Multiresolution Techniques for Interactive Volume Visualization

Eric C. LaMar

Department of Computer Science

University of California, Davis

November 2, 2000

Multiresolution Techniques for Interactive Volume Visualization

Eric C. LaMar

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Committee in Change

November, 2000

Abstract

In part one of this dissertation, we present a scheme for rendering higher-order isosurfaces. Animation and visualization of rectilinear data require interpolation schemes for smooth image generation. Piecewise trilinear interpolation, the de facto standard for interpolating rectilinear data, usually leads to significant visual artifacts in the resulting imagery. These artifacts reduce the fidelity of the resulting visualization, and may even lead to false interpretations of the data. This part is concerned with the generation of smooth isosurface image sequences, obtained by casting rays through the image plane and computing their intersections with an isosurface. We describe a novel solution to this problem: We replace trilinear interpolation by tricubic interpolation, smoothing out the artifacts in the images; and we simplify the ray-isosurface intersection calculations by rotating and resampling the original rectilinear data onto a second rectilinear grid – a grid with one family of grid planes parallel to the image plane.

In part two of this dissertation, we discuss multiresolution techniques for hardware-accelerated, texture-based visualization of very large data sets. We discuss both volume visualization and cutting-planes techniques and how we have implemented them. In general, these methods use an adaptive scheme that renders the volume in a region-of-interest at a high resolution and the volume away from this region at progressively lower resolutions. The developed algorithm is based on the segmentation of texture space into an octree, where the leaves of the tree define the original data and the internal nodes define lower-resolution approximations. Rendering is done adaptively by selecting high-resolution cells close to a center of attention and low-resolution cells away from this area. We discuss four items particular to multiresolution volume visualization: (a) the special attention that must be paid to the transfer function; (b) the proxy geometry; (c) indexed texture maps to allow for quick changes to the transfer function; and (d) error evaluation in the context of index texture maps. We discuss a method for removing artifacts in multiresolution cutting planes.

Contents

Chapter 1

Introduction

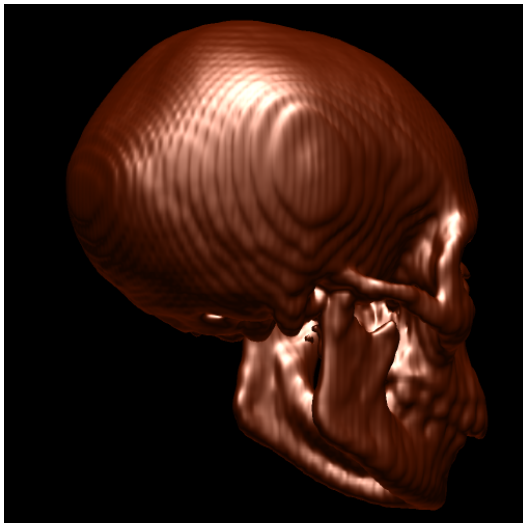
Motivation

Scientific visualization is concerned with the creation of imagery to understand data resulting from scientific measurements or numerical simulations. Understanding is usually best enabled via pictures and animations when (a) data set are large, (b) the data sets contain small-scale details, or (c) the phenomenon represented by a data set are not well understood.

A medical doctor asks a simple question: Does this patient have a tumor? Where is it? A structural engineer asks a simple question: Will this structure fail? Where and Why? A well chosen visualization of the underlying data processes the data "instantaneously" and allows the person to locate the tumor, find the fail points, etc.

Speed and quality of a rendering process significantly affect a person's ability to understand data: the computer must produce images in a "reasonable" amount of time, and the images must be free of distracting or misleading artifacts. To this end, we discuss two contributions to the scientific visualization field. The first contribution is a method for removing surface artifacts typically resulting from traditional trilinear interpolation isosurfacing schemes. The second one is the development of multiresolution technique for volume visualization and cutting plane applications.

Higher-order Isosurfaces


Rendering of trilinear spline.

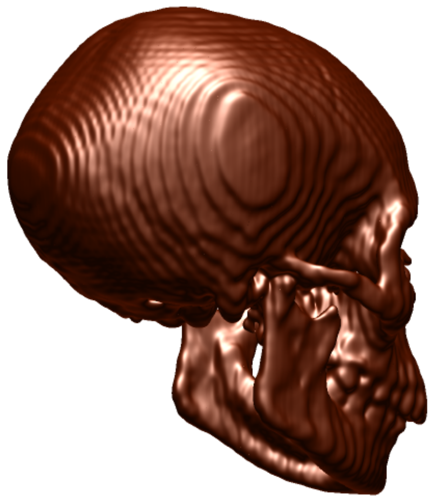

Rendering of tricubic B-spline.

Figure 1.1: A view of a skull data set: the left image is obtained by using a trilinear interpolation scheme with normals calculated by linearly interpolating central differences; the right image is obtained by using a tricubic interpolating B-spline with normals computed analytically from the partial derivatives of the B-spline.

In part one of this dissertation, we present a scheme for rendering higher-order isosurfaces. Animation and visualization of data defined on rectilinear meshes require interpolation schemes for smooth image generation. Piecewise trilinear interpolation, the de facto standard in the visualization community for interpolating rectilinear data, usually leads to significant visual artifacts in the resulting imagery. These artifacts reduce the fidelity of the resulting visualization and may even lead to false interpretations. This part is concerned with the generation of "smoother" isosurface image sequences, obtained by casting rays through the image plane and computing their intersections with an isosurface (*i.e.*, an implicitly defined surface). We describe a novel solution to this problem: we replace trilinear interpolation by tricubic interpolation, smoothing out the artifacts in the images. We simplify the ray-isosurface intersection calculations by rotating and resampling an original rectilinear grid onto a second rectilinear grid – a grid with one family of grid planes parallel to the image plane. Figure 2 shows the advantage of this technique: the left image is obtained by using the traditional trilinear scheme with the normals calculated by linearly interpolating central differences (computed at the grid vertices); the right image is obtained by using a tricubic interpolating B-spline with the normals calculated analytically from the partial derivatives of the B-spline.

Multiresolution Techniques

Multiresolution techniques are used to process data sets that are too large to process directly, due to time or compute resource constraints. These methods allow for a data set to be rendered with a high level of detail in regions of high interest and rendered with a low level of detail in regions of low interest. Also, these methods allow for progressive rendering of a data set, where a coarse approximation is rendered first, with progressively higher resolutions rendered as time and computing resources allow.

