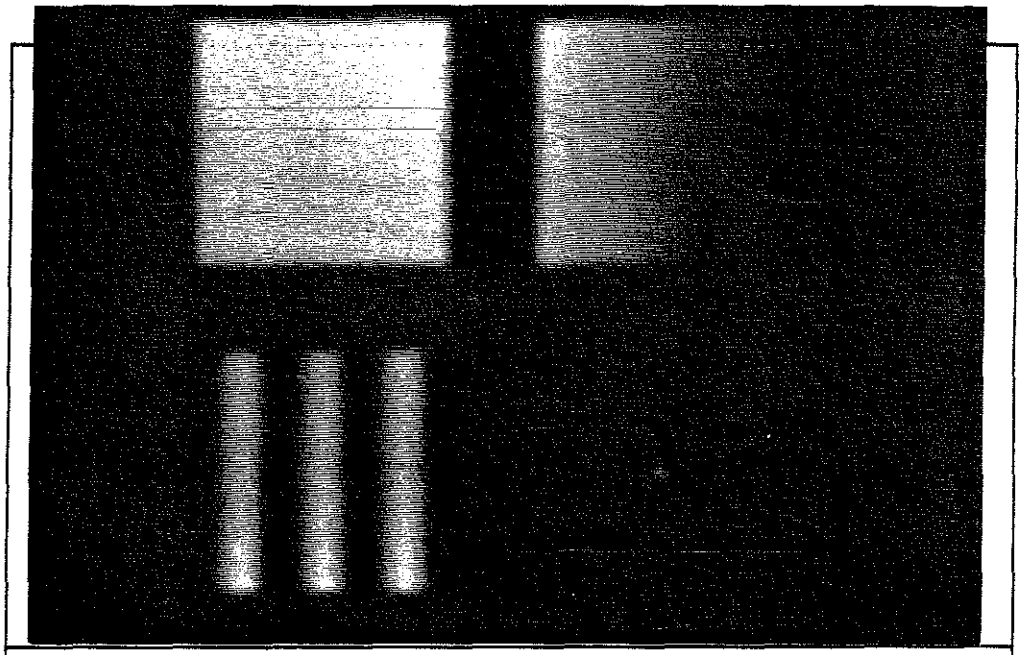


Figure 2.40: Cubic B-spline Interpolation Example.

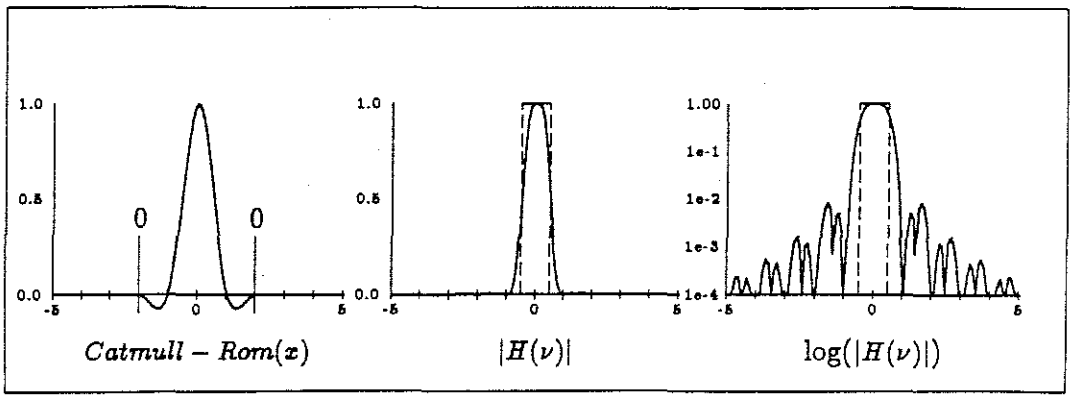


Figure 2.41: Cubic Catmull-Rom Interpolation in One Dimension.

Figure 2.42 illustrates the Catmull-Rom version of cubic reconstruction of the four example functions described above.

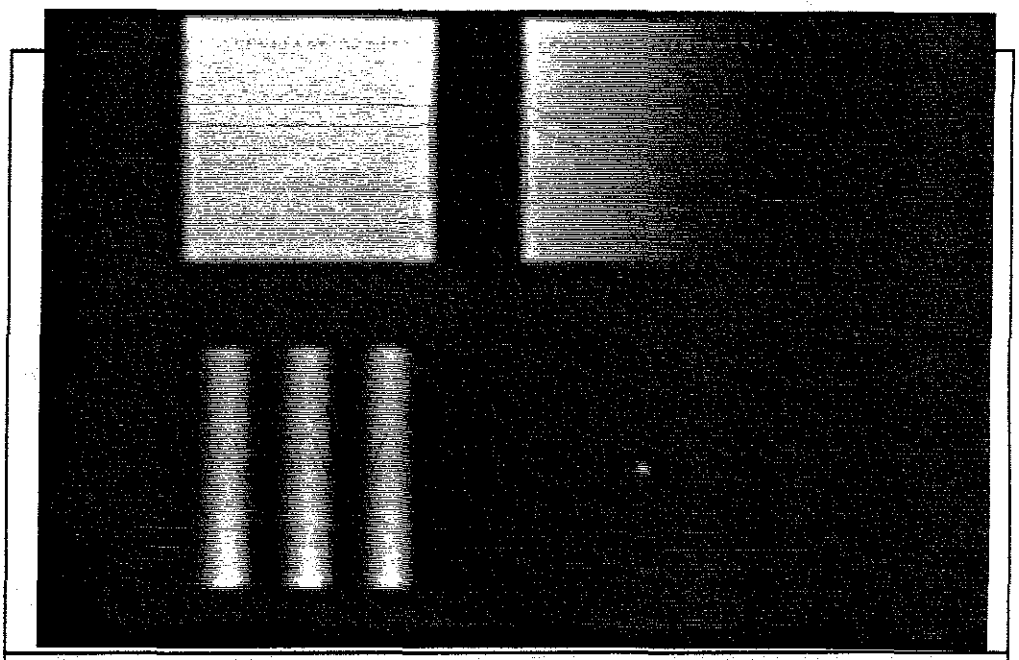


Figure 2.42: Cubic Catmull-Rom Interpolation Example.

2.7.6 Gaussian Interpolation

Some researches use a Gaussian as an interpolation function. A Gaussian's width is controlled by the parameter σ. A Gaussian, illustrated in Figure 2.43, is defined as

$$f_{Gaussian}(x) = e^{-\frac{x^2}{2\sigma^2}}.$$

In both the spatial and frequency domain, the kernel has infinite extent. However, the Gaussian asymptotically falls toward zero quickly and does not collect much energy from the stopband. Since it falls toward zero quickly, the Gaussian may be truncated in the near zero area without causing much ringing. Another feature of the Gaussian is that it is the only filter that is both separable and rotationally symmetric, (section 2.8). A problem with the Gaussian is that the filter distorts the passband by attenuating the high-frequency components of the main replica and causes blurring in the output.

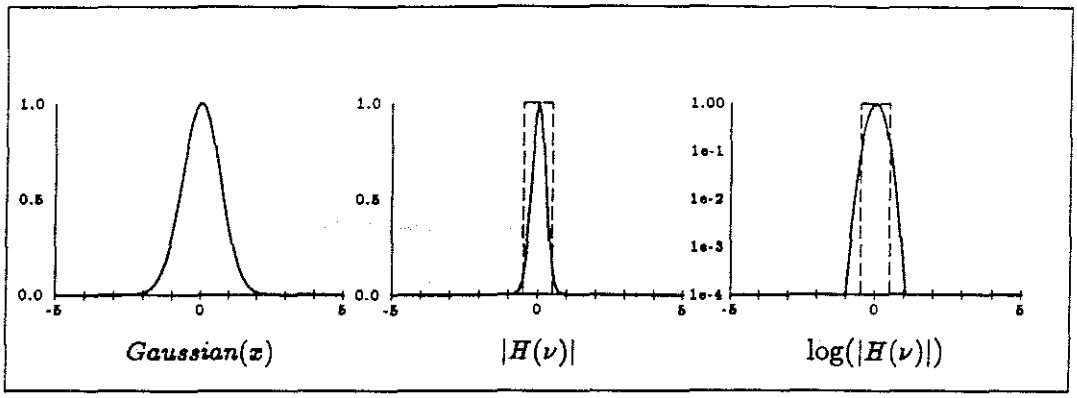


Figure 2.43: Gaussian Interpolation in One Dimension.

Figure 2.44 illustrates Gaussian reconstruction of the four example functions described above.

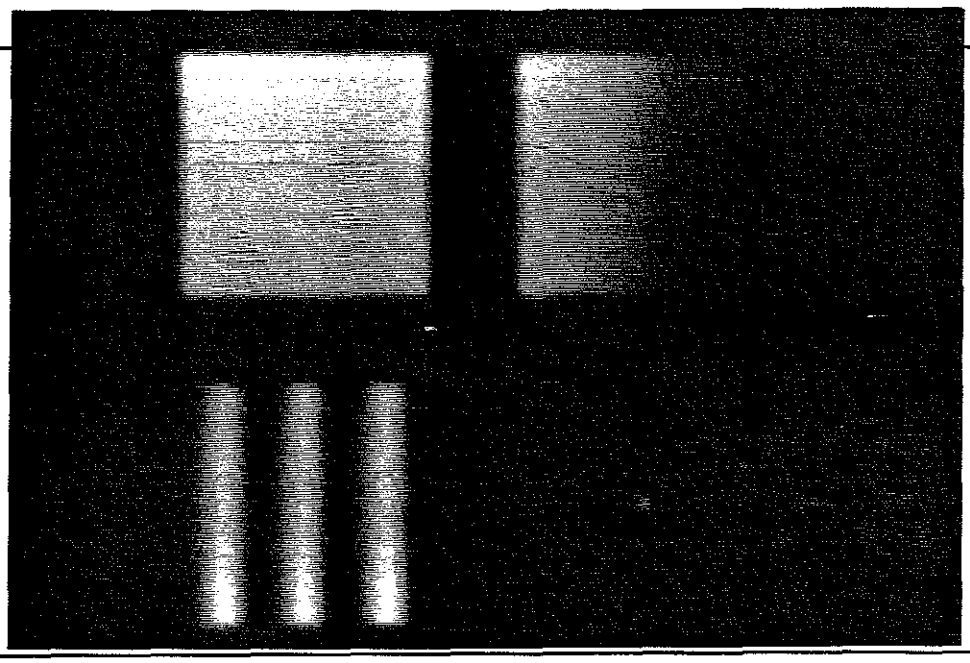


Figure 2.44: Gaussian Interpolation Example.

2.8 Multi-Dimensional Interpolation

There are two ways to extend the above kernels to higher dimensions.

One method is to use each coordinate as a free variable, separate the one-dimensional kernels, and multiply their results. A three-dimensional linearly-separable kernel is

$$h(x,y,z) = h'(x) \times h'(y) \times h'(z). \qquad Method(1)$$

The second method is to use the radius from the kernel center as the free variable. This produces rotationally-symmetric kernels. These kernels suffer from sample frequency ripple, as illustrated in Figure 2.38, where equal amplitude samples do not produce a flat field [Castleman 79, page 42]. Instead they produce a field that differs from the constant amplitude at points between the equal valued input samples. A three-dimensional rotationally-symmetric kernel is

$$h(x,y,z) = h'(\sqrt{x^2+y^2+z^2}). \qquad Method(2)$$

The correct way to extend reconstruction filters to higher dimensions is by method 1, because rotationally-symmetric kernels, except for the Gaussian which is also separable, will not correctly reconstruct a regular mesh.

2.9 Interpolation In Current Methods

Volume renderers seldom generate images that are axis-aligned and have identical sampling rates for the input data and the image. Only in this case can the renderer use the actual data samples to generate the image. In all other instances the renderer must generate samples from the data at locations between the input values by calculating a continuous function over the input and resampling this continuous function. The generation of the continuous function is reconstruction. This is typically done by interpolating the input samples with an interpolation function and using the interpolated values to generate the image samples.

The four main volume-rendering approaches perform the interpolation at different points in the rendering pipeline and with different interpolation functions.

Data-coercion methods generate the continuous function at the first step by approximating the function with lines or surfaces. In the "marching-cubes" method [Lorensen 87], reconstruction is the linear interpolation in the three dimensions required to generate the surface triangles. This choice of coercion primitive is made without regard for the signal processing aspects of volume rendering. Their goal, as stated in [Lorensen 87], is to generate hard-surface images from computed tomography data. Since linear interpolation generates C^1 discontinuities, as shown in section 2.7.3, and binary classification introduces high frequencies in the output, as shown in section 2.4, images generated by the "marching-cubes" method have many artifacts, such as sharp changes in the image values. In addition to reconstruction artifacts, these images also exhibit artifacts of the surface rendering process. These artifacts may vary widely depending on the quality of the surface renderer.

Ray-casting methods [Van Hook 86], [Levoy 88] would be equivalent to the "marching-cubes" method, if they strictly used binary classifiers and linearly searched each volume element for the desired isovalue surface value. However, these methods typically interpolate the input volume along rays at a predetermined sampling rate and may miss the isovalue surface. To solve this problem, they use non-binary classifiers that change the range of the acceptable density values for a given isovalue surface based on how fast the data is changing near the sample point [Levoy 88]. When the data values are changing slowly, a narrow range of values is acceptable as the isovalue surface. When the data values are changing rapidly, a wide range of values is acceptable as the isovalue surface and the isovalue surface is not missed.

Some ray-casting methods allow the user to change from nearest-neighbor interpolation for quick and aliased preview, to linear interpolation for day-to-day image generation, to cubic interpolation for publication-quality imagery. Different interpolation functions require different amounts of computation because they have different spatial extents. In the three-dimensional case, the nearest-neighbor method only needs to know the input value for the nearest sample, whereas the linear method needs to know the 8 nearest samples, and the cubic method needs to know the 64 nearest samples. As the complexity of the interpolation methods increases, the quality of the image increases. Nearest-neighbor interpolated images suffer from blockiness caused by the sharp transitions when neighboring pixels are closest to different input samples, all seen in Figure 2.35. Linearly interpolated images suffer from C^1 discontinuities caused by the transitions when neighboring pixels are generated from different sets of input samples, as seen in Figure 2.37. This is most noticeable as the common star pattern seen in bilinearly enlarged images and would show up as a three-dimensional star pattern in volume-rendered images. Since cubic interpolation has twice the extent of linear interpolation and the effect of any given sample smoothly goes to zero as the sample no longer affects the interpolation result, there are fewer sharp changes in the resulting images. The trade-off is the interpolator tends to blur the result. Viewers are often dissatisfied with the blurring because they are accustomed to seeing the sharp edges that linear interpolation generates.

Some ray-casting methods shade the original samples and then attempt to interpolate these values. For the interpolation process to be free of reconstruction errors, the shaded signal should be filtered to lower the signal's Nyquist rate so that it is below the resampling rate.

Affine transformation methods must use separable kernels so the methods can use the two-pass and three-pass image warping methods. These methods are much less sensitive to kernel extent sizes, because for a kernel extent of n, the amount of computation is $O(n)$ instead of $O(n^3)$ for the ray-casting methods. Therefore, the three-pass approaches typically use high-quality reconstruction kernels for all image generation. Hanrahan [Hanrahan 90] is careful to consider signal processing concerns in his method. He chooses the order of the three passes that performs all magnification before minification. If a minification step precedes a magnification step, much of the information in the

signal would be lost, since multiple samples are collapsed into a single sample during minification and they cannot be separated during a subsequent pass. This ordering allows the renderer to retain the maximum amount of the signal's information during the transformation.

Element-tossing approaches vary widely on how they do interpolation because they toss different elements. Polyhedra-tossing approaches [Max 90] use linear interpolation to scan-convert the front and back faces of each polyhedron. This exhibits the C^1 discontinuities common to linear interpolation in the images [Max 90]. The method usually runs on unstructured meshes, and in general the reconstruction process for non-uniform sampling is not well understood and is significantly more complex than the uniform sampling case. Because the goal of the method is to generate images from unstructured meshes and the problem of reconstructing non-uniformly sampled samples is not well understood, linear interpolation is the best interpolation method we currently know how to use on unstructured meshes.

Point-tossing approaches, such as splatting [Westover 90], use Gaussian reconstruction, since the Gaussian is both separable and rotational-symmetric. This allows the method to integrate one dimension out of the kernel and perform two-dimensional reconstruction instead of three-dimensional reconstruction. While this is not as good as the computational savings of the one-dimensional reconstruction used in the affine transformation methods, it is better than the computational requirements of the three-dimensional reconstruction used by either data-coercion methods or ray-casting methods. Gaussian reconstruction tends to blur the resulting image and viewers are often dissatisfied with the absence of image sharpness, even though the image sharpness of other methods is a reconstruction artifact.

Chapter 3

Splatting Method

3.1 Introduction

This chapter presents the feed-forward splatting rendering algorithm. Feed-forward algorithms are those that directly map data onto the image plane. The algorithm presented here was inspired by the structure of the traditional graphics pipeline in which a single element passes through a single pipeline stage one at a time. Examples in surface graphics include the z-buffer algorithm and the "painter's" algorithm. This renderer maps each data element onto the image plane and then adds the element's contribution to the accumulating image. The renderer terminates when it has added each data element to the image. The splatting method uses feed-forward convolution to distribute energy from input samples onto the image plane. The renderer transforms and shades each input sample and passes samples that have nonzero opacity to the reconstruction process. The reconstruction process calculates an image-plane footprint for each data sample and uses the footprint to spread the sample's energy onto the image plane. The renderer calculates visibility by compositing each sample's footprint into an incrementally updated accumulation buffer. This works because the input data may be traversed in a presorted order, either a back-to-front or front-to-back, since it is a rectilinear mesh. When the renderer has processed all the input samples, the accumulation buffer is the final image.

Many volume renderers run in a non-interactive mode, because an image may take many hours to compute. When the renderer completes the image, the user may change the input parameters and try again, iterating until satisfied. While this batch method of volume rendering may be satisfactory for presentation-quality image generation, it does not lend itself to data exploration. The length of time between iterations makes experimentation time-consuming and destroys idea continuity.

The goal of the splatting algorithm is a system for interactive exploration of volume data with enough flexibility to encourage the user to try numerous and unconventional mappings. The splatting algorithm lends itself naturally to parallelism because it treats each sample independently and multiple processes can operate concurrently on multiple samples. In addition, as the samples are treated independently, they can be operated on by a series of functional blocks in a pipeline. Each functional block performs a single function on each sample, and these single-function blocks may be written as table-driven processes. The renderer uses only table-driven CRIO processes, thus the user can

understand the data mappings. The renderer also allows the user to control every step in the generation process. When users change an input viewing parameter, they should see the change instantly. The altered image may not be the final image, but should be adequate for the user to quickly steer through the data [Brooks 86], [Greenberg 86].

The reconstruction step is the most complicated portion of the algorithm. The renderer must determine each sample's image-space contribution to the final image. A brute-force method would one-dimensionally integrate the reconstruction kernel for every pixel for every input sample. If the renderer can calculate the image space extent of the kernel, the number of integrations reduces to the number of samples times the number of pixels that fall within the extent, however, this is still many integrations.

In an orthographic view, the footprint of the projected reconstruction kernel is the same for all samples except for an image-space offset. This allows the renderer to build a footprint function table once per image and use the table for all samples. Since the table is discrete, the renderer builds it at sub-pixel resolution to help reduce aliasing artifacts. The renderer could perform the actual integrations of the kernel for each sample in the footprint table. However, this is still many integrations and integration is usually a compute-intensive operation.