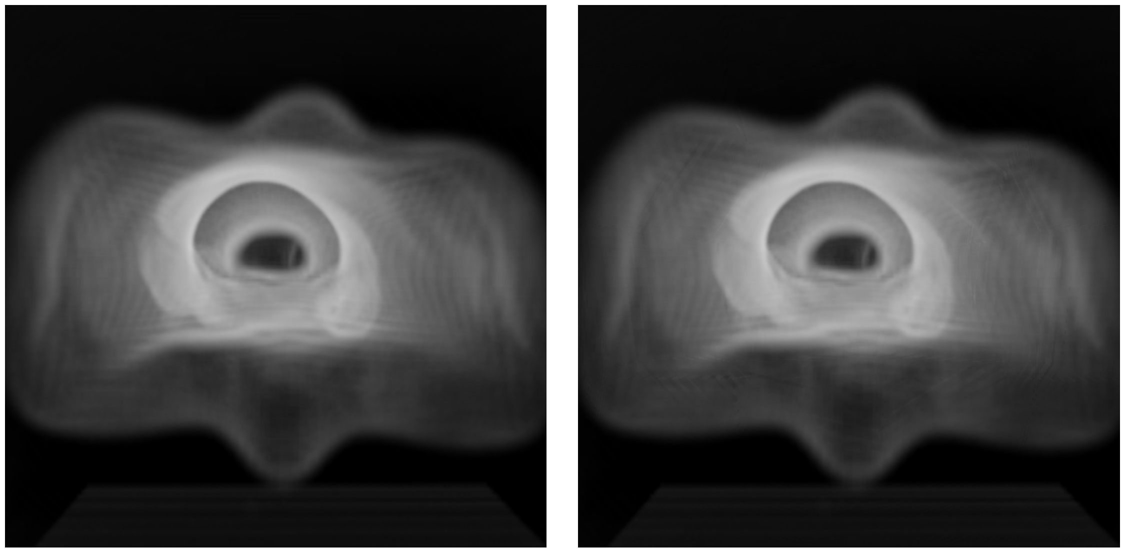
Hardware-accelerated texture-based rendering techniques leverage recent improvements in graphics hardware: instead of processing voxels on a per-voxel basis in software, as ray-tracing and splatting do, the rendering is performed on commodity graphics hardware, where a volume is represented as a large texture. A volume is rendered by applying the texture to a set of proxy-geometries, which can be rendered as standard textured polygons.

In part two of this dissertation, we discuss multiresolution techniques for hardware-accelerated, texture-based visualization of very large scientific data sets. We discuss both volume visualization and cutting-plane techniques and how we have implemented them.

In general, these methods use an adaptive scheme that renders a volume in a region-of-interest (ROI) at a high resolution and the regions further away from this ROI at progressively lower resolutions. The developed algorithm is based on the segmentation of texture space into an octree, where the leaves of the tree define the original data and the internal nodes define lower-resolution approximations. Rendering is done adaptively by selecting high-resolution cells close to a center of attention and low-resolution cells away from this area.

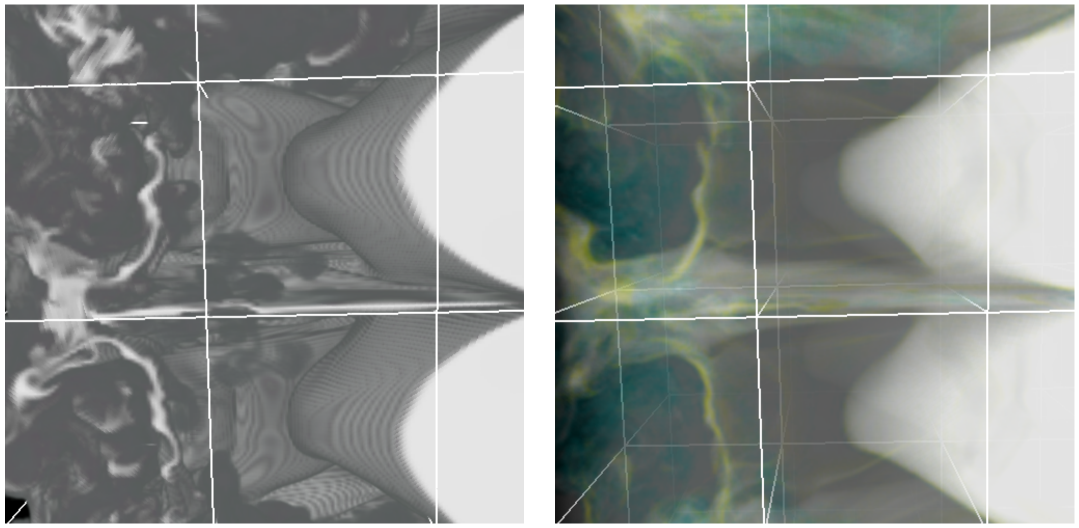
Figure 1.2 shows an equine metacarpal data set ($128^2 \times 108$ voxels). The left image shows an image of the full data set, while the right image shows an image based on a multiresolution approximation. The data and time used for rendering the approximation is roughly half the full-resolution version. Two items are important in multiresolution volume visualization: special attention must be paid to the transfer function such that an approximation-based rendering has the same “optical depth” as one of the original data; and spherical proxy geometries are necessary to regularly sample the data volume.

Interactive manipulation of the transfer function is necessary for exploration of a large volumetric data set, and we show how to use indexed texture maps to allow for quick changes of a transfer function. Figure 1.3 shows a single time step from a Raleigh-Taylor flow simulation ($512^2 \times 768$ voxels). Both images are based on the same approximation, but with different transfer functions.



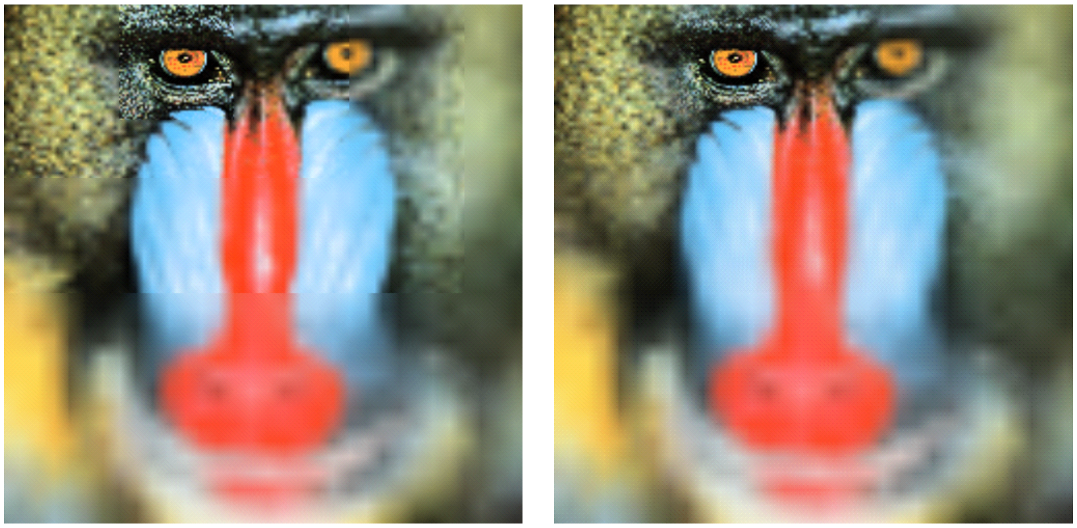
Fixed-resolution rendering (391 tiles, 2.87 seconds). Multiresolution-based rendering (195 tiles, 1.53 seconds).

Figure 1.2: An equine metacarpal data set: the left image shows a rendering of the full resolution version while the right shows a multiresolution approximation. The data closer to the viewer is rendered at high resolution, while the data further away is rendered at lower resolution.



Transfer Function #1 Transfer Function #2

Figure 1.3: Two images rendered from multiresolution approximations of the Raleigh-Taylor flow simulation data set. The images are rendered using two different transfer functions.



Blocky Multiresolution. Smooth Multiresolution.

Figure 1.4: Two images of the Mandrill, the left is rendered without attention to level-of-detail changes. The right is rendered with attention to level-of-detail changes; note how the artifacts in the left image are not present in the right image.

Arbitrarily oriented multiresolution cutting planes are useful for exploring large data sets. While rendering the cutting-planes is very simple, this technique points out the problems with level-of-detail changes. The borders between two different levels of detail can be obvious and distracting, as can be seen in the left image of Figure 1.4. (For simplicity, we show a 2D example). To remove border artifacts, we require that the level of detail not change too quickly across an image, and blend between levels of detail, with only a small penalty of storage. The rendering speed of the current implementation is half that of the non-blending technique, which is explained by the two-pass nature of the algorithm. However, next-generation graphics hardware is likely to allow this technique to be implemented as a one-pass operation.

Assessing the error inherent in a multiresolution approximation is very expensive: a naive approach might evaluate an error function at all original data sites and compare the value of the dependent variables at those sites with those of an approximation. Considering byte data sets (where each dependent variable is represented by an 8-bit integer), the total number of error pairs is quite large (one pair for each data value in the domain of the original data set), but the number of unique error pairs is very small. Each term of the error pair is a byte value, and there are only 256^2 unique pairs. We use this insight to simplify error calculation: we construct a table that stores the frequency of the possible error pairs. To evaluate the error, we accumulate the following: we multiply the error function for a unique error pair by the frequency of that error pair. In the case of the Visible Female CT data set ($512^2 \times 1734$ voxels), this technique allows us to re-compute the error for the entire approximation hierarchy in 1.23 seconds. With a "lazy" evaluation scheme, we can perform a progressive error evaluation, where each tile requires about 0.0028 seconds to for error calculation.

Part I

Higher-order Isosurfaces

Chapter 2

Ray-tracing high-order isosurfaces

Animation and visualization of rectilinear data require interpolation schemes for smooth image generation. Piecewise trilinear interpolation, the de facto standard for interpolating rectilinear data, usually leads to significant visual artifacts in the resulting imagery. These artifacts reduce the confidence in the resulting visualization, and may even lead to false interpretations of the data. This chapter is concerned with the generation of smooth isosurface image sequences, obtained by casting rays through the image plane and computing their intersections with an isosurface. We describe a novel solution to this problem: We replace trilinear interpolation by tricubic interpolation, smoothing out the artifacts in the images; and we simplify the ray-isosurface intersection calculations by rotating and resampling the original rectilinear data in a second rectilinear grid – a grid with one family of grid planes parallel to the image plane. Our solution significantly reduces artifacts in individual images and leads to smooth animations.

2.1 Introduction

With ever increasing speed, data sets are being generated that represent three-dimensional, volumetric information. They arise as a result of complex computational simulations or of empirical data-collection procedures, and present a challenge to current visualization techniques. As these are large problems, the approximation techniques used to render the data produce artifacts that are visible in the resulting images, and especially in the animations. These artifacts frequently present misleading information about the data and may reduce the reliability of the visualization. It is crucial to clearly display all information contained in a data set, and not to simplify the data for the sake of rendering ease.

Scientific data sets are typically scalar- or vector-valued and defined on a grid in two- or three-dimensional space. In many cases, the grid is rectilinear and the data represents either point values associated with the nodes (or vertices) or constant values associated with the voxels (or grid cells). Three methods have been developed to visualize these data sets: the construction of isosurfaces, *i.e.*, surfaces defining the region in space for which a particular field variable is constant [Lev87, LC87, NH91]; volume visualization methods, where rays are cast through the volume, averaging color and opacity values along the ray [Lev87]; and splatting, which is based on projecting the voxel data onto the image plane and considering relative depth to simulate transparency [CLL$^+$88, LH91, WV91].

We describe a method based on a C^1-continuous interpolation scheme that leads to normal-continuous isosurfaces. We construct a cubic tensor-product spline interpolating the function values at the grid points ensuring an overall C^1-continuous representation of the field. The algorithm extends the de facto standard of piecewise-

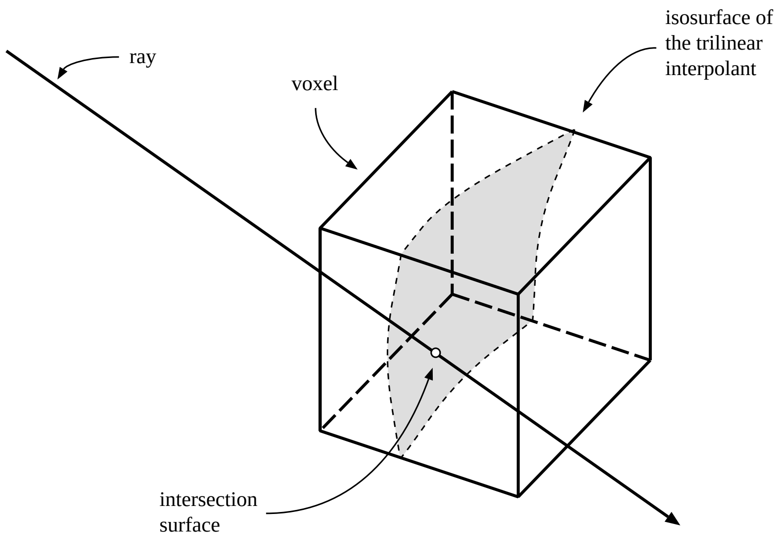


Figure 2.1: Ray-isosurface intersection in a voxel: The shaded area shows the isosurface of the trilinear approximant. The bounding curves of the isosurface on the faces of the voxel are hyperbolic arcs.

trilinear interpolation of the data in a cell to piecewise-tricubic interpolation.

Piecewise trilinear interpolation (Figure 2.1) of rectilinear data interpolates the eight data values at the corners of a cell and yields an overall C^0-continuous approximation of a field. The resulting isosurfaces have normal discontinuities on the shared boundary faces of each pair of neighboring voxels. The resulting normal discontinuities are clearly visible when rendering isosurfaces. Images obtained by ray casting also suffer from the lack of smoothness in the trilinear spline interpolation. The resulting artifacts – “ring patterns” – are particularly visible in animations of a data set.

Piecewise tricubic interpolation utilizes the 64 vertices at the corners of a $3 \times 3 \times 3$ array of cells, and calculates a tricubic approximant to the values within the cell. To determine a point on an isosurface we insert a parametric ray equation

$$\mathbf{r}(t) = \mathbf{p}_0 + t\vec{\mathbf{d}} = \begin{pmatrix} x_0 + tx_d \\ y_0 + ty_d \\ z_0 + tz_d \end{pmatrix}$$

into the individual tricubic polynomial segments approximating the field over each voxel, *i.e.*,

$$f(x, y, z) = \sum_{k=0}^{3} \sum_{j=0}^{3} \sum_{i=0}^{3} c_{i,j,k}\, x^i y^j z^k.$$

Thus, we have to solve the equation

$$\sum_{k=0}^{3} \sum_{j=0}^{3} \sum_{i=0}^{3} (x_0 + tx_d)^i (y_0 + ty_d)^j (z_0 + tz_d)^k = \overline{f}$$

to determine points on the isosurface $f = \overline{f}$. This is an equation of degree nine in t, and thus, intersection of a ray with the isosurface of a tricubic polynomial requires the solution of a ninth-degree polynomial – a time-consuming task.

However, if the direction vector $\vec{\mathbf{d}}$ of the ray $\mathbf{r}$ is parallel to one of the axes of the coordinate system in which $f(x, y, z)$ is defined, then two of the values of x_d, y_d, and z_d must be zero, and the intersection problem leads to a third-degree equation. Thus, we reduce the general ray-isosurface intersection problem by rotating the given rectilinear grid such that one of its coordinate "directions" is parallel to the ray direction, and by resampling the field at the vertices of the "ray-aligned" grid. Thus, each intersection calculation is reduced to root-finding for a third-degree polynomial. We note that since all rays are required to be parallel, this technique is not suitable for perspective projections.