To reduce the computational complexity of generating the single footprint table, the renderer uses an approximating function to build the footprint table for the particular view. This approximating function is intended to model the result of integrating the reconstruction kernel. The renderer evaluates the approximating function for each sample in the footprint table, instead of calculating any integrations. The renderer must make two calculations to use the approximating function. First, the renderer computes the image-space extent of the projection of the reconstruction kernel. Second, the renderer computes a mapping from this extent to the extent that surrounds the approximating function. Then for each entry of the footprint table, the renderer maps the entry from the transformed region extent to the approximating function extent and evaluates the approximating function. The renderer builds the footprint table once per image and uses the table for all input samples. The renderer centers the table at the sample's projected image-space location and accesses the table at the center of each pixel that falls within the table's extent.

3.2 Design Trade-offs

A primary goal of this thesis is to identify an algorithm suitable for interactive volume rendering that generates high-quality images. This desire led to many specific decisions during algorithm development.

A feed-forward method was chosen because it can treat each volume element independently, making the algorithm easy to parallelize. Since each element is independent of the other elements, there can be a set of processes each doing the same algorithm, working on subsets of the data. Each of these processes is identical to the process that runs on the single processor version. The parallel aspects of the feed-forward method are discussed in Chapter 4.

The renderer runs in either a back-to-front order or a front-to-back order. The back-to-front order allows the user to see features in the data set that are subsequently hidden by other features. The front-to-back order allows the user to understand the final image sooner, so he can change the viewing parameters if he is dissatisfied with the current image.

As much of the rendering process as possible is table-driven because table-driven computations are much less computationally intensive than explicit computation. This mostly affects the CRIO process and the reconstruction process. The reconstruction process uses a footprint table for speed, which is described in section 3.7. The CRIO process uses four shading tables to eliminate much of the CRIO computation. Section 3.6 describes the CRIO process in detail.

The renderer shades the input samples. While this is contrary to the ideal volume rendering process, as described in Chapter 2, it is an important optimization. The most important factor affecting rendering time in a feed-forward renderer is pipeline throughput. Reducing the number of input samples traveling down the rendering pipeline reduces the rendering time. Shading first allows the renderer to send only nonzero-opacity tuples to the reconstruction process. The reconstruction process is the most compute-intensive portion of splatting, and reducing the amount of work that the reconstruction process must do directly increases rendering speed. Many volume-rendered images contain only a small percentage of nonzero-opacity tuples. Even if the entire volume generates nonzero-opacity tuples, shading first does not increase the amount of data for the pipeline. Therefore, shading first cannot adversely affect rendering times, it can only help.

The renderer models each input sample as a reflective, light-emitting, semi-transparent cloud. This retains the transparent nature of clouds and allows for shading effects which give users many three-dimensional cues about the data [Levoy 88].

For maximum quality, the renderer should not make any binary classification or shading decisions [Levoy 88], [Drebin 88]. The shading tables, described in section 3.6, do not force the user to define tables that are soft classifiers, but certainly allow the user to specify soft classifiers without affecting the rendering speed of the system. If the user specifies a binary classifier, the results may have many rendering artifacts, because a binary classifier may significantly change a sample's spectrum.

The algorithm only renders orthographic views. Perspective is difficult for both feed-forward and feed-backward algorithms. For a perspective view, the apparent sampling rate of the input data changes with depth. Feed-forward algorithms must adjust a sample's region of effect for each apparent sampling rate. Feed-backward algorithms must cast rays at sufficient sampling rates to guarantee that the portion of the data that lies furthest from the image plane is adequately sampled. However, orthographic views of volume data are sufficient for many applications, and often viewers prefer orthographic images rather than perspective images, because they maintain the shape of and the distance between features.

3.3 Rendering Algorithm

The splatting algorithm discussed in this chapter is a feed-forward algorithm that shades at input samples and reconstructs a final image from the shaded volume. The algorithm consists of four main parts: transforming, CRIO, reconstruction, and visibility, as illustrated in Figure 3.01. Information move between the processes in the form of a tuple for each sample. These tuples (A, B, C, D) contain different intermediate results as described below.

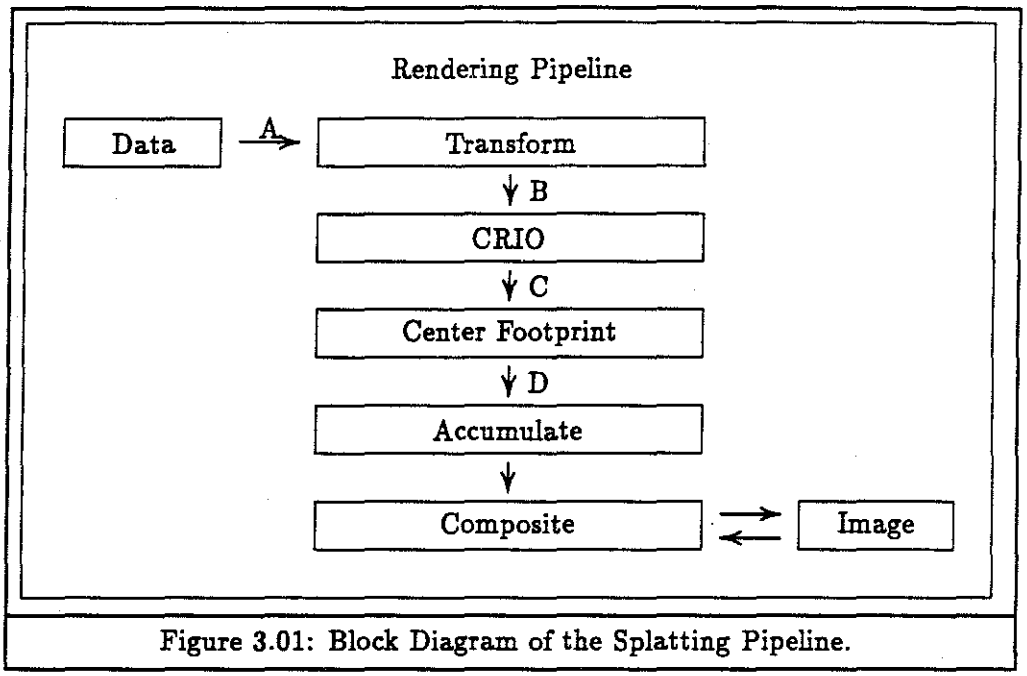


Figure 3.01: Block Diagram of the Splatting Pipeline.

3.4 Pipeline Structure

The input to the renderer is a input tuple, A:

density value	$< d >$	$densityunits/linearunits^3$
gradient strength	$< m >$	$densityunits/linearunits^3/linearunit$
gradient direction	$< \theta, \phi >$	degrees
mesh coordinates	$< i, j, k >$	integers specifying mesh location

The renderer view-transforms the input tuple, converting the mesh coordinates into image coordinates and generates an image-space tuple, B:

density value	$< d >$	$densityunits/linearunits^3$
gradient strength	$< m >$	$densityunits/linearunits^3/linearunit$
gradient direction	$< \theta, \phi >$	degrees
image coordinates	$< x, y, z >$	pixels for x and y, depth for z

The renderer runs the CRIO process on the image-space tuple, where color and opacity are arbitrary functions of the elements in the image-space tuple. This step generates a CRIO tuple, C:

color	$< r, g, b >$	0.0 to 1.0

opacity	$< a >$	0.0 to 1.0
image coordinates	$< x, y, z >$	pixels for x and y, depth for z

If the opacity of the CRIO tuple is not zero, the renderer passes the CRIO tuple on to the reconstruction process. The reconstruction process determines the image-space effect of the CRIO tuple, called its *footprint*. This generates a single splat tuple, D:

color	$< r, g, b >$	0.0 to 1.0
opacity	$< a >$	0.0 to 1.0
footprint coordinates	$< x, y >$	pixels

The renderer then combines the splat tuple into the image using a visibility rule.

The following sections describe these four processing steps in detail.

3.5 View Transformation

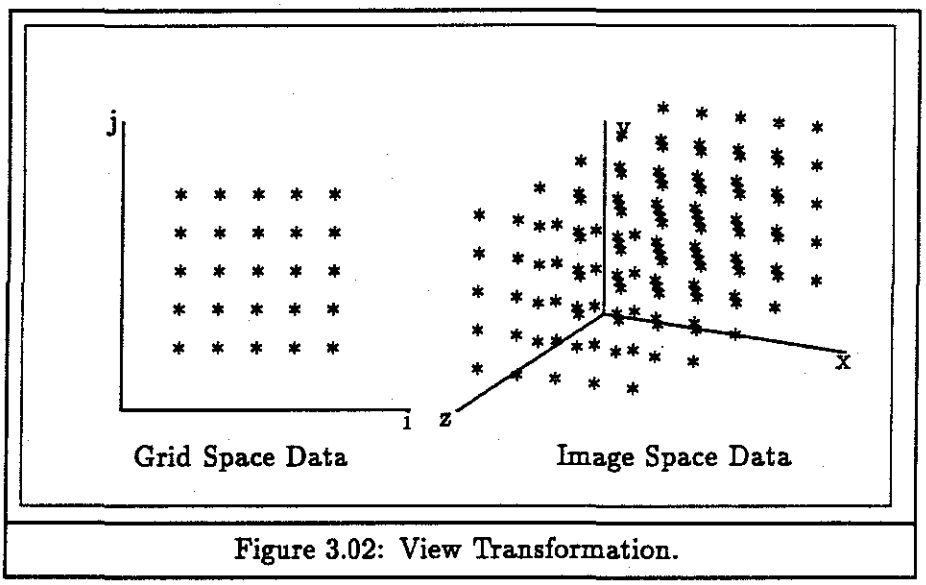


Figure 3.02: View Transformation.

The view-transformation process's job is to convert the input tuple's mesh space $< i, j, k >$ into an image space $< x, y, z >$, illustrated in Figure 3.02. Orthographic views, defined by the view-transformation matrix, are simple to generate, because the input volume is a rectilinear mesh.

3.5.1 Method

The homogeneous transformation matrix for an orthographic view is

$$T = \begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ j & k & l & 1 \end{pmatrix}.$$

The view-transformation matrix is a 4 x 4 matrix, because the view may have translations and the footprint generation process uses homogeneous transformations to build the footprint. Entries a through i represent the scaling and the rotation portion of the transformation and entries j, k, and l represent the x, y, and z translations.

The renderer maps an input sample at mesh point $< i, j, k >$ to image-space point $< x, y, z >$ by

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ k \\ 1 \end{pmatrix}.$$

We see that the incremental step in image-space coordinate for each step along a mesh direction is

$$\begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix} = \begin{pmatrix} \frac{\Delta x}{\Delta i} & \frac{\Delta x}{\Delta j} & \frac{\Delta x}{\Delta k} \\ \frac{\Delta y}{\Delta i} & \frac{\Delta y}{\Delta j} & \frac{\Delta y}{\Delta k} \\ \frac{\Delta z}{\Delta i} & \frac{\Delta z}{\Delta j} & \frac{\Delta z}{\Delta k} \end{pmatrix} \times \begin{pmatrix} \Delta i \\ \Delta j \\ \Delta k \end{pmatrix}.$$

or,

$$P_{x+\Delta x} = P_x + \Delta M \times \Delta x.$$

Thus, the renderer can read the step sizes directly from the transformation matrix, for each of x, y, and z for each input i, j, and k. These step sizes are constant throughout the entire volume, since we begin with uniformly sampled input data and we are generating only orthographic projections. By inspecting the sign and magnitude of the change in z values, the renderer can determine a traversal that guarantees a back-to-front ordering or a front-to-back ordering. The ordering traverses either the i, j, or k mesh direction fastest, one of the remaining two directions second fastest, and the last remaining direction slowest. The fastest changing axis is the axis that has the smallest absolute change in z for each step along that mesh direction. The sign of the delta z value for that mesh direction determines which end of the cube the renderer starts traversing for that axis. Similarly the second fastest changing axis is the axis with the second smallest change in z for each step in along that mesh direction. The two fastest changing axes define a plane of data perpendicular to the other axis, called a *sheet*. There are three possible sheet groupings for a data set. The first contains all the samples in the i and j direction for each k, the second contains all the samples in the i and k direction for each j, and the third contains all the samples in the j and k direction for each i. A sheet of data is a slice of the data set perpendicular to one of the mesh axis that is closer to being parallel to the image plane after the view transformation than the other two possible groupings.

Both the back-to-front traversal order and front-to-back traversal order have merits. The back-to-front ordering allows the renderer to display the partial images during rendering and show hidden features. The front-to-back ordering allows the users to stop image generation sooner if he does not like the image, because the current image displayed is always part of the final image. The current renderer will generate images in either order, depending on the value of a run-time flag.

Once the renderer knows these deltas and the ordering, the renderer transforms the furthest or nearest input point with a full matrix multiplication. This is the origin for the incremental updates to the image-space coordinates one sample at a time. The renderer adds the appropriate deltas as it processes the input volume, generating image-space tuples. The renderer sends these tuples to the CRIO process.

3.6 CRIO Process

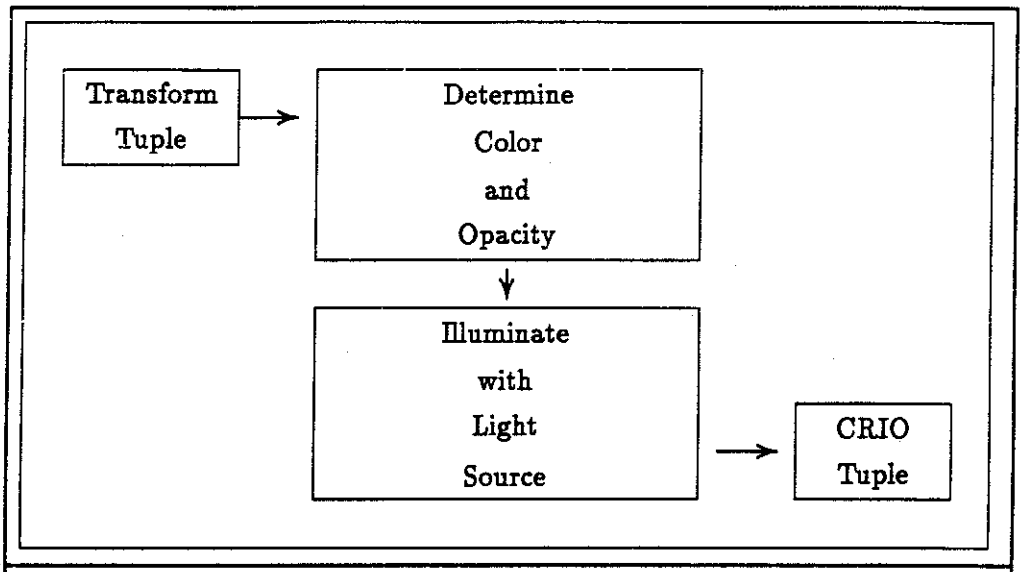


Figure 3.03: Block Diagram of the CRIO Process.

The CRIO process's job is to convert the image-space tuple into a CRIO tuple, illustrated in Figure 3.03. If the renderer's CRIO process uses only information contained as either part of the sample tuple or information generated once in a preprocessing step, any CRIO process can be used with the splatting algorithm. The CRIO process uses non-binary classification and the application of a user-specified shading model. Both of these steps are table-driven for speed. A common table-driven CRIO process has four parts: emittance, diffuse reflection, specular reflection, and opacity calculations [Westover 89]. The CRIO process requires the density, gradient strength and gradient direction for each input sample. Density and gradient strength are often used to select a density value's color and opacity properties. The gradient direction is used as a pseudo surface normal during the illumination calculation.

3.6.1 Method

As the gradient operator requires knowledge of neighboring samples, either a preprocessor generates the gradient information or the renderer duplicates a small shell around each volume so the gradient is defined on the outer faces of the volume. While similar applications use a variety of gradient operators [Horn 81], [Van Hook 86], [Drebin 88], this renderer uses the following:

$$gradient_i(i,j,k) = data(i+1,j,k) - data(i-1,j,k)$$

$$gradient_j(i,j,k) = data(i,j+1,k) - data(i,j-1,k)$$

$$gradient_k(i,j,k) = data(i,j,k+1) - data(i,j,k-1).$$

The magnitude of the gradient is its length normalized to the maximum gradient possible for the input data set. For a data set with a maximum value of Max_{sample} and a minimum value of Min_{sample}, the maximum gradient magnitude is

$$MaxMagnitude_{gradient} = \sqrt{(3(Max_{sample} - Min_{sample})^2)}.$$

Once the renderer calculates the magnitude of the gradient, it normalizes the gradient vector and stores two components of the vector in the input tuple. The gradient vector is used as the surface normal for the illumination calculation.

The CRIO process uses four tables to determine a image-space tuple's color and opacity. A table is used to determine emitted color, a second table is used to determine opacity, a third table is used to determine reflected color, and a fourth table is used to vary the sample's opacity based on a second value in the tuple. The renderer indexes these tables by a value available in the image-space tuple. These values include the density value, gradient strength, gradient direction, and the image-space position. The renderer uses the first two tables to determine the color and the opacity of the value specified by the input. It uses the third table for the shading model and the fourth table for feature enhancement. In addition, the illumination calculation uses a set of user-specified light sources. These are infinite light sources specified by a color and a light-source direction. Rather than transforming the gradient vector for each sample, the renderer transforms each light source direction by the inverse of the view-transformation matrix. Thus, the renderer is required to only transform the light source direction once, instead of transforming each gradient vector. Various components of the CRIO process may be turned off for two reasons. First, portions of the illumination calculation are more time consuming than others and the user may turn these parts off to decrease rendering times. Second, the some illumination cues occasionally impede understanding the data and the user turns these parts off to simplify the image.

Below are the equations for calculating the result color and opacity of a image-space tuple [Westover 89]. The final color intensity is composed of three parts: the intensity due to the emitted light from the sample, the intensity due to the diffuse portion of the illumination model, and the intensity due to the specular portion of the illumination model.

Let I denote the intensity, A denote the opacity, L denote the light-source direction vector, G denote the gradient direction vector, H denote the vector half-way between the eye vector and the light vector, and n denote the specular exponent. Since the value of a component of the intensity vector may be less than 0 or more than 1, the renderer clamps each intensity component to fall between 0 and 1.

The emittance rule for shading is

$$I_{emit} = Table_{emit}[index_{emit}].$$

The diffuse rule for shading is

$$I_{diff} = Table_{reflection}[index_{reflection}] \times (L \bullet G).$$

The specular rule for shading is

$$I_{spec} = Table_{reflection}[index_{reflection}] \times (H \bullet G)^n.$$

The final color intensity is

$$I_{result} = I_{emit} + I_{diff} + I_{spec}.$$

The opacity rule for shading is

$$A_{result} = Table_{opacity}[index_{opacity}] \times Table_{modulate}[index_{modulate}].$$

3.6.2 Application of Shading Rules Examples

Surface enhancement is an example of a use for the opacity-variation table. If the table contains a ramp from 0 to 1 and the gradient strength selects a value, samples that have a low gradient magnitude will have a low opacity-variation value and samples that have a high gradient magnitude will have a high opacity-variation value. The result of multiplying the sample's opacity by the opacity-variation value is to make samples that are in areas where the density is changing slowly more transparent and make samples that are in areas where the density is changing rapidly more opaque. The underlying assumption is that areas where the density is changing rapidly are probably surfaces. This is shown in Figure 3.04, where skin densities have a red color, bone densities have a white color, and the opacity variation table contains a ramp that selects areas of high gradient magnitude. Notice the skin surface on the right-hand image and the double bone surface in the cut of the jaw bone. The bone appears to be hollow, because the inside of the bone has slowly varying density and the opacity-variation table makes it transparent.