In Section 2.2, we review research related to this problem. In Section 2.3, we review the tricubic spline that we use to approximate a tri-variate scalar field. In Section 2.4, we discuss the rotation and resampling steps that allow us to efficiently render smooth isosurfaces by using axis-aligned rays. The ray-isosurface intersection method is discussed in Section 2.5. In Section 2.6, we discuss ray-isosurface intersection and implementation details of the algorithm. We present results of our approach in Section 2.7 and conclude with an outlook on future work in Section 2.8.

2.2 Related Work

Two basic approaches are used in scientific visualization for the identification, extraction, and rendering of isosurfaces: those that generate an intermediate representation of the isosurfaces for rendering purposes and those that render the data directly.

The marching-cubes algorithm [LC87, NH91] generates an intermediate triangle mesh from a set of trilinear interpolants. Considering each trilinear interpolant independently, the algorithm determines a set of triangles that approximates the isosurface within each voxel. The resulting triangle mesh is displayed using conventional rendering hardware. The number of cases to be considered for generating the mesh within a voxel can be simplified by symmetry and rotation to 14 cases, and the algorithm uses a look-up table to quickly generate the triangular mesh approximating an isosurface. The marching-cubes paradigm has been adapted to tetrahedral meshes by Shekhar *et al.* [SFYC96], where only three cases must be considered, ignoring certain degenerate cases.

Hamann *et al.* describe a method for constructing a “smoother” isosurface by fitting rational-quadratic surface patches to the triangle mesh resulting from the marching-cubes algorithm [HTF97]. Since the contour of the bilinear interpolant on a face of a grid cell is a hyperbolic arc, rational quadratic curves can represent this curve exactly. These boundary curves, together with curves in the interior of a grid cell, can be used to generate the control nets of triangular rational-quadratic patches that approximate the isosurface within the cell. The use of this type of surface patch yields a "smoother" isosurface, but tangent plane discontinuities still exist at the boundaries of each grid cell.

Levoy [Lev87] was one the first to describe the concept of casting rays directly through volumetric data to produce transparent images. His algorithm uses tri-linear spline interpolation along with the spline gradient to assign normal vectors points along the rays for shading purposes. In order to emphasize regions in a three-dimensional data set where the (scalar) function values are in a certain range, one can assign higher weights to data in these regions by increasing their opacity values. Furthermore, one can use the gradient to emphasize boundaries between different "tissue types": High gradients imply such boundaries and can therefore be used to increase the opacity of boundary regions. Levoy describes an efficiency improvement of his algorithm in [Lev90a, Lev90b].

Several volume visualization algorithms that utilize and render the data directly are based on transforming a rectilinear data set so that it coincides with an "image-space cube" [Lac95, Lac96, SS92]. These methods factor the viewing matrix into a three-dimensional shear and a two-dimensional "warp," thus projecting the given data set into image space. The shear-warp algorithm is a resampling scheme that is advantageous for algorithms utilizing SIMD (Single-Instruction, Multiple-Data) architectures [TQ97].

Yagel and Kaufman [YK92] present an algorithm for volume rendering that is based on exploiting coherency between rays in parallel projections. Rays are cast from a base plane, which simplifies the paths through the volume in a serial image-order algorithm, insuring uniform sampling of the volume.

Webber [Web90] describes an algorithm that casts rays through the grid directly. This algorithm constructs a biquadratic isosurface over each grid cell, and intersects the ray with this surface. This biquadratic surface is derived by examining the $3 \times 3 \times 3$ volume of voxels around the target voxel and using one of the 3×3 sides of this voxel array as a base for a biquadratic function. This approximating function is used for ray-isosurface intersection tests. This method allows the derivation of a precise intersection and a precise normal over a voxel.

Our approach combines ray-casting methods with a tricubic spline interpolation scheme for high-quality image generation and animation. To eliminate the root-finding for ninth-degree polynomials, we rotate and resample the original rectilinear data set at the vertices of a grid with one family of grid planes being parallel to the image plane. We cast rays that are normal to the image plane and thus perform ray-intersection calculations for third-degree polynomials. These methods are all independent calculations and can be efficiently implemented on a parallel processing system.

2.3 Tricubic Interpolation

To solve the general ray-isosurface intersection problem, we must compute the intersection of a (parametric) ray and an isosurface of a tricubic approximant. We utilize the cubic tensor-product Catmull-Rom spline scheme [CR74, Far97] to represent the scalar field for all grid cells. We briefly review this spline scheme for the univariate case.

Given a set of scalar values $f_0, f_1, ..., f_n$, the univariate cubic Catmull-Rom spline is a piecewise cubic polynomial function

$$f(x) = \sum_{i=0}^{n} f_i(x) F_i(x),$$

where each $f_i(x)$ is a function of the values f_i, and the $F_i(x)$ denotes a set of blending functions [CR74]. The functions $f_i(x)$ depend on the f_i values but are allowed to vary with x. Catmull and Rom discovered that certain choices of the functions $f_i(x)$ allowed the curve to interpolate the control points f_i.

For our application, it is sufficient to assume that the knots of the spline curve are the integers, and, by utilizing the basis functions $F_i(x) = (x - \lfloor x \rfloor)^i$, where $\lfloor x \rfloor$ denotes the floor of x - the greatest integer less than or equal to x, we can obtain an interpolating spline segment (see the discussing in [CR74] by setting $f_j(x) = 0$ for $j < \lfloor x \rfloor - 1$ and for $j > \lfloor x \rfloor + 2$, and

$$\begin{bmatrix} f_{k-1}(x) \\ f_k(x) \\ f_{k+1}(x) \\ f_{k+2}(x) \end{bmatrix} = M_C \begin{bmatrix} f_{k-1} \\ f_k \\ f_{k+1} \\ f_{k+2} \end{bmatrix},$$

where $k = \lfloor x \rfloor$, and

$$M_C = \frac{1}{2}\begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix}.$$

Thus, one can write $f(x)$ as

$$f(x) = \begin{bmatrix} 1 & x-k & (x-k)^2 & (x-k)^3 \end{bmatrix} M_C \begin{bmatrix} f_{k-1} \\ f_k \\ f_{k+1} \\ f_{k+2} \end{bmatrix},$$

where $k = \lfloor x \rfloor$. For $x = k$ the spline interpolates the point f_k, and the derivative at $x = k$ is $\frac{1}{2}\left(f_{k+1} - f_{k-1}\right)$. Figure 2.2 illustrates a Catmull-Rom spline scheme.

A segment of the Catmull-Rom spline can also be written as

$$f(x) = \sum_{i=0}^{3} f_{\lfloor x \rfloor + i - 1} C_i(x - \lfloor x \rfloor),$$

where

$$\begin{bmatrix} C_0(x) & C_1(x) & C_2(x) & C_3(x) \end{bmatrix} = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} M_C,$$

and the functions $C_j(x)$ denote the Catmull-Rom basis functions.

We can bound the values of a Catmull-Rom spline for a particular knot interval

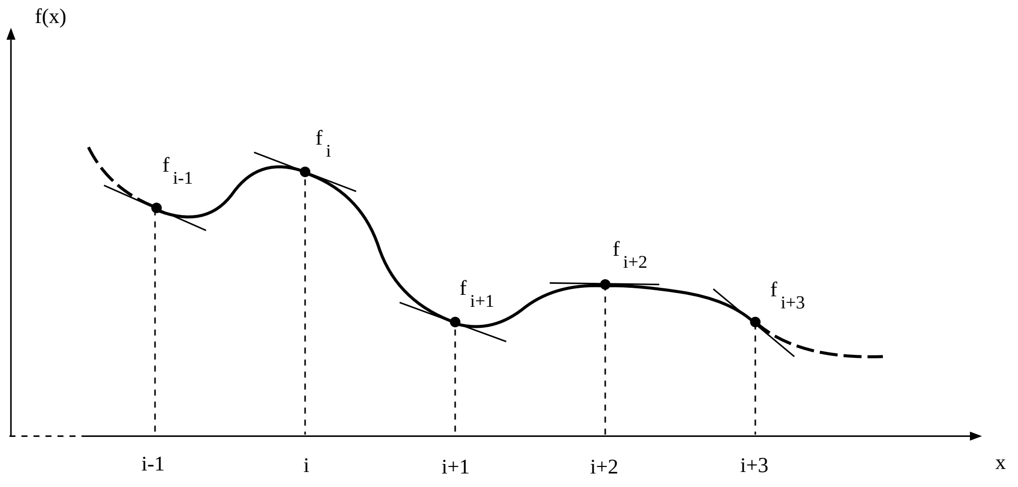


Figure 2.2: A Catmull-Rom spline: The curve interpolates the control points at integer values of the parameter; the slope of the curve at the integer knots is given by the central difference of the two values on either side of each knot.

by converting it to Bernstein-Bézier form. This conversion is given by

$$f(x) = \begin{bmatrix} B_0(x) & B_1(x) & B_2(x) & B_3(x) \end{bmatrix} M_B^{-1} M_C \begin{bmatrix} f_{k-1} \\ f_k \\ f_{k+1} \\ f_{k+2} \end{bmatrix},$$

where

$$M_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & 6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}.$$

The matrix M_B converts from the cubic power basis to the Bernstein basis $\{B_j(x)\}$

[Far97]. The convex-hull property of Bézier curves ensures that each segment of the spline lies between the minimal and maximal values of its four defining Bernstein-Bézier control points. The Bernstein-Bézier "control coefficients" for a polynomial are given by

$$\begin{bmatrix} b_{k-1} \\ b_k \\ b_{k+1} \\ b_{k+2} \end{bmatrix} = M_B^{-1} M_C \begin{bmatrix} f_{k-1} \\ f_k \\ f_{k+1} \\ f_{k+2} \end{bmatrix}.$$

We define I as the smallest interval containing all four values b_{k-1}, b_k, b_{k+1}, and b_{k+2}. The convex-hull property ensures that the curve $f(x)$ is contained within the interval I.

The univariate Catmull-Rom spline can be extended to the trivariate case by constructing a tensor-product spline. In the trivariate case, an individual tricubic segment is given by

$$f(x,y,z) = \sum_{k=0}^{3} \sum_{j=0}^{3} \sum_{i=0}^{3} f_{\lfloor x \rfloor + i-1, \lfloor y \rfloor + j-1, \lfloor z \rfloor + k-1} C_i(x - \lfloor x \rfloor) C_j(y - \lfloor y \rfloor) C_k(z - \lfloor z \rfloor).$$

We use this spline scheme to represent a piecewise tricubic approximation of a scalar field. We must consider a stencil of 64 data values to define the Catmull-Rom spline for a particular grid cell. These 64 data values include the scalar values at the corners of a grid cell C, and the scalar values at the corners of the 26 neighboring cells sharing at least one vertex with C. The 64 values are the elements of the set

$$\{f_{i,j,k} : i \in \{i_0 - 1, i_0 + 2\}, j \in \{j_0 - 1, j_0 + 2\}, k \in \{k_0 - 1, k_0 + 2\}\},$$

where f_{i_0,j_0,k_0} is the value at the vertex of C with minimum i, j, k indices. Again, it is possible to determine the range of function values for a particular tricubic polynomial by converting it to Bernstein-Bézier form and computing the minimal and maximal coefficient.

2.4 Rotation and Resampling

We use the ray tracing paradigm as the mechanism for finding and rendering an isosurface of a trivariate scalar function. Suppose we are to render the isosurface defined by the expression

$$f(x, y, z) = \sum_{k=0}^{3}\sum_{j=0}^{3}\sum_{i=0}^{3} c_{i,j,k}\, x^i y^j z^k = \overline{f} \tag{2.1}$$

for a particular voxel. We must intersect the ray

$$\mathbf{r}(t) = \mathbf{p}_0 + t\vec{\mathbf{d}} = \begin{pmatrix} x_0 + tx_d \\ y_0 + ty_d \\ z_0 + tz_d \end{pmatrix} \tag{2.2}$$

with the isosurface $f = \overline{f}$. To intersect $\mathbf{r}(t)$ with the isosurface, we substitute the x, y and z values of equation (2.2) into equation (2.1) and obtain, in general, a polynomial of degree nine in t. We reduce this problem to solving a third-degree equation by using

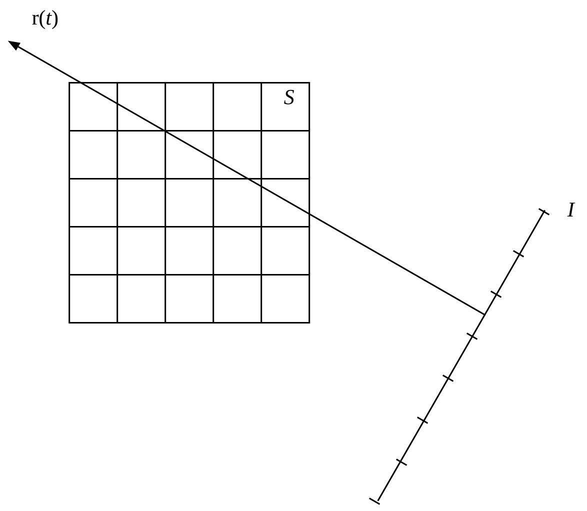


Figure 2.3: Casting rays through the grid S: ray $\mathbf{r}(t)$ passes through a pixel center in the image plane I and intersects the arbitrarily oriented grid cells of S.

rays that are aligned with the z axis, *e.g.*, $x_d = y_d = 0$.

Our algorithm is an image-space algorithm, *i.e.*, we cast rays through each pixel in the image plane I and find intersections with an isosurface. A ray, perpendicular to the image plane I, is cast through the source grid S (see Figure 2.3). We replace the grid S by a second rectilinear grid T, a grid with one family of grid planes parallel to the image plane. Rotating grid S into grid T allows us to perform ray-isosurface computations with axis-aligned rays (see Figure 2.4).

The rotation of S onto T is effectively performed by resampling the scalar field approximation at the vertices of T, *i.e.*, we must approximate new function values for the vertices of each cell in T from the original values at the vertices in S. These resampling problems arise in many visualization applications and they can be resolved in a number of ways [FN80, Lac96, LL94, NFHL91, YK92]. We compare three different approximation techniques for the estimation of function values at the vertices of T: (1) trilinear approximation; (2) tricubic spline approximation using the Catmull-Rom

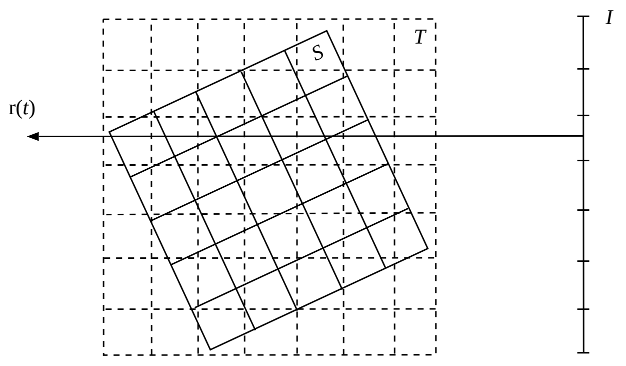


Figure 2.4: Definition of the grid T: One family of grid planes of T is parallel to the image space plane I. Ray $\mathbf{r}(t)$ is axis-aligned with T. After resampling at the vertices of T, ray tracing can be performed with axis-aligned rays.

spline; and (3) Hardy's multiquadric method [Har71, Har90].