The computed tomography data set is 256 x 256 x 93 samples. The 93 axial slices are 1.5mm thick and are sampled at 0.8mm in the x and y directions. The data is courtesy of William Lorensen, General Electric Company, and was generated by D. C. Hemmy, MD, of the Medical College of Wisconsin.

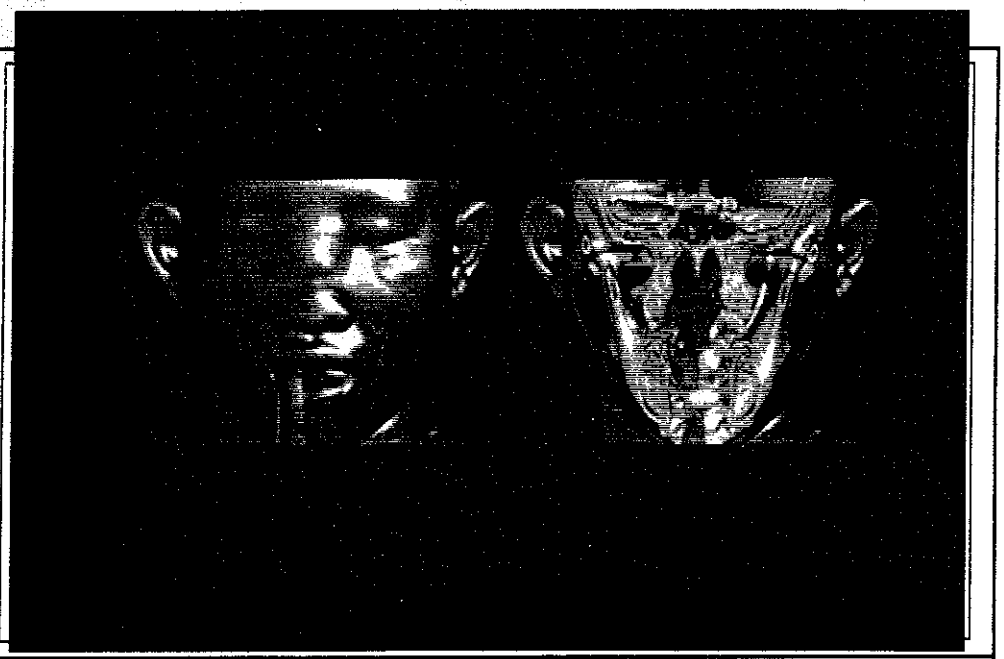


Figure 3.04: Using the Opacity-Variation Table for Surfaces.

Another use of the opacity-variation table is to index the opacity-variation table with the packet's z value. If the opacity-variation table contains a ramp from 0 to 1 and z values near the image-plane select values near 1 and z values far from the image-plane select values near 0, the result is a pseudo depth-cueing image, the left side image in Figure 3.05. The right side image in Figure 3.05 is the molecule shaded without depth cueing.

The molecular data set is 137 x 113 x 59 samples. The three mesh directions are sampled at the same rate. The molecule is ribonuclease. The data is courtesy of Christopher Hill, Chemistry Department, University of York. The original data was shaded by Marc Levoy, Computer Systems Laboratory, Stanford University, using his isovalue-surface shading model [Levoy 88]. The splatting renderer used an identity mapping so it did no shading and only reconstructed the volume and projected the image.

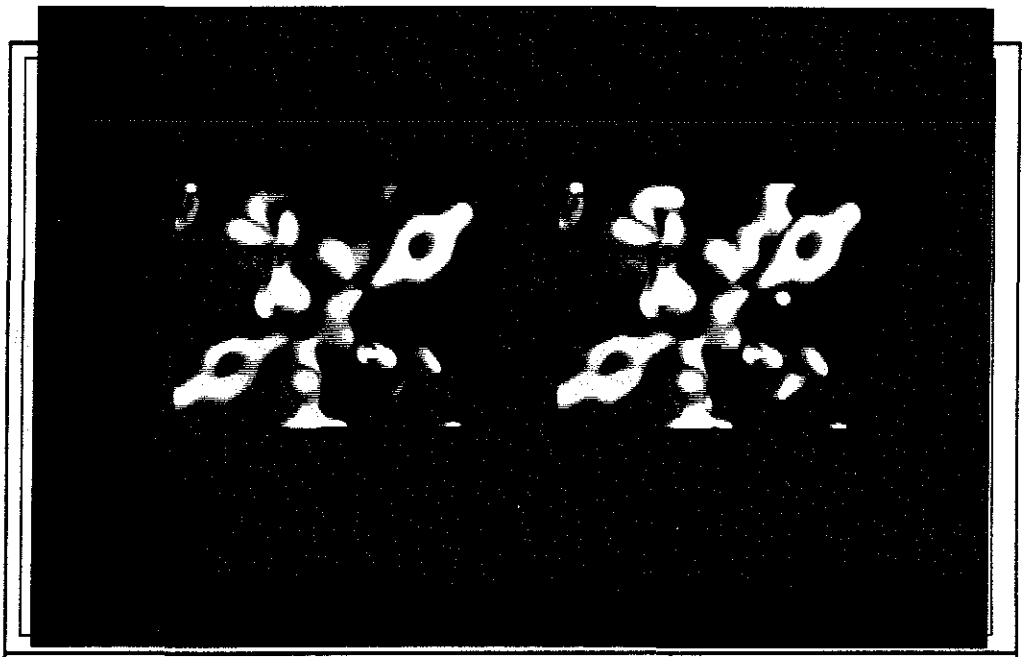


Figure 3.05: Using the Opacity-Variation Table for Depth Cueing.

Biochemists achieve other interesting effects by using the gradient magnitude to select the emitted color for a sample. This colors the density by gradient strength which may help select atom types in an electron density map. Two views of the same gradient-shaded molecule are shown in Figure 3.06. The right view is the left view rotated about the x-axis by 90°.

This molecular data set it 55 x 53 x 57 samples. The three mesh directions are sampled at the same rate. The molecule is a transfer RNA. The data is courtesy of Frank Hage, Biochemistry Department, University of North Carolina at Chapel Hill.

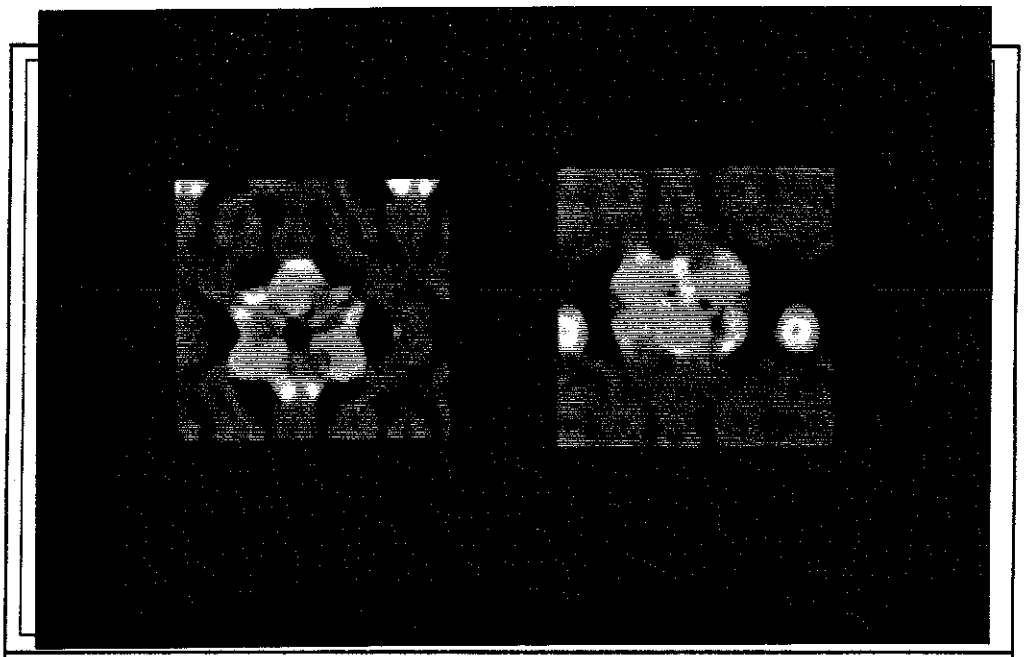


Figure 3.06: Gradient Shading.

3.7 Reconstruction

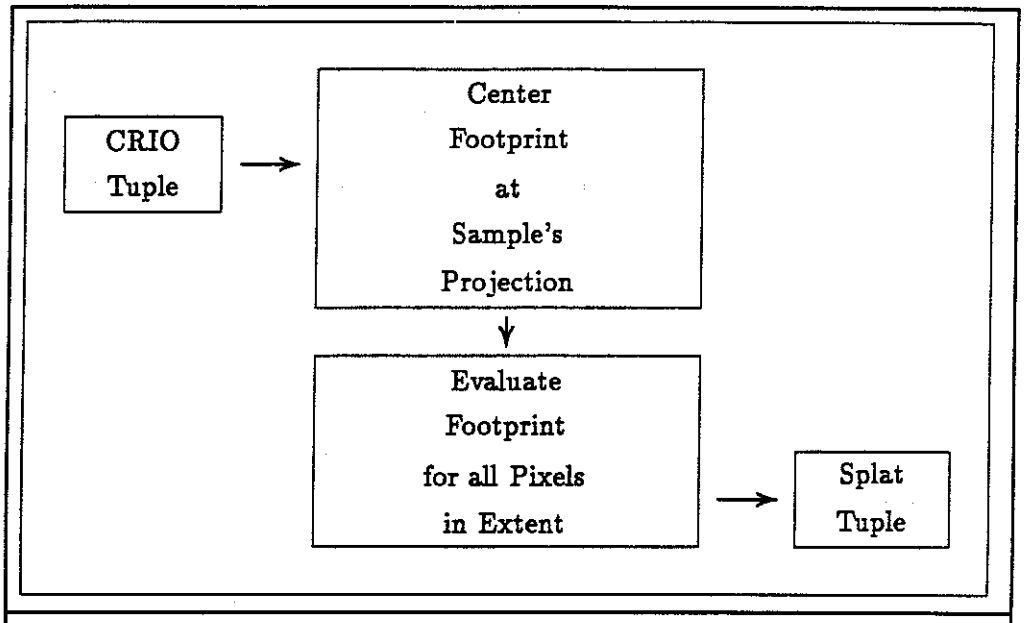


Figure 3.07: Block Diagram of the Reconstruction Process.

The renderer must reconstruct a continuous signal from discrete samples during the volume-rendering process, as shown in Figure 3.07, so it can resample the signal at image resolution.

3.7.1 Footprint Function

As discussed in Chapter 2, reconstruction is performed by convolving the reconstruction kernel with the sampled signal. Let $f()$ be the input samples, $h()$ the reconstruction kernel, and $g()$ the reconstructed signal. Let i, j, and k range over the input mesh. The volume reconstruction equation for a discrete input, $f(i,j,k)$, is

$$g(x,y,z) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} f(i,j,k)h(x-i, y-j, z-k).$$

This equation implies that each output point is a weighted average of many input points.

Instead of considering how multiple samples contribute to a pixel, consider how a sample contributes to multiple pixels. The effect a single input sample $< i,j,k >$ has at point $< x,y,z >$ is

$$effect_{(i,j,k)}(x,y,z) = f(i,j,k) \times h(x-i, y-j, z-k).$$

For speed, the renderer models the contribution from a single input to a given $< x,y >$ screen location by the sum of the input's contribution to all the points in z that lie behind the $< x,y >$ screen location:

$$effect_{(i,j,k)}(x,y) = \int_{-\infty}^{\infty} f(i,j,k) \times h(x-i, y-j, z-k)\, dz.$$

Here the renderer departs slightly from the ideal process. If visibility were an additive process, the above would be correct because it would not matter in what order the sample arrived at the image plane. However, visibility is not an additive process and collapsing the three-dimensional kernel into a two-dimensional footprint prevents the visibility process from accurately modeling the effects of part of the sample's contribution being in front of the other parts of the sample's contribution.

In the case of scattering, the intensity exiting a sample would be the intensity entering the sample multiplied by an exponential decay function, which models light traveling through a scattering medium (section 2.8).

$$I_{exit} = I_{enter} \times exp[-\int_{enter}^{exit} K_\lambda(S)\, dS]$$

Instead of scattering light throughout the sample, the above approximation combines a sample's density into a single value and treats the sample as a discrete semi-transparent layer. The difference in these two approaches is discussed in section 2.8.

For a given sample, $f(i,j,k)$ is a constant. Moving it outside the integral gives

$$effect(x,y) = f(i,j,k) \int_{-\infty}^{\infty} h(x-i, y-j, z-k)\, dz.$$

Now the integral is independent of the input and is only dependent on the reconstruction kernel. Define the footprint function centered at the origin as

$$footprint(x,y) = \int_{-\infty}^{\infty} h(x,y,z)\, dz.$$

Thus, the footprint is a two-dimensional image-space projection of the reconstruction kernel.

3.7.2 Method

For orthographic views of rectilinear meshes, the footprint of each sample is identical except for an image-space offset. Therefore, the renderer need only calculate the footprint function once for each view of the data set. Once the renderer calculates the footprint, it can evaluate the footprint function at each pixel that lies within the footprint's extent to determine the the effect this sample has that pixel. The renderer weights the sample color and opacity by the footprint value and adds that amount to the pixel.

The weight at pixel $< x, y >$ for a footprint centered at $< i, j >$ is

$$weight(x,y) = footprint(x-i, y-j).$$

Evaluating the footprint function requires an integration. The renderer could use numerical integration techniques, because many kernels are difficult to integrate analytically. This is a compute-intensive operation, and we do not want the renderer to recompute the footprint every time the renderer needs to use it. Since the footprint function is constant

for all the input samples, the renderer builds a footprint table once, at the beginning of the rendering process, and uses the footprint table for every sample.

The easiest way to build the table is to first determine the image-space extent of the projection of the reconstruction kernel and then select a sub-pixel sampling rate. The renderer chooses a sub-pixel sampling rate based on how many pixels the sample affects. If the sample affects a large number of pixels, the samples are probably far apart in image-space and do not need a high-degree of sub-pixel resolution. If the sample affects only a small number of pixels, the samples are probably near each other in image-space and small errors in footprint placement will cause visible artifacts. For these reasons the renderer builds a footprint table that has a sub-pixel sampling rate that ranges between one and eighty-one samples per pixel. The sampling rate is chosen so the size of the footprint table is about 128 x 128 total samples. The renderer builds the footprint table on a mesh with many samples per pixel, because without over-sampling, rendering artifacts will occur [Whitted 83]. The renderer integrates the kernel at each of these sub-pixel table locations. This reduces the number of required integrations, but the renderer must still integrate the reconstruction kernel to build the footprint table once per image.

An alternate way to generate the footprint table is to model the result of the integration with some simple function—for example, a Gaussian, because the result of integrating one dimension of a three-dimensional Gaussian is itself a Gaussian. In this way, the renderer evaluates the simple function at the table entries and does no integration.

The renderer must calculate two things to build the footprint table in this way. First, it calculates the image-space extent of the projection of the kernel. If the extent of the kernel is infinite, the renderer truncates the kernel where the kernel is sufficiently close to zero. The decision of where to truncate the kernel has a rendering-time vs. image-quality trade-off. Truncating the kernel quickly decreases the rendering time but adversely effects image quality, as shown in Chapter 2. Truncating the kernel far from its center increases the quality of the images but causes the renderer to make more computations. Second, the renderer calculates a mapping from the footprint extent to an extent that surrounds the approximating function. These two calculations are presented in section 3.7.4.

Once the renderer builds the footprint table, it centers the footprint table at each sample's projected screen position, X in Figure 3.08. This footprint has an extent of three by three pixels, the crosses. The footprint has a sub-pixel sampling rate of three by three sub-samples per pixel (the dotted lines). When the footprint is centered on X, exactly nine pixel centers fall within the extent of the footprint. It calculates the image-space extent of the kernel by offsetting the transformed region extent to the projected screen position. Then for each pixel in the extent, labeled O in Figure 3.08, the renderer samples the footprint table to determine the weight for this sample and this pixel. The weighted sample is passed to the visibility process.

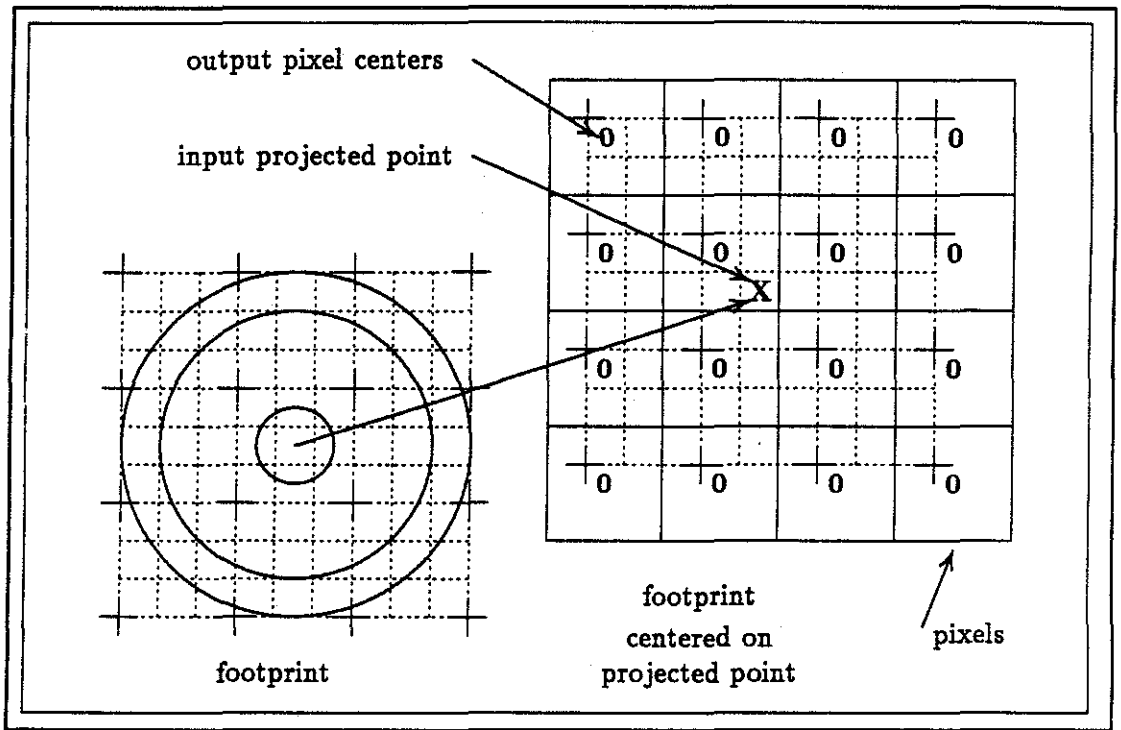


Figure 3.08: Use of Footprint Function.

3.7.3 Using the Approximation Function

This splatting method assumes that the extent of the reconstruction kernel is a sphere. If the extent is not a sphere, the renderer bounds the kernel by a sphere so it may calculate the extent using the sphere methods. For efficiency reasons, the bounding sphere should be as tight as possible. A loose-fitting sphere will cause the renderer to build a footprint table that has many zero entries, and to visit many pixels that a sample does not affect. For a spherical kernel, the radius of the sphere is equal to the width of the reconstruction kernel. This sphere, called the *unit region sphere*, as seen in Figure 3.09, defines the three-dimensional region that a sample will effect. The projection of the unit region sphere on the image plane is a circle called the *unit region extent*, as seen in Figure 3.09. The renderer does not integrate the actual reconstruction kernel over the unit region extent, but models the result of the integration with a simple function—for example, a Gaussian. The function which models the result of the integration is called the *approximating function.*

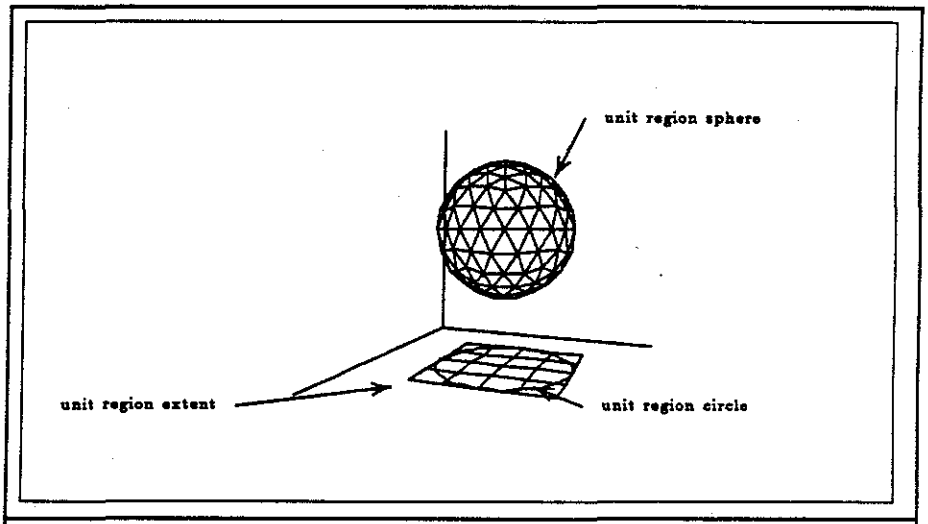


Figure 3.09: Unit Region Kernel.

During image generation, the renderer transforms the unit region sphere by the view-transformation matrix and generates the *transformed region ellipsoid*, as seen in Figure 3.10. If the sampling rates in the mesh directions are equal and the image is scaled equally in both image dimensions, then the transformed region ellipsoid is a sphere. The renderer then projects the transformed region ellipsoid onto the image plane and calculates the extent of the projection, called the *transformed region extent*, as seen in Figure 3.10. This projection is always an ellipse in an orthographic projection, as shown below in section 3.7.4. The ellipse is called the *projected region ellipse*, as seen in Figure 3.10. The renderer also calculates a mapping from the transformed region extent to the unit region extent.

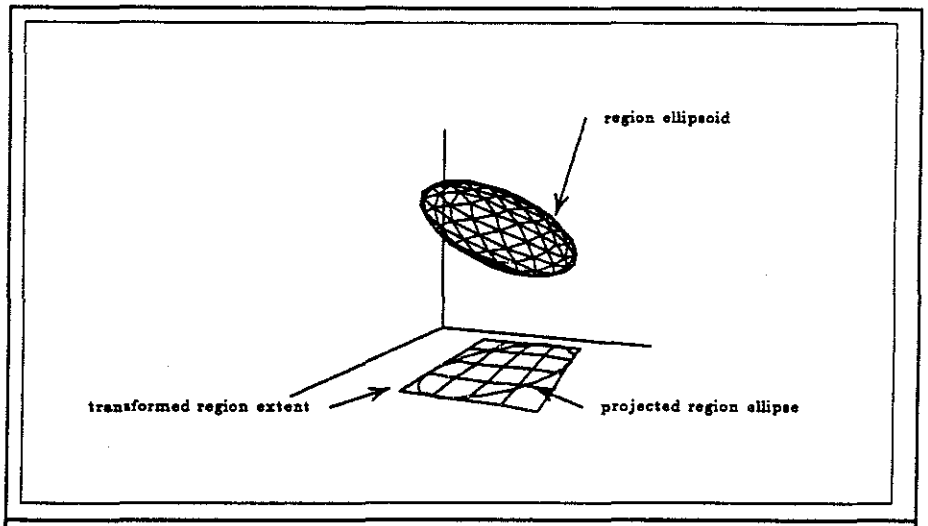


Figure 3.10: View-Transformed Kernel.

To build the footprint table, the renderer maps each sub-pixel table entry from the transformed region extent onto the unit region extent, evaluates the approximating function and stores the result in the footprint table. This table is built once for an image and is used by the reconstruction process each time it operates on a CRIO tuple. This step contains several approximations that produce rendering artifacts which are described in Chapter 5.

3.7.4 Extents and Mappings

The renderer must determine extents and mappings for two cases: when the unit sphere maps to a sphere after applying the view-transformation matrix, and when the unit sphere maps to an ellipsoid. The result is a sphere when the input volume has equal spacings in each of the mesh directions and the view-transformation matrix has uniform scaling. The result is an ellipsoid when the input volume has non-uniform spacing in each of the mesh directions or the view-transformation matrix has non-uniform scaling. Since a sphere is a special case of an ellipsoid, the renderer uses the elliptical method described below for all volumes.

The projection of transformed region ellipsoid is always an image-space ellipse. The extent of a kernel's effect is the extent of the projected region ellipse and the mapping from the footprint table to the unit region circle is a mapping from the projected region ellipse to the unit region circle.

The renderer builds the region ellipsoid by scaling the unit region sphere by the mesh spacing scaling factors and then transforming by view-transformation matrix. By treating the unit region sphere as a quadric surface, these transformations can be represented mathematically by matrix multiplications.

Let the original unit sphere be U:

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The *mesh spacing scaling factors* are the relative scales of the spacing in each mesh direction. For example, computed tomography data is usually sampled at a higher rate in the x and y directions than in the z direction. This causes the spacing in the three mesh directions to be different. Let S_x be the scale factor in the x direction, S_y be the scale factor in the y direction, and S_z be the scale factor in the z direction, then the mesh spacing scaling factor is

$$S = \begin{pmatrix} S_i & 0 & 0 & 0 \\ 0 & S_j & 0 & 0 \\ 0 & 0 & S_k & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The view-transformation matrix is

$$V = \begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The mesh space region ellipsoid is

$$E = S\ U.$$

The renderer calculates both the inverse view-transformation matrix and its transpose to transform the quadric surface, E. The resulting region ellipsoid is

$$R = V^{-1^T}\ E\ V^{-1}.$$

The region ellipsoid, R, can be written as

$$R = \begin{pmatrix} A & \frac{D}{2} & \frac{E}{2} & 0 \\ \frac{D}{2} & B & \frac{F}{2} & 0 \\ \frac{E}{2} & \frac{F}{2} & C & 0 \\ 0 & 0 & 0 & -K \end{pmatrix}.$$

This gives an ellipsoid defined by

$$(x, y, z, 1)\ R \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0,$$

or

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz - K = 0. \tag{1}$$

By rearranging terms, completing the square, and solving for x and y, the renderer can calculate the image-space extent of the transformed ellipsoid. The x and y extents are

$$x = \pm \sqrt{\frac{K}{A - \frac{D^2}{4B} - \frac{(E - \frac{DF}{2B})^2}{4(C - \frac{F^2}{4B})}}} \quad \text{and} \quad y = \pm \sqrt{\frac{K}{B - \frac{D^2}{4A} - \frac{(F - \frac{DE}{2A})^2}{4(C - \frac{E^2}{4A})}}}.$$

The renderer also needs to calculate the mapping from the projection of the region ellipsoid back to the unit circle. To do this, the renderer first calculates the projected region ellipse from the region ellipsoid. This projection is an ellipse. To find the ellipse, first rewrite (1) as a quadratic in z. The quadratic is

$$Cz^2 + (Ex + Fy)z + (Ax^2 + By^2 + Dxy - K) = 0.$$

Points on the edge of the projection of the region ellipsoid, R, have only one root in this quadratic. Only one root to the quadratic $at^2 + bt + c = 0$ exists when $b^2 - 4ac = 0$ or when

$$(Ex + Fy)^2 - 4C(Ax^2 + By^2 + Dxy - K) = 0.$$

Grouping the x^2, the y^2, and the xy terms gives the projected region ellipse

$$Xx^2 + Yy^2 + Zxy - K = 0, \tag{2}$$

where

$$X = A - \frac{E^2}{4C}$$

$$Y = B - \frac{F^2}{4C}$$

$$Z = D - \frac{EF}{2C}.$$

Once the renderer calculates the projected region ellipse, it can define a transformation that takes points from the ellipse into the unit circle. This is the inverse of the mapping that takes the unit circle into the ellipse. To calculate the second mapping, the renderer needs to calculate two things: the amount to scale along the x and the y axes, and the amount of rotation about the view direction, illustrated in Figure 3.11.

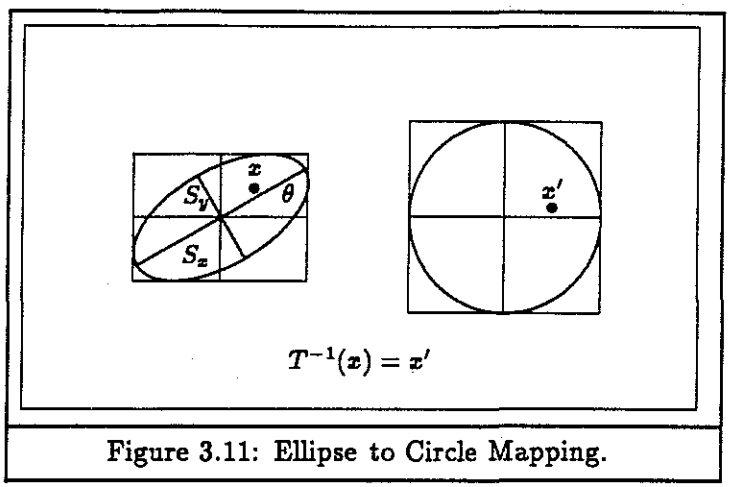


Figure 3.11: Ellipse to Circle Mapping.

The renderer finds these values so it can calculate the mapping T^{-1}, which takes points in the projected region ellipse onto the unit circle. An ellipse can be generated by rotating and scaling a circle and this operation can be expressed mathematically as

$$P = T\ U\ T^T, \tag{3}$$

where T is the transformation matrix containing the rotation and the scaling, and U is the unit circle expressed as a quadric surface. Since we know P, the projected region ellipse and U, we solve the above equation for T.

The projected region ellipse is

$$P = \begin{pmatrix} X & Z & 0 & 0 \\ Z & Y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

Let T be

$$T = \begin{pmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

U is

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We know the values for X, Y, and Z from (2). We also know, from (3), that X, Y, and Z are

$$a^2 + b^2 = X, \tag{4}$$

$$c^2 + d^2 = Y, \tag{5}$$

$$ac + bd = Z. \tag{6}$$

There are three equations and four unknowns, a, b, c, and d. However, T is a matrix that is a scale in the x and y directions followed by a rotation about the z axis. So we also know

$$a = S_x \times \cos\theta, \tag{7}$$

$$b = -S_x \times \sin\theta, \tag{8}$$

$$c = S_y \times \sin\theta, \tag{9}$$

$$d = S_y \times \cos\theta. \tag{10}$$

Plugging (7), (8), (9), and (10) into (4), (5), and (6) and applying some algebraic manipulation produces

$$\frac{(X-Y)}{Z} = \frac{\cos\theta}{\sin\theta} - \frac{\sin\theta}{\cos\theta}. \tag{11}$$

This seems to be a problem when Z is zero but upon investigation of P, when Z is zero, P is a scaling of U. Therefore T is simply

$$a = \sqrt{X}, d = \sqrt{Y} \qquad \text{and} \qquad b = c = 0.$$

When Z is nonzero, let

$$G = \frac{(X-Y)}{Z} \qquad \text{and} \qquad w = \frac{\cos\theta}{\sin\theta} \tag{12}$$

then plugging (12) into (11) produces

$$G = \frac{(w^2-1)}{w} \qquad \text{or} \qquad w^2 - Gw - 1 = 0.$$

Using the quadratic formula, w is

$$w = \frac{G \pm \sqrt{G^2+4}}{2}$$

Given w, θ is $arctan(\frac{1.0}{w})$. This gives both $\sin\theta$ and $\cos\theta$ and allows the renderer to solve for S_x and S_y using (7), (8), (9), and (10). The above equations leave S_x and S_y undefined when $\theta = 45$ degrees. When this occurs, the renderer rotates the view an additional 0.01 degrees about the view direction. This fudge allows the renderer to calculate S_x and S_y with little effect on the image.

With θ, S_x, and S_y the renderer builds T by multiplying the identity matrix by a scale matrix of S_x and S_y, followed by a z rotation matrix of θ. The mapping from P into U is then the inverse of T : T^{-1}.

The renderer uses T^{-1} to map entries from the footprint table to the approximating function.

3.8 Visibility

There is one major difficulty in trying to model the interaction of light through the absorbing, emitting, and scattering media of a volume data set. The absorption, emission, and scattering of energy is occurring at every point in the volume. A complete solution to the problem requires knowledge about the properties of every point of volume and this makes the exact solution intractable [Siegel 72, page 141]. Approximations to the ideal solution typically study the effects on the light along a single path through the volume. As light passes through a volume element, its intensity is reduced by absorption and scattering. The change is dependent on how much volume the light passes through and on the local properties of the volume. Given a coefficient of proportionality K_λ, called an *extinction coefficient*, the decrease in intensity caused by traveling a step along a path S is given by

$$\Delta_{Intensity} = -K_\lambda(S) \times Intensity \times \Delta_S \tag{13}$$

Integrating (13) along a path S that enters and exit a volume gives

$$I_{exit} = I_{enter} \times exp[-\int_{enter}^{exit} K_\lambda(S)\, dS] \tag{14}$$

Therefore, the intensity of light is attenuated exponentially while passing through an absorbing-scattering medium [Siegel 72, page 413]. This shows the nonlinear nature of visibility because the visibility operator violates both the superposition,

$$exp(a) + exp(b) \neq exp(a+b),$$

and scalability,

$$A \times exp(a) \neq exp(A \times a),$$

criteria of linear systems.

Volume rendering methods often model (14) with the composite operator. Let $Intensity_{front}$ be the intensity of light a volume element is emitting and $Opacity_{front}$ be the amount of light a volume elements blocks. Let $Intensity_{back}$ be the amount of light coming from behind the volume element. Then $Intensity_{out}$, the amount of light the volume element sends to the viewer, is

$$Intensity_{out} = Opacity_{front} \times Intensity_{front} + (1.0 - Opacity_{front}) \times Intensity_{behind}.$$

Sabella [Sabella 88] and Max [Max 90] have pointed out that as sample spacing in the z direction approaches zero, and the compositing operator calculates the amount of light emanating from increasingly thinner volume elements, the compositing operation gives equivalent results to the integration techniques.

Splatting uses the compositing operator to perform visibility because it receives discrete samples from the rendering pipeline and because compositing is much less compute-intensive than integrating the actual exponential.

In volume rendering, the extinction coefficient or opacity is constantly changing for each sample in the data set. In addition, the exponential decay function described above

is itself an approximation to the intractable calculation of how light travels through an absorbing, emitting, semi-transparent medium. Since the medium is constantly changing properties, it is impossible to characterize the errors introduced by collapsing a sample's contribution into a two-dimensional footprint and using the compositing operator to calculate visibility.

3.8.1 Method

The visibility process receives a color and opacity for each sample and a weight for each pixel in that sample's extent. For each pixel in the footprint, the renderer weights the sample by the footprint value and composites the sample's color and opacity into the accumulation buffer.

The visibility rules are different for front-to-back and back-to-front traversals, but the rules are equivalent.

Let I denote the intensity, A denote the opacity, o denote the output, c denote what is currently in the accumulation buffer, and n denote the new sample.

For a front-to-back traversal, the formulae are

$$I_o = I_c + ((1 - A_c) \times (I_n \times A_n))$$

and

$$A_o = A_c + ((1 - A_c) \times A_n).$$

For a back-to-front traversal, the formulae are

$$I_o = ((1 - A_n) \times I_c) + (I_n \times A_n)$$

and

$$A_o = ((1 - A_n) \times A_c) + A_n.$$

The initial implementation of the visibility process calculated the footprint for each sample and composited the sample's footprint onto the accumulation buffer sample-by-sample [Westover 89]. This does not properly model reconstruction (section 5.3) as neighboring samples whose kernels overlap are treated independently. Reconstruction is an additive process and the value of points between two samples is a combination of both sample's values. The renderer can not treat the samples independently because visibility does not adhere to superposition. For example, in a back-to-front traversal samples that arrive later at the visibility process have visibility priority over samples that arrive earlier even though these samples should sum their contributions in the overlap region before the visibility process composites the result onto the accumulation buffer.

To address the problem of correctly calculating the value of a reconstructed point that lies between samples with overlapping kernels, we introduced the sheet buffer method. A sheet buffer is a buffer the size of an image that acts as storage for the reconstruction process. As the renderer reconstructs a sheet of samples (section 3.5.1) if first zeros the sheet buffer between sheets and then simply adds the weighted contribution for each

sample into the sheet buffer over the sample's footprint. When the renderer finishes with all the samples in a single sheet, the visibility process composites the entire sheet onto the accumulation buffer. This division of reconstruction and visibility has two benefits. First, it allows both the reconstruction process and the visibility process to more accurately model convolution and energy attenuation respectively. Second, the compositing operation is performed coherently. This reduces the number of calculations that the reconstruction and the visibility processes perform (section 5.3). The cost of the sheet buffer method is the memory cost of storing the sheet buffer.

Chapter 4

Enhancements

4.1 Introduction

Two enhancements can increase the utility of the splatting algorithm. First, the approach permits successive refinement. The method can quickly generate preview images, so that users can judge the effect of changing viewing parameters without having to wait for the renderer to complete the entire image. If the user does not change the viewing parameters, the image improves by successive refinement. If he does, rendering begins anew with a fresh preview image. Second, the feed-forward algorithm lends itself to parallel execution, which increases the rendering speed.

4.2 Successive Refinement

One way to improve the apparent update rate of image generation is to display partial images during image generation. This allows the user to view the data with the current viewing parameters quickly. The user can change a viewing parameter without waiting for the image to be completely rendered, and image generation restarts with the new viewing parameters [Bergman 86]. The splatting renderer builds preview images three ways: by displaying partial images which are incremental updates of the accumulation buffer, by displaying low-quality but complete images generated by changing the reconstruction kernel, and by displaying partial but unbiased images generated by subsampling the input data.

Ideally, when a researcher uses an interactive renderer to explore a static data set, the renderer should give immediate feedback whenever any viewing parameters change. These viewing-parameter changes can include the viewing position, the clipping planes, or the shading function. The feedback need not be a high-quality full-resolution image which may take many seconds for the renderer to compute, but should be of good enough quality for the researcher to judge the effect of the change. The preview update should be fast enough so that the lag in response will not destroy effective interaction. For example, when the user is rotating the data set with a mouse, rotation should not continue after mouse movement stops.

4.2.1 Incremental Updates

The feed-forward method uses two incrementally-updated buffers that can be displayed during image generation: the sheet buffer and the accumulation buffer. The renderer uses the sheet buffer when reconstructing one sheet of the data. It fills this buffer with zeros between sheets and then splats each sample of the sheet into the sheet buffer

by adding the sample's weighted contribution over the sample's footprint. This buffer changes quickly and contains information about only a single sheet of data. The renderer also maintains the accumulation buffer. After the renderer finishes calculating the sheet buffer, it composites the sheet buffer onto the accumulation buffer. The accumulation buffer contains the image that would result if the renderer stopped processing samples. This buffer is updated many times per second and provides interesting views that look as if a clipping plane is moving through the data either back-to-front or front-to-back depending on the rendering order.