Trilinear interpolation is the standard technique used to estimate function values for rectilinear data sets. In this method the function value for a vertex $\mathbf{p}$ of T is approximated by trilinearly interpolating the eight values at the vertices of the cell in S that contains $\mathbf{p}$ (see Figure 2.5). Denoting the coordinates of $\mathbf{p}$ in S by x_p, y_p, and z_p, $\mathbf{p}$'s local parameter values with respect to the cell in S containing $\mathbf{p}$ are

$$\begin{aligned} u &= x_p - \lfloor x_p \rfloor, \\ v &= y_p - \lfloor y_p \rfloor, \text{ and} \\ w &= z_p - \lfloor z_p \rfloor. \end{aligned} \tag{2.3}$$

Thus, the function value $f(\mathbf{p})$ is

$$f(\mathbf{p}) = \sum_{k=0}^{1} \sum_{j=0}^{1} \sum_{i=0}^{1} f_{\lfloor x_p \rfloor + i, \lfloor y_p \rfloor + j, \lfloor z_p \rfloor + k} \, L_i(u) L_j(v) L_k(w),$$

where the linear Lagrange polynomials are $L_0(t) = 1 - t$ and $L_1(t) = t$.

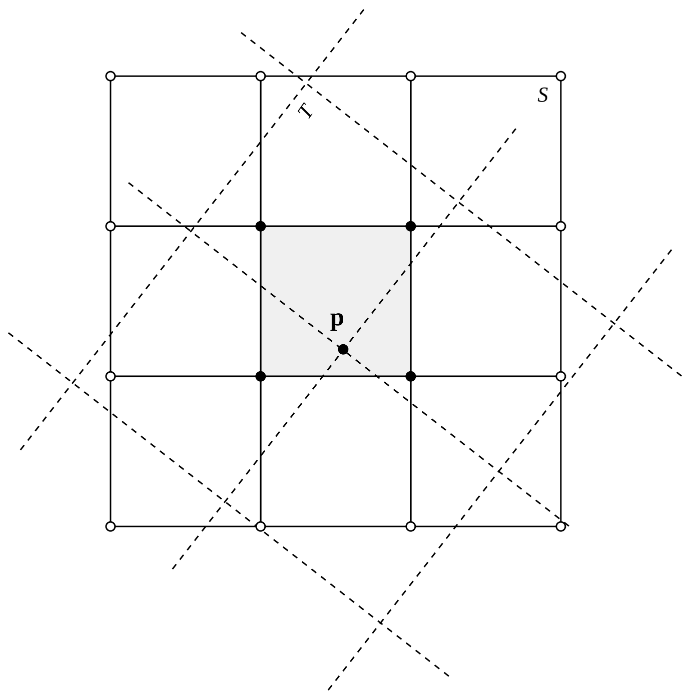


Figure 2.5: A two-dimensional illustration of resampling: Given a vertex **p** of the grid T, bilinear approximation uses the four vertices of the two-dimensional grid cell in S containing **p** to approximate a function value at **p**; bicubic spline approximation and Hardy's method use the 16 vertices of S in the stencil of the cell containing **p** to approximate a function value.

We can extend the trilinear spline approximation scheme to tricubic spline approximation by using the Catmull-Rom spline. In this case, the function value for a vertex $\mathbf{p}$ of T is approximated by considering the 64 vertices of the "stencil cell" in S associated with the cell containing $\mathbf{p}$. If u, v and w denote the local parameter values of $\mathbf{p}$ (equation (2.3)), then the function value $f(\mathbf{p})$ is

$$f(\mathbf{p}) = \sum_{k=0}^{3}\sum_{j=0}^{3}\sum_{i=0}^{3} f_{\lfloor x_p \rfloor+i, \lfloor y_p \rfloor+j, \lfloor z_p \rfloor+k} C_i(u) C_j(v) C_k(w),$$

where $C_i(t)$ denotes the cubic Catmull-Rom blending functions.

Hardy's multiquadric method [FN95, Har71, Har90, NFHL91] is a standard scattered data approximation scheme. It is used in cases where the data sites are randomly distributed and (typically) no connectivity information is known for the data sites. Hardy's approximant smoothly interpolates the values at the data sites and we use it for estimating function values for the vertices of the T grid.

Given a vertex $\mathbf{p}$ in the T grid and $n+1$ vertices $\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_n$ of S, with associated function values $f_0, f_1, ..., f_n$, the coefficients of Hardy's multi-quadric interpolant is defined by the linear system

$$f_j = \sum_{i=0}^{n} \alpha_i \sqrt{(||\mathbf{p}_j - \mathbf{p}_i||)^2 + R^2}, \quad j = 0, ..., n,$$

and the estimate at $\mathbf{p}$ is

$$f(\mathbf{p}) = \sum_{i=0}^{n} \alpha_i \sqrt{(||\mathbf{p} - \mathbf{p}_i||)^2 + R^2},$$

where R^2 is some positive constant. The computation of the coefficients α_i requires the inversion of an $(n+1) \times (n+1)$ matrix.

If $\mathbf{p}$ is a vertex of the grid T, we consider the 64 data sites given by the vertices of the "stencil" in S of the cell containing $\mathbf{p}$ (see Figure 2.5). Assuming that the spacing of the S grid is uniform, the distances $||\mathbf{p}_j - \mathbf{p}_i||$ are the same throughout the grid, and one needs to invert the 64×64 matrix only once. Hardy's method is considerably slower than the tricubic spline approximation due to computation of the square roots. The visual results of the two approximation schemes are similar.

2.5 Ray-Isosurface Intersection Tests

To intersect the parametric ray $\mathbf{r}(t) = (x_0, y_0, z_0 + tz_d)$ with the isosurface

$$f(x,y,z) = \sum_{k=0}^{3}\sum_{j=0}^{3}\sum_{i=0}^{3} c_{i,j,k}\, x^i y^j z^k = 0 \tag{2.4}$$

we substitute the coordinate functions for x, y, and z into equation (2.4) leading to the (normalized) cubic equation

$$t^3 + at^2 + bt + c = 0. \tag{2.5}$$

To find the roots of this equation, we use Cardan's solution [Lit50], which first transforms equation (2.5) by substituting $s = t - \frac{a}{3}$ leading to the polynomial

$$s^3 + ps + q, \tag{2.6}$$

where

$$p = -\frac{a^3}{3} + b \ \text{ and } \ q = 2\left(\frac{a}{3}\right)^3 - \frac{ab}{3} + c.$$

Equation (2.6) has three roots. The characteristics of these roots are determined by the discriminant of the cubic equation

$$D = 4p^3 + 27q^2,$$

which leads to three cases:

- If $D > 0$, then the cubic equation has one real root and two complex roots. The single real root is given by

  $$s_1 = A + B,$$

  where

  $$A = \sqrt[3]{\frac{q}{2} + \sqrt{\frac{D}{27}}} \text{ and } B = \sqrt[3]{\frac{q}{2} - \sqrt{\frac{D}{27}}}.$$

- If $D < 0$, then the equation has three different real roots. The roots are given by

  $$s_1 = k \cos \frac{\alpha}{3},\ s_2 = k \cos \frac{2\pi + \alpha}{3}, \text{ and } s_3 = k \cos \frac{4\pi + \alpha}{3},$$

  where

  $$k = \sqrt{-\frac{4p}{3}} \text{ and } \cos \alpha = \frac{-4q}{k^3}.$$

- If $D = 0$, then the equation has three real roots, of which two are equal. The roots are given by

$$s_1 = \sqrt[3]{\frac{q}{2}} \text{ and } s_2 = -2\sqrt[3]{\frac{q}{2}},$$

where s_1 is the double root.