4.2.2 Footprint Extent: Speed vs. Quality

The most compute-intensive part of the feed-forward renderer for high-quality images is the reconstruction process. Since the renderer performs a multiply and an add for each pixel within the extent of a sample's footprint, the footprint's extent determines the amount of computation required to use the footprint. Three factors affect the size of the footprint's extent: the relative spacing in the three mesh directions, the scale factor, and the kernel's spatial extent.

Many data sets have different sampling rates the the three mesh directions. For example, computed-tomography data sets are sampled many more times in the cross-sectional direction than the longitudinal direction. The renderer has to scale the footprint to adjust for the difference so it can preserve the correct aspect ratio of the data. If an image was generated of an axis-aligned data set and the data set has twice as many samples in the x direction as the y direction, the footprint would have to be twice as large in the y direction because of the absence of samples in the y direction. The renderer sets the minimum spacing to one and then normalizes the other two spacings. These spacings are called the *mesh factor*.

The *scale factor* is the ratio of pixel sampling rate to image sampling rate—for example, a scale factor of two would cause the renderer to magnify the input mesh spacing by two and generate an image that has twice the resolution of the input in both image dimensions. Thus, increasing the scale factor increases the size of a footprint's extent in pixels. As the scale factor increases by a factor of N, the renderer must do N^2 more calculations. Scale factors of one cause the renderer to generate images that have the about same pixel resolution as the data–for example, the renderer would generate an image that has between 64 x 64 and 110 x 110 covered pixels for a 64 x 64 x 64 data set. The difference is due to possible data rotations. The small size is for axis-aligned view and the large size is for a data set that was rotated 45° in each of the x, y, and z directions. If a user wanted larger coverage in his image, he would increase the scale factor. A scale factor of three causes the renderer to generate a 192 x 192 image in the pixel-aligned case and a 332 x 332 image in the rotated case.

Different filter kernels have different extents independent of the scaling factor. The filters described in section 2.7 have extents which cover an actual range from infinite to a single pixel and cover a practical range from 7 pixels to a single pixel. Changing the reconstruction kernel can have large effects on computational requirements as shown below in this section.

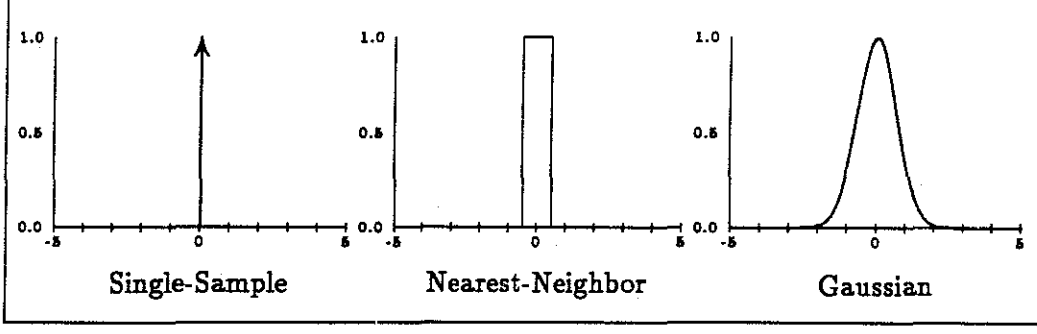


Figure 4.01: The Three Kernels.

A low-quality and computationally-inexpensive kernel is the single-point kernel, as illustrated in Figure 4.01, where the renderer maps a sample to a single pixel. The footprint for this function is always one pixel, a spatial impulse function, and the footprint computation requires only a single multiply and add per sample per color channel. A problem with the single-point kernel is that for certain rotations, such as 45° about the viewing axis or for scale factors larger than one the image will have holes as seen in the upper left image of Figure 4.02.

An intermediate-quality and moderately computation-intensive kernel is the nearest-neighbor kernel, as illustrated in Figure 4.01, where the renderer maps a sample to all the pixels that are closer to this sample than any other sample in the sheet. The extent of the nearest-neighbor kernel is approximately

$$((2 \times \lceil \textit{scale factor} \times \textit{mesh factor} \rceil) - 1).$$

The actual extent is calculated using the extent formulae of section 3.7.4, which take rotations into account. When the scale factor is one, the nearest-neighbor kernel is equivalent to the single-point kernel; however, when the scale factor is greater than one, the footprint for the nearest-neighbor kernel is larger than one pixel. For example, when the renderer uses the nearest-neighbor kernel and uses a scale factor of two, the footprint extent is three pixels square and the footprint computation requires nine multiplications and additions per sample per color channel. This kernel will not cause holes in the image, but the image will appear blocky for scale factors greater than one as seen in the upper right image of Figure 4.02.

A high-quality and computationally-expensive kernel is one using a Gaussian to determine the amount of a sample's contribution to the pixels in its extent, as illustrated in Figure 4.01.

$$f_{Gaussian}(x) = e^{-\frac{x^2}{2\sigma^2}}.$$

Since the Gaussian has infinite spatial extent, it must be truncated. This should be done far enough from the center of the Gaussian so that the value of Gaussian is small to reduce the ringing effects of truncation. The renderer chooses the value of 0.004 as the point of truncation. This value is approximately $\frac{1}{255}$, the smallest nonzero value that can be represented in 8-bit color components. Values below this threshold will have little effect on the output result. The extent of the Gaussian kernel is the number of pixels in its footprint. The *width* of the one-dimensional Gaussian kernel is how many input samples lie under the truncated Gaussian. The extent of the Gaussian kernel is approximately

$$(\lceil \textit{scale factor} \times \textit{width} \times \textit{mesh factor} \rceil).$$

The actual extent is calculated using the extent formulae of section 3.7.4, which take rotations into account. A five-sample-wide Gaussian, is one whose value falls to under 0.004 at two input samples from the center. This Gaussian has a $\sigma = 0.6$. When the renderer uses this Gaussian and uses a scale factor of two, the footprint extent is nine pixels square and the footprint computation requires 81 multiplications and additions per sample per color channel. These three kernels span an 81-to-1 range in the computational requirements for reconstruction. The corresponding rendering times are shown below in Table 4.01.

The images in Figure 4.02 are renderings of a data set using the above three classes of kernels. The image in the upper left was generated using the single-point kernel. The image in the upper right was generated using the nearest-neighbor kernel. The image in the lower center was generated using the five-sample-wide Gaussian kernel.

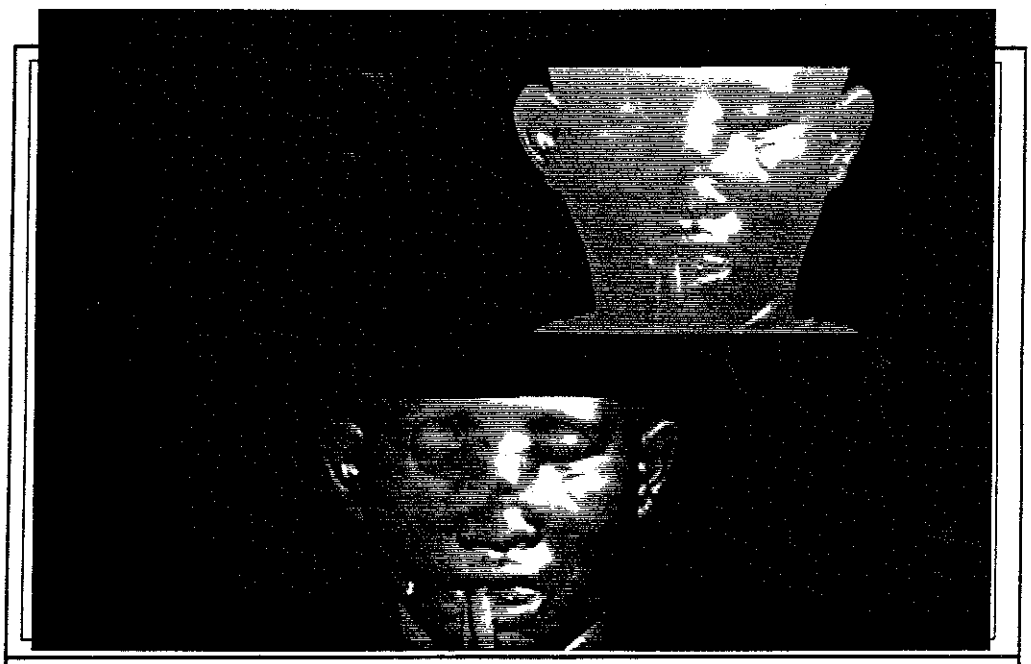


Figure 4.02: Image from the Three Classes of Kernels.

These relative differences in computational requirements do not remain constant with changes to the scale factor. The above multiplication and addition counts are for images with a scale factor of two. For a scale factor of three, the number of multiplications and additions stays at one for the single-point kernel and changes to 25 for the nearest-neighbor kernel and 169 for the five-sample-wide Gaussian kernel. Thus, for magnified images with a scale factor of three, this choice of kernels produces a 169 to 1 range in the number of reconstruction calculations.

4.2.3 Subsampled Rendering: Speed vs. Quality

Since the renderer uses a feed-forward method, the most important factor affecting rendering time is pipeline throughput. The renderer can adjust the number of samples that it processes, and thus adjust pipeline throughput, by subsampling the input data set. For example the renderer may generate a low-resolution version of the data by operating on every $4th$ input sample in each mesh direction. With a three-dimensional volume, this reduces the number of samples for the pipeline by a factor of 64. The renderer may generates a middle-resolution version by operating on every other input sample in each mesh direction. With a three-dimensional volume, this reduces the number of samples for the pipeline by a factor of eight.

The images in Figure 4.03 are images of a data set rendered at the subsampled resolutions. The low-resolution data set image is in the upper left (64x reduction), the middle-resolution data set image is in the upper right (8x reduction), and the high-resolution data set image is in the lower center (no reduction).

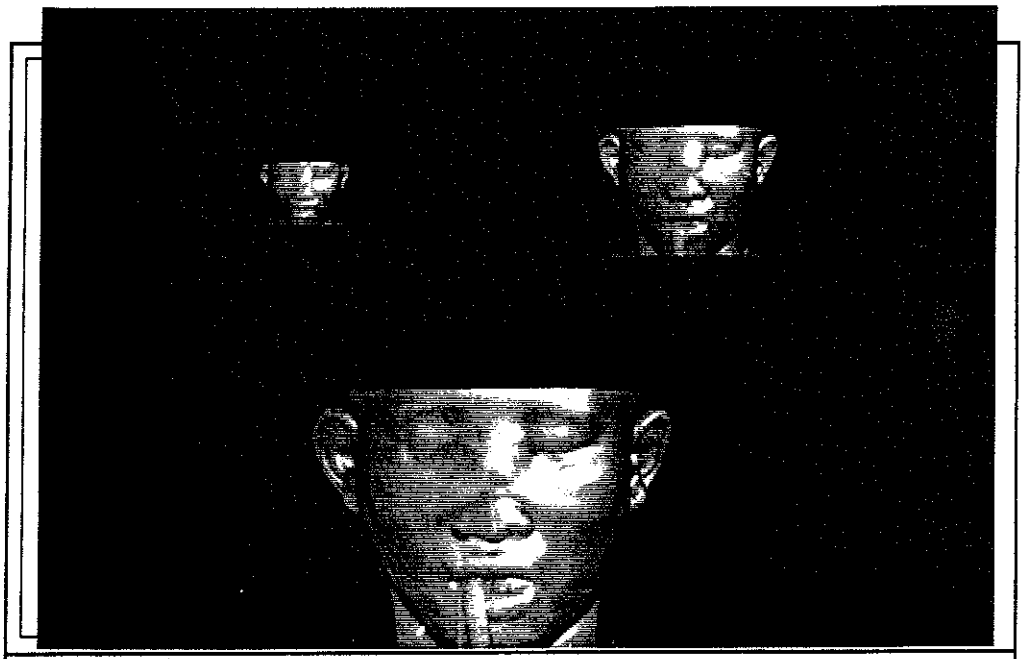


Figure 4.03: Image from the Three Resolutions.

The quality of the images of the subsampled volumes depends on the frequency content of the volume. If the original volume was band-limited and was sampled at four

times its Nyquist rate (an unlikely event) , then the subsampled volume will not introduce artifacts. More likely, the volume contains high frequency information and the subsampling process will cause this energy to alias. This violates the premise that signal processing issues need to addressed during volume rendering, however, these images of the subsampled volume are only for quick preview and will refine to high-quality images over time.

4.2.4 Rendering Computation Comparison

As seen in the previous section, the choice of kernel and choice of subsampling rate obviously have large effects on how much computation the renderer does during reconstruction. For example, consider when the renderer generates an image of a 128 x 128 x 128 input data set with a scale factor of three. When the renderer works with the low-resolution subsampled data set (32 x 32 x 32) and uses the single-point kernel, it processes 32,768 samples, each with a one pixel footprint, requiring 32,768 multiplications and additions. When the renderer works with the high-resolution data set (128 x 128 x 128) and uses the five-sample-wide Gaussian kernel, it processes over two million samples, each with a 13 x 13 footprint, requiring over 350 million multiplications and additions. This is a 10,000-to-1 difference in the computation requirements between the low-quality low-resolution preview image and the high-quality full-resolution image. These choices allow the renderer to generate a range of views trading off rendering speed for image quality as shown in Table 4.01.

Rendering Times				
Input Resolution	Kernel	Footprint Size	Time (seconds)	Shown in Figure
32x32	single-point	1x1	0.6	Not Shown
32x32	nearest-neighbor	3x5	4.1	Not Shown
32x32	Gaussian	9x17	36.9	Figure 4.03
64x64	single-point	1x1	1.2	Not Shown
64x64	nearest-neighbor	3x5	7.0	Not Shown
64x64	Gaussian	9x17	49.6	Figure 4.03
128x128	single-point	1x1	4.0	Figure 4.02
128x128	nearest-neighbor	3x5	20.2	Figure 4.02
128x128	Gaussian	9x17	109.2	Figure 4.02 and Figure 4.03

Table 4.01: Rendering Times During Successive Refinement

For the measurements in Table 4.01, the splatting renderer generated a set of sample images of a 256 x 256 x 93 computed-tomography study of a human head. The spacing

between samples is twice as large in the z direction as the x and y directions. The images are 512 x 512 pixels and the scale factor is set to two. Rendering times for the three resolutions of the data set and for the three reconstruction kernels are listed in the table.

4.3 Parallel Execution

A naive parallel feed-backward algorithm, e.g. ray-casting, requires that the renderer either replicate the entire data volume at each computation node or resend the data to the computation nodes for each frame, since the renderer cannot know what portions of the data each computation node needs for an arbitrary view. Since a feed-forward algorithm can treat each data sample independently, splatting is easily parallelizable without the need to replicate the input volume at each computation node. In addition, the processes that run on the parallel computation nodes use exactly the same algorithms that run on the single process version except that they run on a subset of the data.

All of the parallel approaches in this section assume the parallel processor is a multiple-instruction-multiple-data (MIMD) architecture without shared memory. The earliest parallel implementation of splatting ran on a collection of workstations connected via ethernet. Later, splatting was implemented on the Sun VX/MVX visualization accelerator, which is also a MIMD machine without shared memory. Parallel splatting does not require this type of architecture [Neumann 90], it was just the type of parallel machine that was available for the conduct of this research.

4.3.1 Pipeline Balance in the Initial Parallel Implementation

The most obvious way to parallelize a feed-forward process is to use functional parallelism, where each pipeline element runs as a separate process. Splatting can be done with the transformation process, the CRIO process, the reconstruction process, and the visibility process running as separate processes, where the parallel renderer passes tuples from functional block to functional block, as illustrated in Figure 4.04.

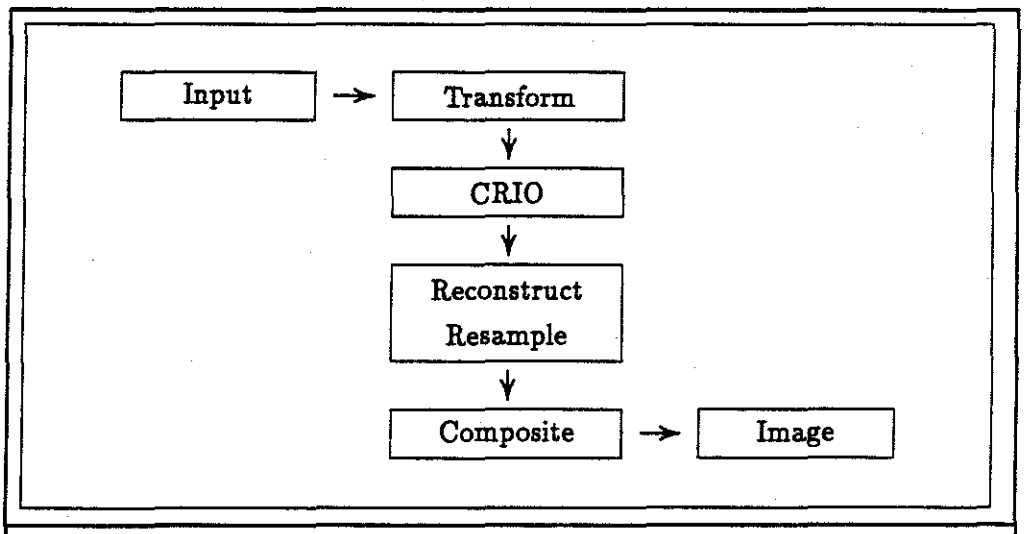


Figure 4.04: Functional Parallelism.

The problem with this brute-force approach is that the renderer must pass too much data too many times between the separate processes, some of which do relatively little work. Because the input volume is a rectilinear mesh, the transformation process may be performed incrementally, requiring only three additions per volume element. Additionally, the image-space tuple requires more storage than the input tuple, since the image-space coordinates require fractional representations and the mesh coordinates are implied by the rectilinear mesh and need not be explicitly stored. The CRIO process requires more computation than the transformation process and sends only nonzero-opacity tuples down the pipeline. Since the transformation process requires little computation and expands the amount of data while the CRIO process requires significant computation but can reduce the amount of information it sends down the pipeline, these two processes were combined into a single process.

For a 128 x 128 x 128 data set, with a 9 x 17 sample footprint, and 50% nonzero-opacity tuples, the functionally parallel approach would require the renderer to transmit approximately 2 million tuples from the transformation process to the CRIO process. The renderer would then transmit approximately 1 million CRIO tuples from the CRIO process to the reconstruction process and would then transmit approximately 1 million 9 x 17 footprints to the visibility process. The above grouping reduces the transfer to a single transmission of the 1 million CRIO tuples from the CRIO process to the reconstruction process.

In the original implementation, reconstruction and visibility were tightly coupled. As soon as the renderer calculated the footprint, it composited the sample onto the accumulation buffer. Since a footprint can be large (i.e. 9 x 17 for the lower center image in Figure 4.02), and sending the footprint requires more bandwidth than sending the the sample, these processes were combined into a single process.

This parallelization has the transformation and the CRIO processes run within a single process, and the reconstruction and the visibility processes run within a second process. The transformation/CRIO process reads the data set once and sends only tuples that have nonzero opacity to the reconstruction/visibility process. Typical volume-rendered images generate fewer than 100% nonzero-opacity tuples [Levoy 90]. For example, in Figure 4.02 only about 8.8% of the samples in the data set generate nonzero-opacity tuples.