2.5.1 Accelerating Ray-Isosurface Intersection Tests using Intervals

To accelerate the ray-isosurface intersection computations, we store an interval with each grid cell in T that represents a lower and upper bound of the function values within the cell. The Catmull-Rom interpolants are polynomial splines, and we find the lower and upper bound by determining minimal and maximal spline coefficients of the corresponding Bernstein-Bézier representation (see Section 2.3). If, for a given grid cell, the desired isosurface value is outside the interval associated with this cell, then the cell cannot contain any part of the isosurface.

We also store intervals for each "column of grid cells."[1] Thus, a ray can intersect an isosurface in a grid cell of a particular column only if the column-specific interval contains the particular isovalue. The calculation of lower and upper bounds turns out to be beneficial for "sparse data sets," *i.e.*, data sets containing large homogeneous areas or thin features.

[1]A column of grid cells in T consists of all the cells of T intersected by an axis-aligned ray.

2.6 Implementation

The algorithm requires an initial grid S and an isovalue $\overline{f}$, and it generates an image of the isosurface $f = \overline{f}$. To reduce sampling artifacts, we use a resolution for the grid T that is at least the twice the resolution of the S grid (Nyquist limit) in each direction [Gla95, Jai89]. We estimate function values in T from those in S by using one of the sampling methods discussed in Section 2.4. We compute intervals for each grid cell in T and intervals for each column of grid cells in T.

For each pixel in the image plane, we cast a ray $\mathbf{r}(t)$ into the grid T. If the column-specific interval of the column of cells intersected by $\mathbf{r}(t)$ does not contain the specific isovalue, we skip to the next pixel. Otherwise, for each grid cell intersected by the ray, we examine the intervals for each cell and determine whether the isosurface exists within the cell. If the interval bounds the isovalue, we trace the ray through the grid cell and determine the possible intersections with the isosurface. If the ray intersects the isosurface, then we compute the intersection closest to the image plane and shade the surface using the gradient of the isosurface. Ray tracing terminates when all pixels have been processed.

Normals are calculated by interpolating (either linear or cubic) the central-differences in the target mesh, and are not calculated analytically.

We have implemented our algorithm on several Silicon Graphics Inc. (SGI) workstations, including systems with multiple processors. Our implementation takes advantage of multiple processors by executing all compute-intensive activities in parallel. We have found a near-linear speedup between the implementation on an SGI O2 (single-processor system) and an Origin2000 (16-processor system).

The mapping of source grid vertices to target grid vertices is done in parallel. The mapping of vertices is structured such that the size of the needed memory stride is kept at a minimum. Each processor runs to completion, and waits until the other processors are done. We have experienced that the difference in completion time, considering all processors, is usually less than ten percent.

The ray-isosurface intersection calculations are parallelizable as well. Since rays may pass through different numbers of grid cells and different types of computations may be performed, we use a more sophisticated load-balancing scheme: We use a job queue that consists of scan line indices. A processor reads a scan line index from the job queue and casts rays for all the pixels in the scan line. The time needed to acquire and update the job queue is very small compared to line scanning.

2.7 Results

The first data set for which we present results is a skull data set of CAT (computerized axial tomography) data, of size $64^2 \times 68$ voxels (see Figures 2.6 and 2.7). Figure 2.6 is a comparison of different techniques for resampling and ray-tracing. The matrix of images shows trilinear, Hardy, and tricubic resampling vertically and trilinear and tricubic ray-tracing horizontally. The image in the upper, left-hand corner is the lowest quality image, produced with trilinear interpolation in the resampling phase and trilinear interpolation in the ray-tracing phase. The images generally improve as the basis technique for resampling moves from trilinear to Hardy to tricubic, and as the basis technique for ray-tacing move from trilinear to tricubic. The image in the lower, right-hand corner is the highest quality image, produced with tricubic interpolation in the resampling phase and tricubic interpolation in the ray-tracing phase.

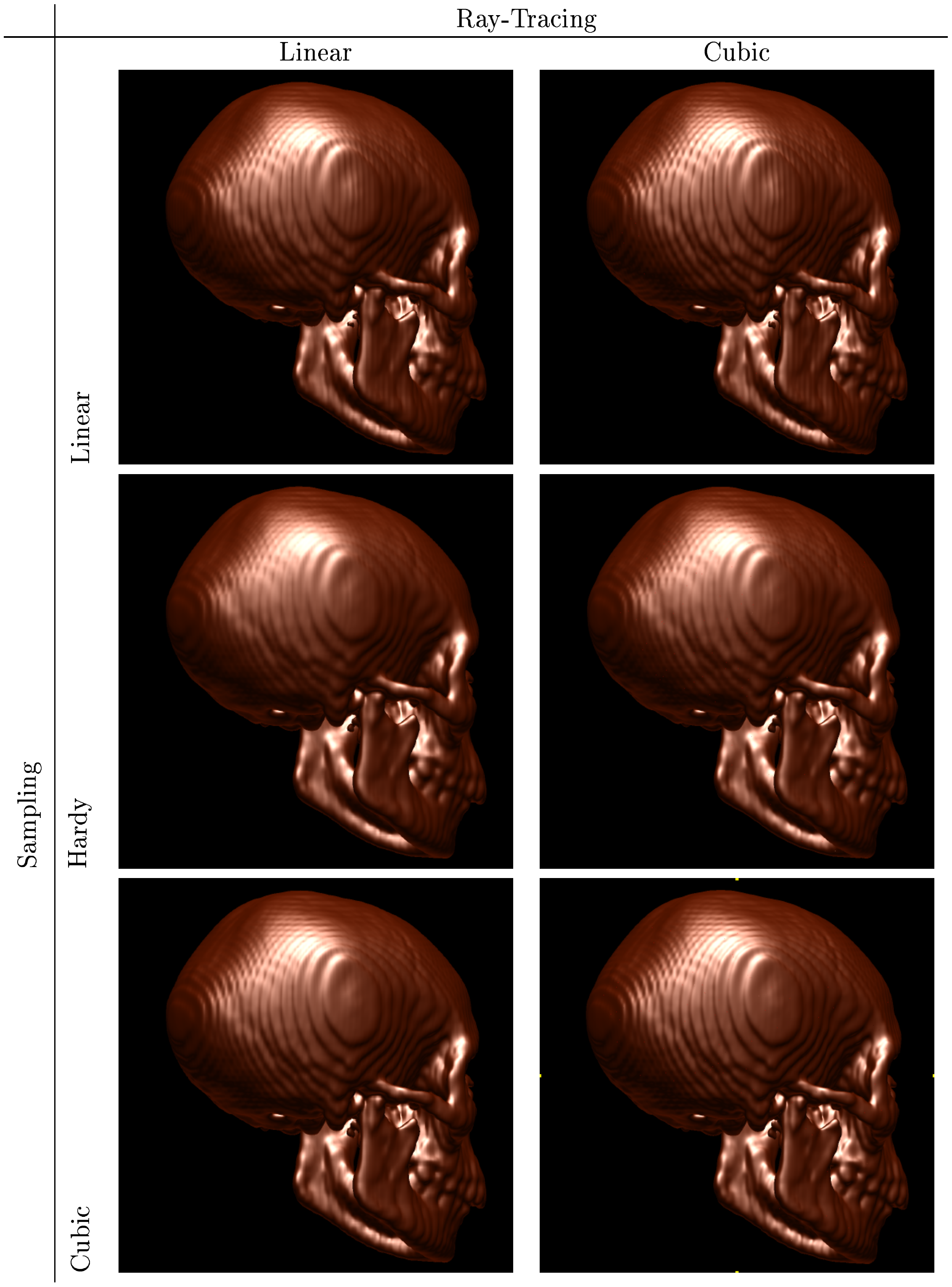


Figure 2.6: Comparison of sample and interpolation methods on the Skull data set.