Since the transformation/CRIO process must deal with all the input whereas the reconstruction/visibility process must only deal with the nonzero-opacity tuples, the renderer uses multiple transformation/CRIO processes. For data partitioning reasons described below, the renderer uses multiple, N^2, transformation/CRIO processes, and a single reconstruction/visibility process where N is a small integer. If N is one, the renderer consists of two processes, as illustrated in Figure 4.05, and essentially runs the single process version of the algorithm described in Chapter 3; the single transformation/CRIO process runs on the entire input volume and sends nonzero-opacity tuples to the single reconstruction/visibility process.

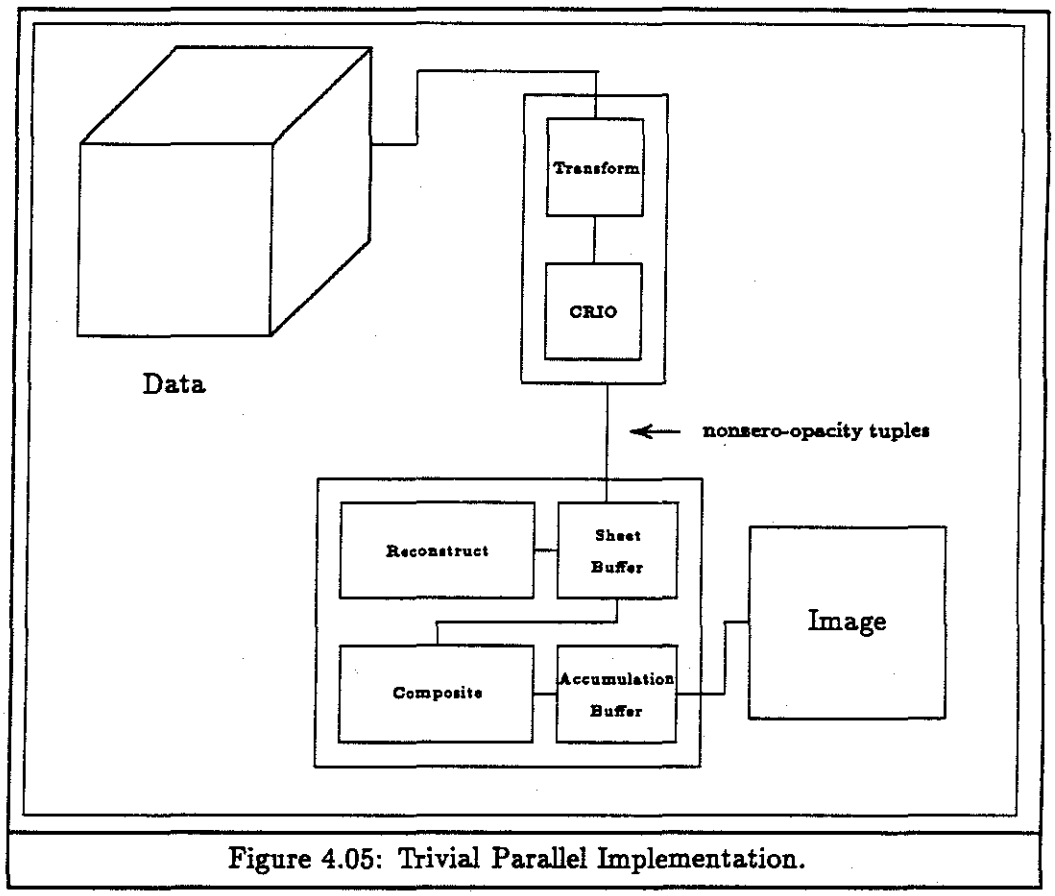


Figure 4.05: Trivial Parallel Implementation.

If there are multiple transformation/CRIO processes, the renderer distributes $\frac{1}{N^2}$ of the input data to each transformation/CRIO process. Figure 4.06 is a block diagram of the initial parallelization with $N = 2$. The renderer breaks the input data set into N^3 subsets, for reasons explained below, and passes N subsets to each transformation/CRIO process. These processes run the algorithms described in Chapter 3 on each subset. The reconstruction/visibility process monitors an input stream from each of the transformation/CRIO processes, splatting each nonzero-opacity tuple into the sheet buffer. When all of these processes have finished with a sheet of data, the reconstruction/visibility process composites the sheet buffer onto the accumulation buffer.

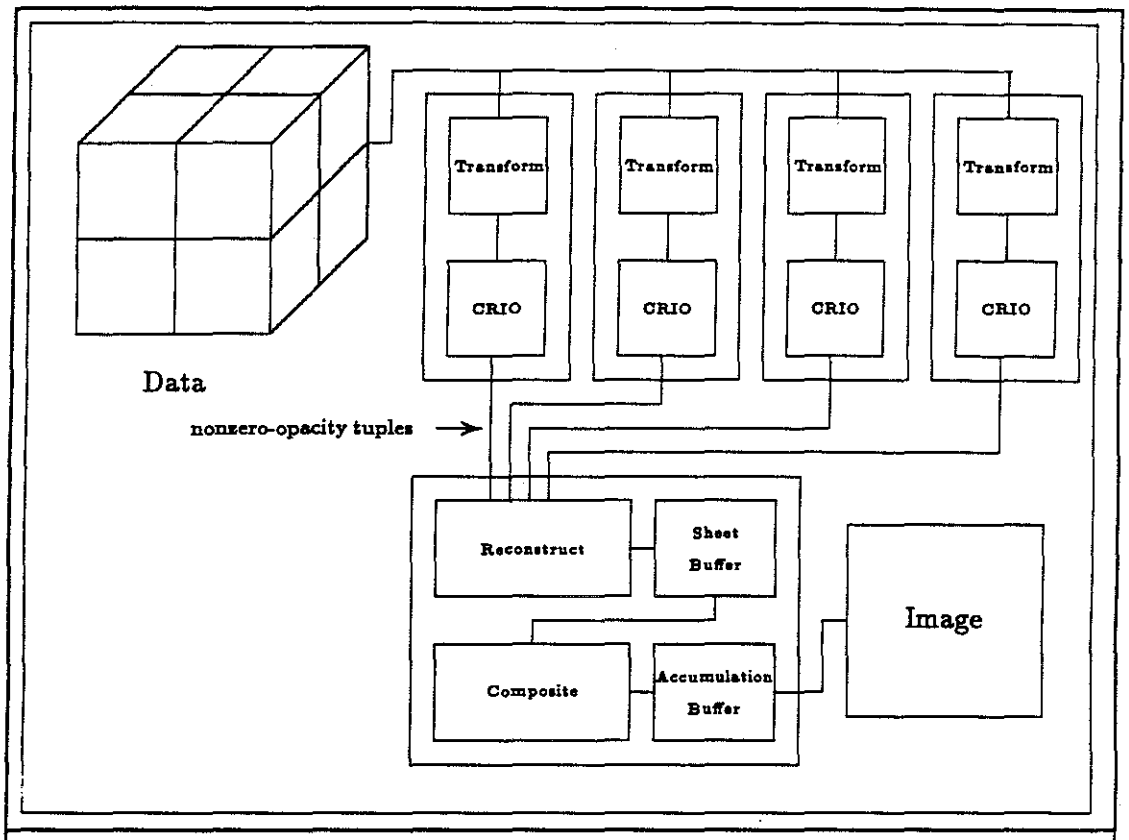


Figure 4.06: Initial Parallel Implementation with $N = 2$.

The renderer must carefully partition the data into the subsets sent to each transformation/CRIO process, because the renderer must operate on the samples in either a front-to-back or a back-to-front order for all orientations. A brute-force subdivision approach would break the data set into N^2 subsets, one for each transformation/CRIO process, each subset containing a range the slices of the data set. If the view looked down slices, this subdivision would work fine, since each transformation/CRIO process would operate on a group of rows for each sheet. However, if this view is rotated 90° about the y-axis, a single transformation/CRIO process would operate on all the samples for a single sheet, while the other transformation/CRIO processes waited for it to finish. An alternative way to subdivide the data is to divide the data into N^3 subsets by dividing the data set N times in each mesh direction. The renderer would then send N subsets to each of the N^2 transformation/CRIO processes. The renderer must still carefully partition which subsets are sent to which transformation/CRIO processes. If a transformation/CRIO process received two subsets that had samples from the same sheet, then for some view directions, that single transformation/CRIO process would be working on multiple subsets at the same time while another transformation/CRIO process would have no subsets to work on. Consider the case for $N = 3$. Number the N^2 processes 0 through 8. If the renderer sent the first transformation/CRIO process,

number 0, the lower left data subset for each z-aligned group of N^2 subsets, the process assignment would look like the left hand side of Figure 4.07. This arrangement is reasonably distributed for views looking down the z axis. However, if the view was rotated 90° about the y axis, as has happened in the right hand side, there would not be an equal distribution of work for each sheet. For example, only processes 2, 5, and 8 have any samples from the front-most sheet.

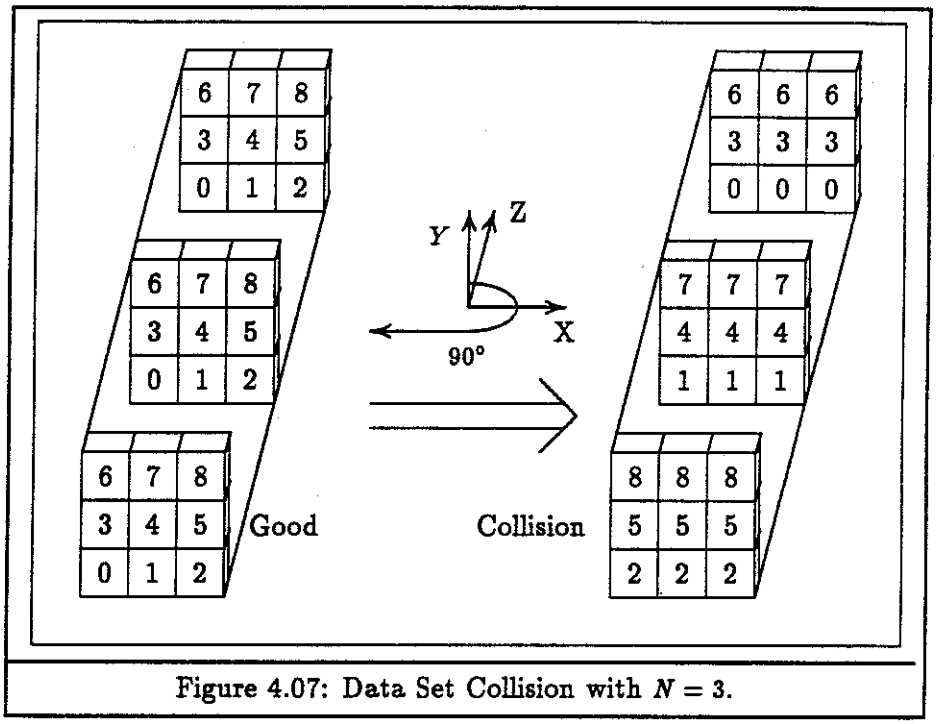


Figure 4.07: Data Set Collision with $N = 3$.

Fortunately there is a way to distribute the N^3 subsets to the N^2 processes so there is no sheet overlap in the any of three mesh directions, which allows the N^2 transformation/CRIO processes work as a group on each sheet in a front-to-back or a back-to-front order. First, the renderer numbers the N^2 processes

$$(i,j) : (0,0),(0,1),\ldots,(0,N-1),(1,0),\ldots,(N-1,N-1)$$

and the N^3 subsets

$$(i,j,k) : (0,0,0),\ldots,(N-1,N-1,N-1).$$

Then for each process, (i,j), and for k going from 0 to N-1, the renderer sends subset $< (i+k) \bmod N, (j+k) \bmod N, k >$ to transformation/CRIO process (i,j). This way, from any view direction, each process has exactly one of its subsets active for any given sheet. For example, in Figure 4.08 the subsets are distributed correctly for the left hand view; and when the view is rotated 90° around the y-axis, all nine transformation/CRIO processes are active on any given sheet.

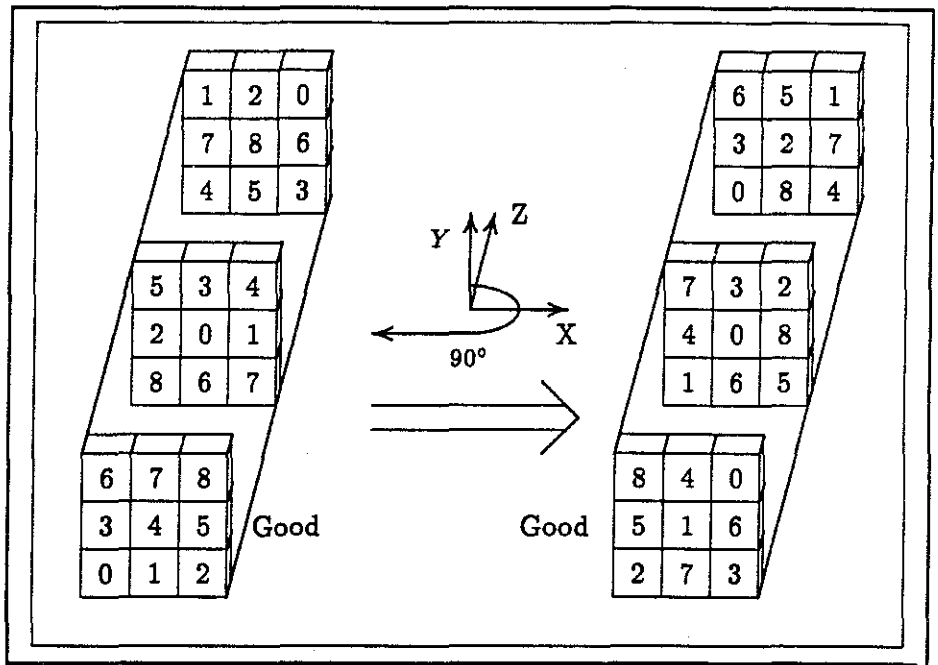


Figure 4.08: Parallel Data Distribution with $N = 3$.

A $N = 2$ parallel version of the splatting renderer, as diagramed in Figure 4.06, is running on a Sun-4/370 workstation and a Sun-VX/MVX visualization accelerator [Sun 91]. The user-interface portion of the renderer runs on the workstation; it controls all user interaction and sends the input data to the VX/MVX. The MVX has four processing nodes, each with four megabytes of local memory, and the VX has a single processing node with four megabytes of local memory. These five processing nodes are connected by a single 20 megabyte/second bus. The four processing nodes of the MVX run transformation/CRIO processes which send CRIO tuples to the single reconstruction/visibility process on the processing node of the VX. The VX also has sixteen megabytes of display memory. The reconstruction/visibility process builds the accumulation buffer, as well as the final image, in the display memory so the user can see the image incrementally update.

This implementation provides three reconstruction kernels (section 4.2.2). The first is the single-point kernel, where the renderer maps each input sample to a single output pixel. The second is the nearest-neighbor kernel, where the renderer maps a sample to all the pixels that are closer to this sample than any other sample in the sheet. The third is the Gaussian kernel, with a $\sigma = 0.6$, that is truncated at two input samples from the center.

This version of the parallel renderer has load-balancing problems. When the renderer is using a small reconstruction kernel and a complicated CRIO process, the transformation/CRIO process dominates the computation and runs most efficiently with a large N. When the renderer uses a high-quality reconstruction kernel and a simple table-driven

CRIO process, the reconstruction/visibility process dominates the computation, and the renderer underutilizes the transformation/CRIO processes.

Table 4.02 demonstrates this phenomenon. Rendering times on the above implementation are given for various stages of the rendering process for generating the images in Figure 4.02. First, the "CRIO" value is the time required by the transformation/CRIO processes to generate CRIO tuples from the input tuples. Second, the "send" value is the time required by the transformation/CRIO processes to send the nonzero-opacity tuples to the reconstruction/visibility process. Third, the "splat" value is the time required by the reconstruction process to build the sheet buffers. Fourth, the "composite" value is the time required by the visibility process to composites the sheet buffers onto the accumulation buffer. Finally, the total number is how long it took the entire rendering. Notice that the "CRIO" and "send" times are independent of the reconstruction process and the reconstruction process takes over two orders-of-magnitude longer to run for the high-quality Gaussian kernel than for the low-quality single-point kernel.

Rendering Times						
Kernel	Footprint Size	CRIO (seconds)	Send (seconds)	Splat (seconds)	Composite (seconds)	Total (seconds)
single-point	1x1	14.3	7.6	0.2	11.4	33.5
nearest-neighbor	3x5	13.9	8.0	2.5	16.8	41.2
Gaussian	9x17	14.1	7.9	55.7	23.1	100.8

Table 4.02: Stage Rendering Times for Three Kernels

4.3.2 Multiple Independent Sheet Buffers

A second parallel version of splatting, as illustrated in Figure 4.09, was designed to address this load-balancing problem. It attempts to achieve better load balancing for high-quality images. The method takes advantage of the fact that the algorithm currently uses an accumulation buffer, instead of compositing every sample independently. This version breaks the reconstruction/visibility process into its two components. The first process determines the footprint for each input tuple and adds the footprint into the sheet buffer. The second process performs the visibility computation by compositing the sheet buffer onto the accumulation buffer. In this version, the renderer distributes the sheet buffer to the $M = N^2$ processes, each holding every Mth scanline. When the reconstruction process finishes with all the samples from a given sheet, it passes its scanlines to the visibility process.

A trial implementation of the above method is running on a Sun-4/370 workstation and a Sun-VX/MVX visualization accelerator. The user-interface portion of the renderer runs on the workstation; it controls all user interaction and sends the input data to the VX/MVX. In this version, a reconstruction process as well as a transformation/CRIO process run on each MVX node. Since footprints for high-quality images are large (9 x 17

for the lower center image in Figure 4.02) and cover many scanlines, each reconstruction process does about the same amount of work. Since the sheet buffer is distributed, each transformation/CRIO process sends its CRIO tuples to all the reconstruction processes. When each reconstruction process finishes a sheet and sends its portion of sheet buffer to the visibility process, the visibility process composites the sheet buffer onto the accumulation buffer and displays it.

Preliminary experiments suggest that this parallelization has an 80% utilization of the VX/MVX processors compared to the approximately 10% utilization of the first VX/MVX approach. While this new approach requires much more bandwidth, to send the tuples from each CRIO process to each reconstruction process, the VX/MVX has plenty of bandwidth for this task.

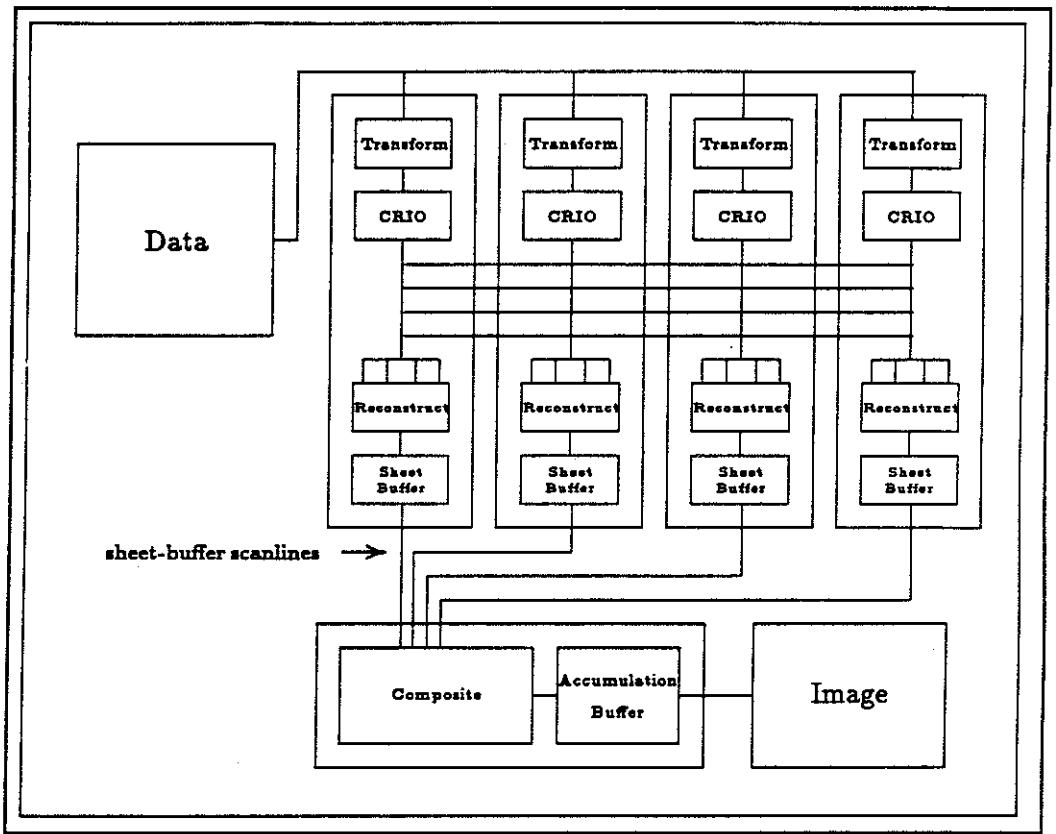


Figure 4.09: Multiple Independent Sheet Buffers with $M = 4$.

4.3.3 Proposed Parallel Implementation

The above parallel approaches require $(N^2 + 1)$ processes with N being a small integer. If a system supports $(N(N^2 + 1) + 1)$ or $(N^3 + N + 1)$ processes another parallelization is straightforward, as illustrated in Figure 4.10. Again, the renderer breaks the data set into N^3 subsets and sends one subset to each of the N^3 processes. For a given view direction, the renderer finds the proper order to traverse the data and uses this order to group the sets of N^2 subsets that contain common sheets into a slab. The renderer assigns each slab

group a single process from the N process pool. These N groups of processes, each with (N^2+1) processes, render an image for their slab in the same way as the above methods render an image. When the groups complete the slab images, they send their images to the single remaining process that combines the slab images into the final image. This method requires the renderer to reconfigure the process tree to reform the slab groups or resend the data each time the view changes, since the processes that make up a slab set may change as the view changes. With $N = 3$, there are 27 transformation/CRIO processes grouped in three sets of nine. Each of these groups feed scanlines to one of three slab-composite processes. These three slab-composite processes send their slab images to a single image-composite process that builds the final image.

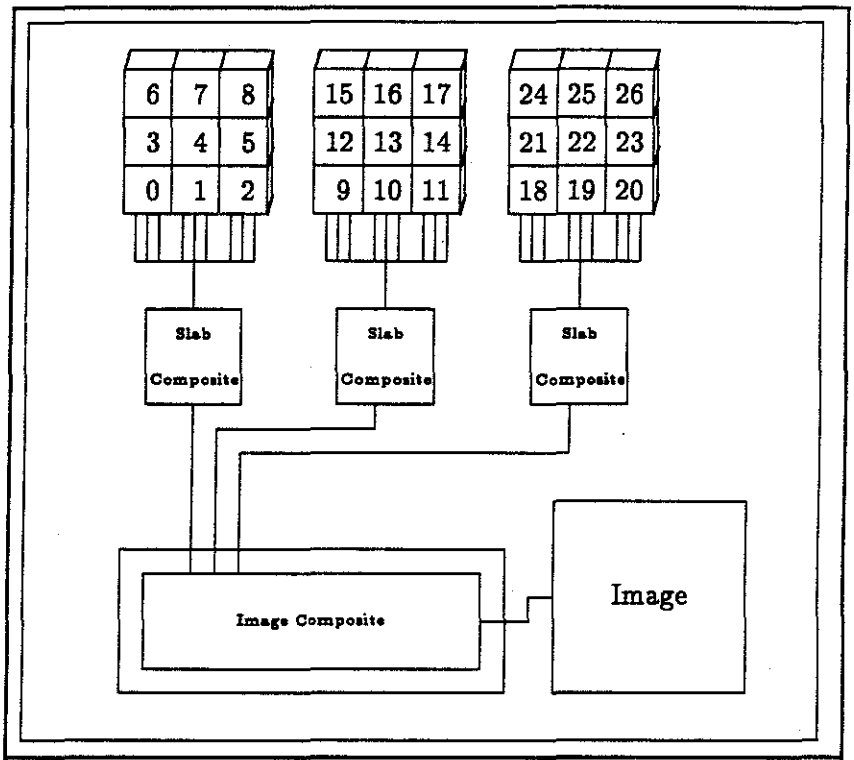


Figure 4.10: Proposed Parallel Implementation with $N = 3$.

This parallelization should run N times faster than the multiple-independent sheet-buffer approach. The final image composite step requires little time, and the N groups of processors are totally independent and are working on one Nth of the problem.

Chapter 5

Judicious Compromises

5.1 Introduction

The ideal feed-forward renderer, outlined in Chapter 2, reconstructs the assumed continuous function, $g()$, by convolving the sampled input, $\hat{f}()$, with the filter kernel, $h()$:

$$g(x,y,z) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} \hat{f}(i,j,k)h(x-i,y-j,z-k).$$

The renderer transforms $g()$ with the viewing matrix, M, to generate the view-transformed function, $v()$:

$$v(x,y,z) = Mg(x,y,z).$$

The ideal renderer shades $v()$ with shading function, $s()$, and filters the result with a low-pass filter, $l()$, to remove frequencies that exceed one-half the image sampling rate. This generates the filtered, shaded, view-transformed function $p()$:

$$p(x,y,z) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} s(v(i,j,k))l(x-i,y-j,z-k).$$

The renderer then calculates visibility for each pixel by operating on $p()$ in the region defined by the projection of that pixel into $p()$.

To increase execution speed, the actual implementation of the feed-forward algorithm introduces three design compromises which make it a less-than-ideal renderer. These compromises are:

- The renderer shades the original input samples before reconstruction instead of shading the view-transformed reconstructed signal.
- The renderer reconstructs the signal and calculates visibility at discrete steps along the region defined by each pixel's projection.
- The renderer uses a table-driven approximation of the footprint function when it reconstructs the shaded signal.

This chapter suggests a variety of implementation alternatives which deviate, to a greater or lesser extent, from the ideal in order to define an efficient renderer.

5.2 Efficient Process Ordering: Shade First

The most critical factor affecting rendering time in a feed-forward method is pipeline throughput. Due to the feed-forward approach, the renderer shades the original samples and sends only nonzero-opacity samples down the rendering pipeline. In some cases, for example Figure 3.04, as few as 8.8% of the input samples have non-zero opacity. At the other extreme, for example Figure 3.06, 40% or more have nonzero opacity. Shading the original input allows the method to run faster in many cases, and never causes the renderer to run slower.

Additionally, there are still fewer input samples than samples generated in the resampling process. because many volume-rendered images magnify the extent of the data. For example, for an input volume of 128 x 128 x 128 and a scale factor of three, the renderer shades approximately two million input samples. If the renderer reconstructed each of the 128 sheets and each sheet had 384 x 384 pixels (three times 128 x 128), the renderer shades over six million resampled samples.

This modification to the process ordering in the rendering pipeline imposes the constraint that the shading process must be band-limited, since the system does not low-pass filter the shaded signal. If the system uses a binary classifier or uses a shading process that introduces frequencies into the signal's spectrum that exceed one-half the image sampling rate, then aliasing will occur and there will be classification artifacts in the resultant images, as described in section 2.4.

5.3 Reconstruction and Visibility

There are many ways to transform the ideal method into one that is better suited to practical implementation. This section will consider four possible implementations of the reconstruction process.

First, if visibility is simply an additive process, then the transformation, reconstruction, filtering, and visibility stages are all linear, and individual samples could be treated independently throughout rendering pipeline. The renderer simply adds each tuple into the accumulation buffer and each pixel is a sum of all the intensity that exists in the region defined by that pixel's projection through the data.

Unfortunately, visibility is not merely an additive process, but is a nonlinear process that must operate sequentially on the input in either a back-to-front or a front-to-back order.

The first reconstruction method, and the method that most accurately models the ideal method, generates a three-dimensional discrete footprint for each sample and uses this three-dimensional footprint to spread a sample's energy into a volume that is axis-aligned with the image, as shown in Figure 5.01. This volume has one sample per pixel in width and height and many samples per pixel in depth. Once the volume is transformed, reconstructed, and resampled on this mesh by splatting the three-dimensional footprint of each sample, the renderer shades these samples. Since the shading process can change the spectrum of the signal, the process low-pass filters the shaded samples to lower the signal's Nyquist rate so that it is below the resampling rate. Once the volume is filtered,

the renderer uses a visibility model to determine visibility for each pixel by processing each depth column of samples that lie behind that pixel.

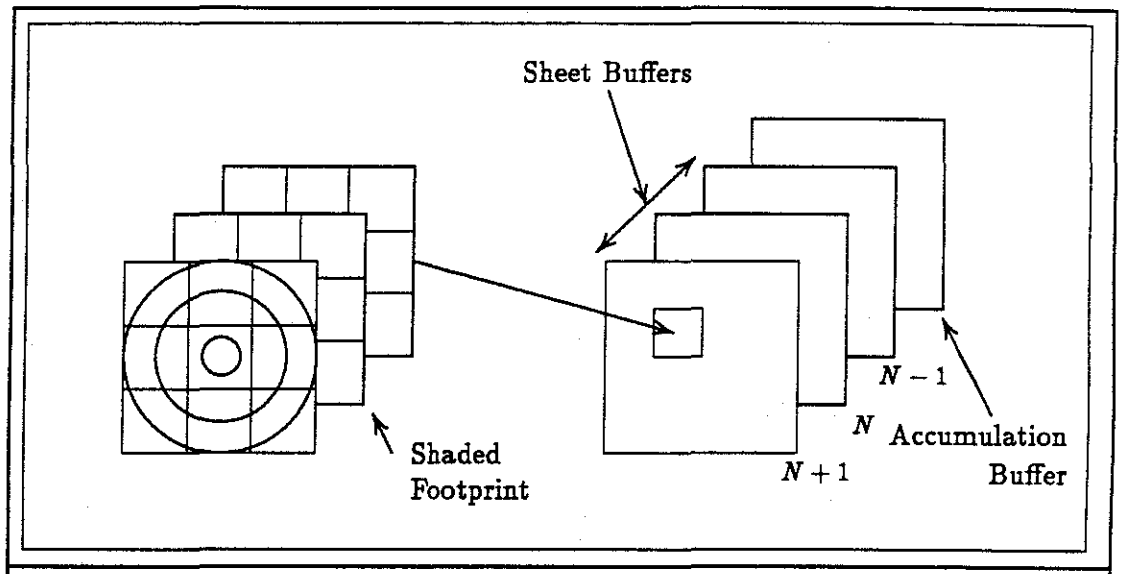


Figure 5.01: Ideal Splatting Method.

One problem with this feed-forward method is it requires a three-dimensional buffer to hold the transformed, reconstructed, and resampled volume. This buffer makes parallel reconstruction difficult, because the buffer is a single resource that has to be accessed by each splatting node. However, once the renderer creates the volume, the renderer could shade, filter, and perform visibility in parallel with simple image subdivision techniques, because the volume is pixel-aligned.

An alternative approach is to treat each sample independently even during visibility, using the *composite-every-sample* method. This version of the renderer operates on the input samples in an order that guarantees either a front-to-back or a back-to-front traversal. The renderer composites each sample's footprint into the accumulation buffer [Westover 89]. The problem with this method is each sample's energy is treated independently of other samples for each image-space point. Digital filtering theory dictates that each image-space point be a weighted average of the input samples. The renderer must add the contribution of all samples which affect an image-space point before the renderer calculates the visibility of that point.

The effect of the approximation is illustrated in Figure 5.02.

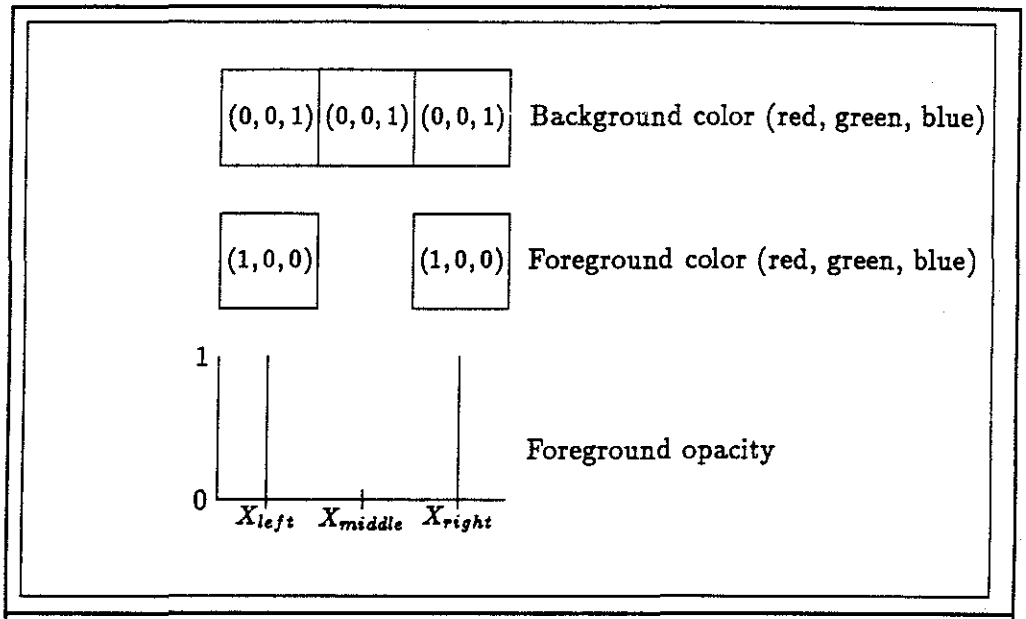


Figure 5.02: Composite-Every-Sample Problem.

If the renderer were linearly interpolating and reconstructing point X_{middle} from points X_{left} and X_{right}, then point X_{middle} should have an opacity of one and block all the data behind the point during the compositing step. A point X consists of its color, C, and its opacity, O.

In the ideal case, the opacity is

$$O_{middle} = 0.0$$

$$O_{middle} += 0.5 \times O_{left}$$

$$O_{middle} += 0.5 \times O_{right}$$

$$O_{middle} = 1.0.$$

And the color is

$$C_{middle} = O_{middle} \times C_{foreground} + (1.0 - O_{middle}) \times C_{background}$$

$$C_{middle} = 1.0 \times (1.0, 0.0, 0.0) + (1.0 - 1.0) \times (0.0, 0.0, 1.0)$$

$$C_{middle} = (1.0, 0.0, 0.0).$$

The composite-every-sample method first composites X_{left}

$$O_{middle} = 0.0$$

$$O_{middle} += 0.5 \times O_{left}$$

$$O_{middle} = 0.5.$$

$$C_{middle} = O_{middle} \times C_{foreground} + (1.0 - O_{middle}) \times C_{background}$$

$$C_{middle} = 0.5 \times (1.0, 0.0, 0.0) + (1.0 - 0.5) \times (0.0, 0.0, 1.0)$$

$$C_{middle} = (0.5, 0, 0.5).$$

This C_{middle} becomes the new background and the renderer composites X_{right}

$$O_{middle} = 0.0$$

$$O_{middle} += 0.5 \times O_{right}$$

$$O_{middle} = 0.5.$$

$$C_{middle} = O_{middle} \times C_{foreground} + (1.0 - O_{middle}) \times C_{background}$$

$$C_{middle} = 0.5 \times (1.0, 0.0, 0.0) + (1.0 - 0.5) \times (0.5, 0.0, 0.5)$$

$$C_{middle} = (0.75, 0.00, 0.25).$$

Here the incorrect ordering of reconstruction and visibility allows the blue background color to bleed through the opaque red slice.

The above observation led to the development of the sheet buffer. All the samples which are part of the same sheet are added together into the sheet buffer before the sheet buffer is used by the visibility process. This allows overlapping reconstruction kernels to add their contribution to the image-space point, which more accurately models reconstruction. A second benefit of using the sheet buffer is that the renderer performs coherent compositing. In the composite-every-sample method, the renderer must do a composite operation, $(Opacity \times Foreground + (1 - Opacity) \times Background)$, for each pixel in each sample's extent. High-quality reconstruction kernels have wide extents and a pixel may get operated on many times for each sheet. For example, when the renderer uses a five-sample-wide Gaussian reconstruction kernel, the renderer will operate on each pixel 25 times for every sheet of data. The sheet buffer method replaces the composite operator with addition and does 25 additions for each pixel within a sample's extent. Once the renderer finishes all the samples in a sheet, it composites the sheet buffer into the accumulation buffer and accesses each accumulated pixel only once.

The composite operation requires one addition, one subtraction, and two multiplications for a total of four arithmetic operations. If the renderer uses a five-sample-wide Gaussian, and the samples in a sheet cover a 128 x 128 pixel region, the composite-every-sample method does approximately 128 x 128 x 25 composite operations, for a total of 1.6 million arithmetic operations per sheet. The sheet buffer method reduces this number to 128 x 128 x 25 additions and 128 x 128 composite operations for a total of just under .5 million arithmetic operations per sheet, a savings of more than three-to-one over the composite-every-sample method.

The sheet buffer approach addresses the problem of overlapping kernels in the image-space x and y directions. It allows the reconstruction process to add the contribution of all samples in a sheet, whose kernels may overlap, to all the pixels in a sheet before it

passes the sheet buffer onto the visibility process. However, the image-space z direction is still collapsed during the generation of the footprint and there is no additive affect of multiple overlapping kernels in depth, even though many kernels overlap in the z direction. The effect of this approximation can be reduced with the use of multiple footprints per sample in z and multiple sheet buffers in z, as shown in Figure 5.01. In this way, the renderer may maintain multiple sheet buffers and overlapping kernels can contribute their energy to pixels in different sheets before the reconstruction process passes the sheet onto the visibility process. If a reconstruction kernel has a five-sample-wide extent, the renderer needs five sheet buffers and five footprint tables, one for each sheet within a sample's z extent. The renderer composites a sheet buffer when no more samples can affect the sheet buffer. In the five-sample-wide kernel case, a sample can affect its sheet as well as two sheets in front of its sheet and two sheets in back of its sheet. When the renderer finishes processing all the samples in a sheet, no more samples can affect the back-most sheet buffer and the renderer can composite the entire sheet buffer into the accumulation buffer. Once the renderer composites the sheet buffer it can reuse the sheet buffer as the new front-most sheet. This method more closely approximates the initial discrete method, described in section 3.8, without requiring an entire volume buffer. It does, however, require multiple sheet buffers. This method performs the same number of composite operations as the sheet buffer method; however, in the case of a five-sample-wide Gaussian reconstruction kernel, this method does five times as many additions during the reconstruction phase, because it operates on five sheets at a time. In this example, the reconstruction process does 128 x 128 x 25 x 5 additions and 128 x 128 composite operations for a total of 2.1 million arithmetic operations. The present implementation does not use the multiple sheet buffer method, because the single sheet buffer method generates images of sufficient quality and the multiple sheet buffer method requires four times as many arithmetic operations during reconstruction (the most expensive part of the splatting method).

5.4 Footprint Approximations

For an orthographic view, the renderer generates the footprint table once per frame. Ideally, the renderer should analytically integrate the view-transformed kernel along z to generate the footprint table, but this integration may be intractable. Alternatively, the renderer should numerically integrate the view-transformed kernel along z to generate the footprint table. This provides an accurate solution, but requires the footprint generation process to evaluate the kernel multiple times for each footprint table entry as it numerically integrates the kernel. To increase speed, an approximation function was introduced.

In order to limit the extent of the footprint table, the renderer truncates the approximation function. For Gaussian reconstruction kernels, the renderer truncates the kernel at the point where its value falls below 0.004, which is approximately equal to $\frac{1}{255}$ or one quantization level in most computer graphics displays. The result of the truncation is that the volume under the kernel does not equal one, but instead is a little less than one

(.9974), which may causes a slight sample frequency ripple for a constant input. However, there are typically few areas of constant value in volume-rendered images, result intensities are quantized to 256 levels anyways, and the selected truncation value is so small that there is little energy in the truncated tail (less than .3%) that this truncation has no noticeable effect on the output images.

For a truncated spherical Gaussian kernel, which has a bounding sphere and equal mesh spacing in each mesh direction, the approximation function introduces no error. The kernel is assumed to look the same from any view and the renderer can model the result of the integration from any view with a function that is simple to evaluate. If this assumption is true, the renderer generates the footprint table by evaluating the approximation function, rather than integrating the original kernel. In some cases this introduces no error—for example, a two-dimensional Gaussian can accurately model the three-dimensional Gaussian, because one-dimensional integration of a three-dimensional Gaussian yields a two-dimensional Gaussian.

When the mesh spacing is different in each mesh direction, or the scale factor is different in the two image directions, the kernel has an ellipsoid of influence. The renderer calculates the transformation from ellipsoid to sphere and uses this mapping to evaluate the approximation function. When the view direction is along one of the mesh axes, the approximation function still does not introduce any errors for a Gaussian kernel. However, when there are extreme differences in spacing, some angles produce ellipsoids that map poorly to a sphere. The approximation function then differs from the correct solution and the footprint table is not a good representation of the ideal solution. The ideal integration of the kernel would be

$$footprint(x,y) = \int_{-\infty}^{\infty} e^{-\frac{x_V^2+y_V^2+z_V^2}{2\sigma^2}}\, dz,$$

where

$$(x_V, y_V, z_V, 1) = V^{-1} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{1}$$

and V^{-1} is the inverse viewing transformation matrix. The approximation function models the integration as

$$footprint(x,y) = \int_{-\infty}^{\infty} e^{-\frac{x_T^2+y_T^2+z_T^2}{2\sigma^2}}\, dz,$$

where

$$(x_T, y_T, z_T, 1) = T^{-1} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{2}$$

and T^{-1} is the ellipse to circle mapping defined in section 3.7.4.

This phenomenon is illustrated visually in Figure 5.03. If the mesh spacing is twice as large in the x direction as the z direction, the kernel is an ellipsoid. If the view direction is rotated 45° about the y-axis, the footprint generation process should integrate the kernel along rays that are not aligned with any of the ellipsoid's axes. However, the mapping from view-transformed extent to the approximation function extent causes the integration to be modeled as if the rays were axis-aligned. Thus, this rotated ellipsoid is not accurately modeled with the approximation function.

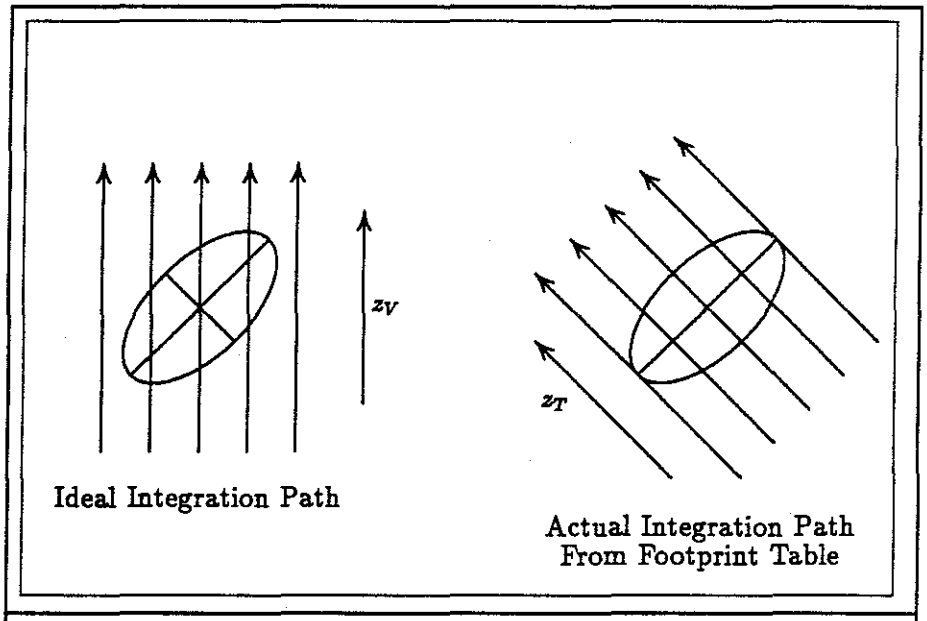


Figure 5.03: Rotated Elliptical Kernels.

The above approximation is a straightforward rendering-time vs. image-quality trade-off. Ideally the renderer should integrate the view-transformed kernel, but this is a compute-intensive operation. Therefore, the renderer uses the approximation function to generate the footprint table quickly.

An error term could possibly be generated by calculating equation (1) - equation (2), but the subjective image quality with this approximation was good enough that we did not calculate or bound the error term.

Chapter 6

Conclusions

6.1 Introduction

This chapter will discuss some conclusions about the splatting method, outline the thesis' contributions, and offer some suggestions for future work.

6.2 Conclusions

This thesis presents a linear-systems-theory based framework for volume rendering. It looks at current volume-rendering methods in this framework to discuss the causes of rendering artifacts. In addition, the thesis presents a new volume-rendering algorithm, splatting, that maps well but not perfectly to the ideal volume-rendering method operating within the framework. The new method can also run in parallel without needing to replicate the input data.

The splatting algorithm presented here differs fundamentally from existing algorithms. It uses feed-forward reconstruction, which touches input samples once and output samples many times. Touching the inputs once is good for distributed systems, since the system does not have to replicate the input data at each processing node or resend the input data many times to various processing nodes. Touching the output many times may be bad for distributed systems, since the output is a resource that may only exist in one place which becomes a memory contention bottleneck.

The renderer can run in parallel in the four ways described in Chapter 4. An advantage of the splatting method is that the processes in the parallel versions are the same as the processes in the non-parallel versions except they operate on a subset of the input data. This makes the parallel process easy to understand and natural to program.

The renderer may make rendering-time vs. image-quality trade-offs at several points in the rendering pipeline.

(1) The renderer may operate on a small version of the input to speed processing thereby sacrificing fine detail in the final images.

(2) The renderer may use poor reconstruction kernels because they have limited spatial extents and reduce the amount of reconstruction computation.

(3) The renderer may use a footprint table that it builds from an an approximation functions instead of the original kernel to reduce the cost of table generation.

(4) The renderer intermixes reconstruction and visibility at different points, depending the memory resources and computation speed of the rendering machine. Separating these process gives better results, but requires more intermediate storage and more computation.

These trade-offs are important in getting a system to run at interactive rates. Researchers can trade how much quality they desire in the final image against how long they are willing to wait for the result.

Splatting is a valid volume reconstruction technique with a sound theoretical basis in signal processing. However, to use the splatting reconstruction techniques for volume rendering, we made some minor concessions. These concessions are:

- We treat shading as a band-limited process, so we can shade first and reduce the amount of computation later.
- We model attenuation/scattering as a discrete compositing process.
- We assume the input is band-limited.
- We do not properly project the z dimension of the signal during visibility.

The renderer generates reasonable quality images. The ideal splatting method may closely model the ideal volume-rendering algorithm with each of the rendering steps adhering closely to the requirements of linear-system theory and each procedure occurring at the proper step in the rendering pipeline. The current implementation of the splatting algorithm make the concessions mentioned above. These concession seem to have only minor effects on the resultant images. The largest effects are seen when the CRIO process is not band-limited and it introduces high frequencies that cause aliasing during the resampling phase. Other noticeable artifacts occur due to the use of the approximation function when the input data set has large differences in the sampling rates along the three mesh directions.

6.3 Contributions

This thesis' major contribution is twofold. First, it presents a theoretical framework for the volume-rendering process. Although the theory is straightforward, it has been largely ignored in previous work. Second it introduces the splatting algorithm, a technique for direct rendering of rectilinear meshes of volume data. It is a naturally parallel algorithm which adheres well (but not perfectly) to the requirements imposed by signal processing theory. The algorithm has several novel features. First, it can render volumes as either clouds or surfaces by changing the shading functions. Second, it can smoothly trade rendering time and image quality by varying the amount of computation during various stages of the rendering pipeline.

6.4 Derivative and Future Work

An important goal for further research is more evenly distributing the reconstruction and visibility process in the parallel implementations. The solution involving a sheet buffer, or a set of sheet buffers, and the accumulation buffer leads to a memory contention bottleneck at the sheet buffer or the accumulation buffer. This contention hampers the rendering speed of parallel implementations, since each computation node must access the common resource at multiple points in the rendering pipeline. Neumann [Neumann 91] has done some interesting work along these lines on the heterogeneous multicomputer Pixel-Planes 5 [Fuchs 89]. He uses the multiple pixel processors to compute the footprint function and the compositing function in parallel for all pixels.

Another area for further research is how volume renderers should display meshes of higher-order elements, such as sets of scalars of vectors. The current shading rules were designed to render scalar fields.

In addition, it is not clear how the straight element-tossing approach will deal with curvilinear and unstructured meshes. The reconstruction footprint can be different for each input sample and using a common footprint is a major way the splatting method reduces the computational requirements of reconstruction. Max, Hanrahan, and Crawfis [Max 90], have used the basic splatting pipeline to render unstructured meshes. Their method models the element as a polyhedron and regenerates the footprint for each data element before splatting that elements contribution to the image.

Another area for further research is modifying the splatting algorithm to handle geometric data intermixed with the volume data. Geometric data is not band-limited and does not easily conform to the independent treatment of the input data. The ray-casting approaches and the data-coercion approaches deal with geometric data trivially, since the ray-casting approaches can trace the geometric data as easily as the volume data, and the data-coercion approaches already generate geometric data as the display primitives. Hanrahan [Hanrahan 91] has an interesting way of modeling a sample's footprint with a collection of semi-transparent polygons. He renders the polygons with conventional high-performance graphics engines. Since these footprints are polygonal, intermixing other polygonal data should be straightforward.

Another area for further research is to bound and characterize the errors introduced by the various approximations made in the implementations described in this thesis.

A final area for further research is process ordering within the splatting pipeline. Splatting can be ordered so that the process more closely adheres to the ideal ordering presented in Chapter 2. It is not clear how this reordering would affect rendering times and memory space requirements. McMillan [McMillan 90] has proposed a method for three-dimensional splatting. The intermediate volume buffer is smaller that the original data set since he takes advantage of maintaining local gradient information to help during reconstruction.

Chapter 7

References

Abram G.D., L.A. Westover, and J.T. Whitted, [1985] "*Efficient Alias-Free Rendering Using Bit-Masks and Look-Up Tables*", Computer Graphics, *vol. 19*, no. 3, July 1985.

Bergman, L., H. Fuchs, E. Grant, and S. Spach, [1986] "*Image Rendering by Adaptive Refinement*", Computer Graphics, *vol. 20*, no. 4, August 1986.

Blinn, J.F., [1982] "*Light Reflection Functions for Simulation for Clouds and Dusty Surfaces*", Computer Graphics, *vol. 16*, no. 3, July 1982.

Brooks Jr., F.P., [1986] "*Interactive Graphics Can Double America's Supercomputers*", 1986 Workshop on Interactive 3D Graphics, October 1986.

Carpenter, L., [1984] "*The A-Buffer, an Antialiased Hidden Surface Method*", Computer Graphics, *vol. 18*, no. 3, July 1984.

Castleman, K.R., [1979] "*Digital Image Processing*", Prentice-Hall Inc., USA.

Catmull, E., and A.R. Smith, [1980] "*3-D Transformations of Images in Scanline Order*", Computer Graphics, *vol. 14*, no. 3, July 1980.

Cline, H.E., W.E. Lorensen, S. Ludke, C.R. Crawford, and B.C. Teeter, [1988] "*Two algorithms for the three-dimensional reconstruction of tomograms*", Medical Physics, *vol. 15*, no. 3, May/June, 1988.

Drebin, R.A., L.C. Carpenter, and P. Hanrahan, [1988] "*Volume Rendering*", Computer Graphics, *vol. 22*, no. 4, August 1988.

Drebin, R.A., L.C. Carpenter [1989] "*Methods And Apparatus For Imaging Volume Data With Shading*", U.S. Patent, no. 4,835,712, May 1989.

Feibush E.A., M.S. Levoy, R.L. Cook, [1980] "*Synthetic Texturing Using Digital Filters*", Computer Graphics, *vol. 14*, no. 3, July 1980.

Frieder, G. , D. Gordon, and R.A. Reynolds, [1985] "*Back-to-Front Display of Voxel Based Objects*", IEEE Computer Graphics and Applications, *vol. 5*, no. 1, January 1985.

Fuchs, H., Z.M. Kedem, and S.P. Uselton, [1977] "*Optimal Surface Reconstruction from Planar Contours*", Communications of the ACM, *vol. 20*, no. 10, October 1977.

Fuchs, H., J. Poulton, J.G. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, [1989] "*Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories*", Computer Graphics, *vol. 23*, no. 3, July 1989.

Gallagher, R. and J. Nagtegaal, [1989] "*An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volumes*", Computer Graphics, *vol. 23*, no. 3, July 1989.

Ganapathy, S., and T.G. Dennehy, [1982] "*A New General Triangulation Method for Planar Contours*", Computer Graphics, *vol. 16*, no. 3, July 1982.

Greenberg, D.P., [1986] "*Scientist Wants a Window into the Database*", Panel on Graphics, Image Processing, and Workstations, October, 1986.

Hanrahan, P., [1990] "*Three-Pass Affine Transforms for Volume Rendering*", Computer Graphics, *vol. 24*, no. 5, November 1990.

Hanrahan, P., [1991] "*Hierarchal Splatting*", Computer Graphics, *vol. 25*, no. 4, July 1991.

Herman, G.T., and H.K. Liu, [1979] "*Three-Dimensional Display of Human Organs from Computed Tomograms*", Computer Graphics and Imaging Processing, January 1979.

Horn, B.K.P., [1981] "*Hill Shading and the Reflectance Map*", Proceedings of the IEEE, *vol. 69*, no. 1, January 1981.

Kajiya, J.T., and B.P. Von Herzen, [1984] "*Ray Tracing Volume Densities*", Computer Graphics, *vol. 18*, no. 3, July 1984.

Kaufman, A., [1991] "*Volume Visualization*", IEEE Computer Society Press, USA.

Lenz, R., B. Gudmundsson, B. Lindskog, and P.E. Danielsson, [1986] "*Display of Density Volumes*", IEEE Computer Graphics and Applications, *vol. 6*, no. 7, July 1986.

Levinthal, C. [1966] "*Molecular model-building by computer*", Scientific American, *vol. 214*, no. 6, July 1966.

Levoy, M.S., and J.T. Whitted, [1985] "*The Use of Points as a Display Primitive*", Technical Report 85-022, University of North Carolina, Chapel Hill, NC, 1985.

Levoy, M.S., [1988] "*Volume Rendering: Display of Surfaces from Volume Data*", IEEE Computer Graphics and Applications, *vol. 8*, no. 3, May 1988.

Levoy, M.S., [1990] "*Efficient Ray Tracing of Volume Data*", ACM Transactions of Graphics, *vol. 9*, no. 3, July 1990.

Lorensen, W.E. and H.E. Cline, [1987] "*Marching Cubes: A High Resolution 3D Surface Construction Algorithm*", Computer Graphics, *vol. 21*, no. 4, July 1987.

Max, N., [1986] "*Light Diffusion through Clouds and Haze*", Computer Vision, Graphics and Image Processing, *vol. 33*, 1986.

Max, N., P. Hanrahan, and R. Crawfis, [1990] "*Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions*", Computer Graphics, *vol. 24*, no. 5, November 1990.

McMillan, L., [1990] Personal Communication, June 1990.

Meagher, D.J., [1984] "*Geometric Modeling Using Octree Encoding*", Computer Graphics and Image Processing, *vol. 19*, no. 2, June 1982.

Mitchell, D.P., and A.N. Netravali, [1988] "*Reconstruction Filters in Computer Graphics*", Computer Graphics, *vol. 22*, no. 4, 1988.

Neumann, U., [1990] "*Accelerating Volume Rendering with a Pixel-Processor Array*", Technical Report 91-000, University of North Carolina, Chapel Hill, NC, 1991.

Oppenheim, A.V. and R.W. Schafer, [1975] "*Digital Signal Processing*", Prentice-Hall Inc., USA

Oppenheim, A.V. and A.S. Willsky, [1983] "*Signals and Systems*", Prentice-Hall Inc., USA

Russel, G., and R. B. Miles, [1987] "*Display and perception of 3-D space filling data*" Applied Optics, *vol. 26*, no. 6, March 1987.

Phong, B.T., [1975] "*Illumination for Computer Generated Images*", Communications of the ACM, *vol. 18*, no. 6, June 1975.

Porter, T., and T. Duff, [1984] "*Compositing Digital Images*" Computer Graphics, *vol. 18*, no. 3, July 1984.

Sabella, P., [1988] "*A Rendering Algorithm for Visualizing 3D Scalar Data*" Computer Graphics, *vol. 22*, no. 4, August 1988.

Schafer, R.W. and L.R. Rabbiner, [1973] "*A Digital Signal Processing Approach to Interpolation*", Proceedings of the IEEE, *vol. 61*, no. 6, June 1973.

Shannon, C. E., [1949] "*Communication in the Presence of Noise*", Proceedings IRE, *vol. 37*, 1949.

Siegel, R. and J.R. Howell, [1972] "*Thermal Radiation and Heat Transfer*", MaGraw-Hill Book Company, USA.

Sun Microsystems Inc, [1991] "*Technical White Paper, Sun VX/MVX Visualization Accelerator*", to be published.

Tuy, H.K., L.T. Tuy, [1984] "*Direct 2-D Display of 3-D Objects*", IEEE Computer Graphics and Applications, *vol. 4*, no. 10, November 1984.

Upson, C., and K. Keller, [1988] "*VBUFFER: Visible Volume Rendering*" Computer Graphics, *vol. 22*, no. 4, August 1988.

Van Hook, T., [1986] Personal Communication, September 1986.

Westover, L.A., [1989] "*Interactive Volume Rendering*" Proceedings of the Chapel Hill Workshop on Volume Visualization, May 1989.

Westover, L.A., [1990] "*Footprint Evaluation for Volume Rendering*" Computer Graphics, *vol. 24*, no. 4, August 1990.

Whitted, J.T., [1983] "*Anti-Aliased Line Drawing Using Brush Extrusion*", Computer Graphics, *vol. 17*, no. 3, July 1983.

Williams, T.R., [1982] "*A Man-Machine Interface for Interpreting Electron Density Maps*", Ph.D. dissertation, University of North Carolina, Chapel Hill, NC, 1982.

Wolberg, G. [1990] "*Digital Image Warping*", IEEE Computer Society Press, USA.

Wright, W.V., [1972] "*An Interactive Computer Graphics System for Molecular Studies*", Ph.D. dissertation, University of North Carolina, Chapel Hill, NC, 1972.