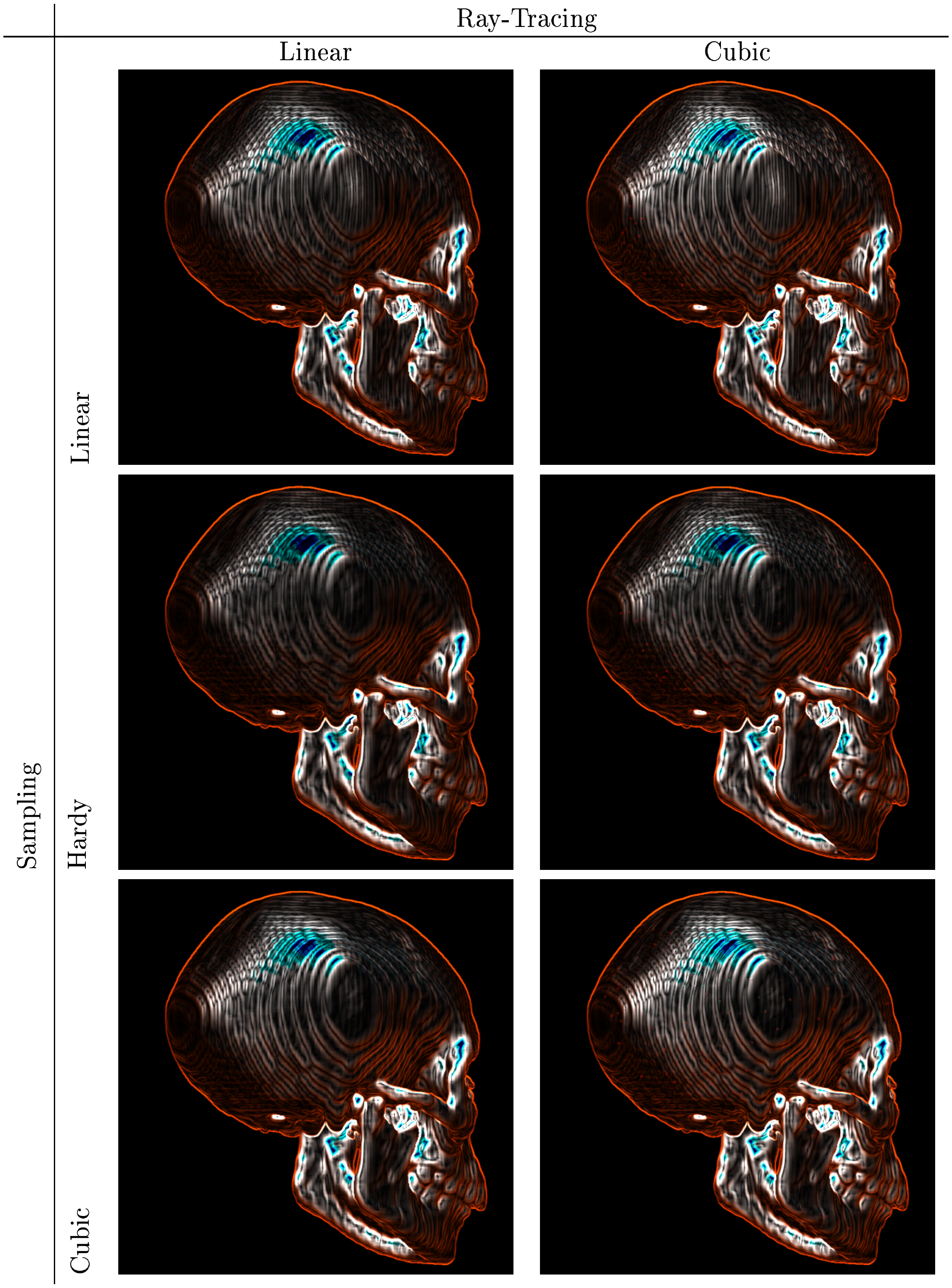


Figure 2.7: Sobol filtered images from Figure 2.6.

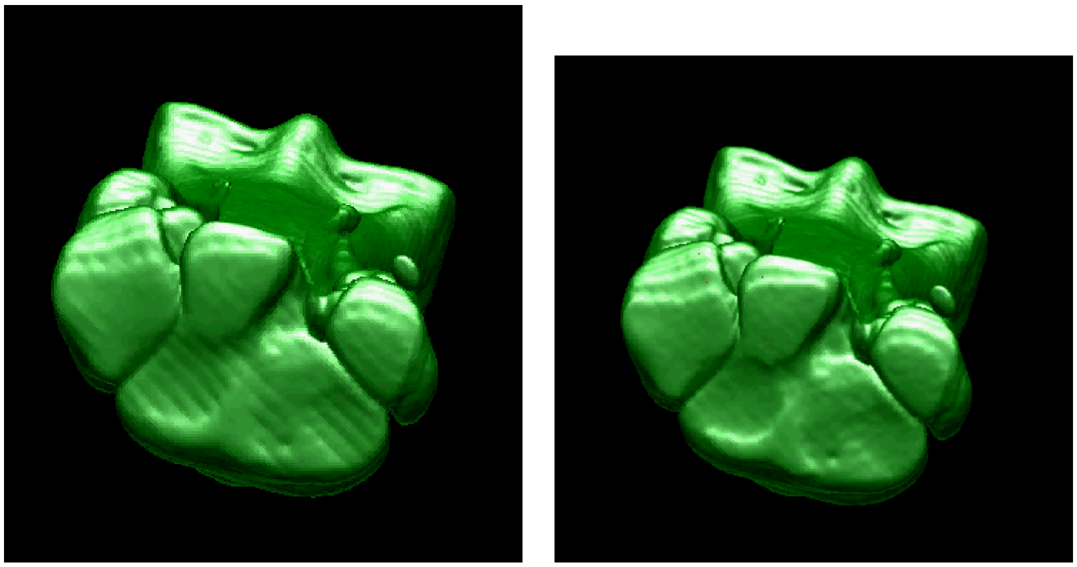


Figure 2.8: Equine metacarpal data set: (a) using tricubic sampling and trilinear interpolation in the ray-tracing step (Note the bands running diagonally through the image); (b) using tricubic sampling and tricubic interpolation in the ray-tracing step. The banding has effectively disappeared.

The vertical bands near the ear of the model (more easily seen in the Sobel filtered images [Jai89], Figure 2.7) appear to move and pulse across the isosurface under animation. In the regions where the angle between the normal to the viewer and light source direction are both acute, one sees alternating bands of light and dark. The Sobel operator illustrates differences in the gradient in the image. We node that the artifacts are reduced in the image with tricubic sample and tricubic ray-tracing.

2.8 Conclusions

We have discussed a new method for rendering smooth isosurfaces of trivariate scalar fields using C^1-continuous spline functions with an efficient ray-tracing scheme. We

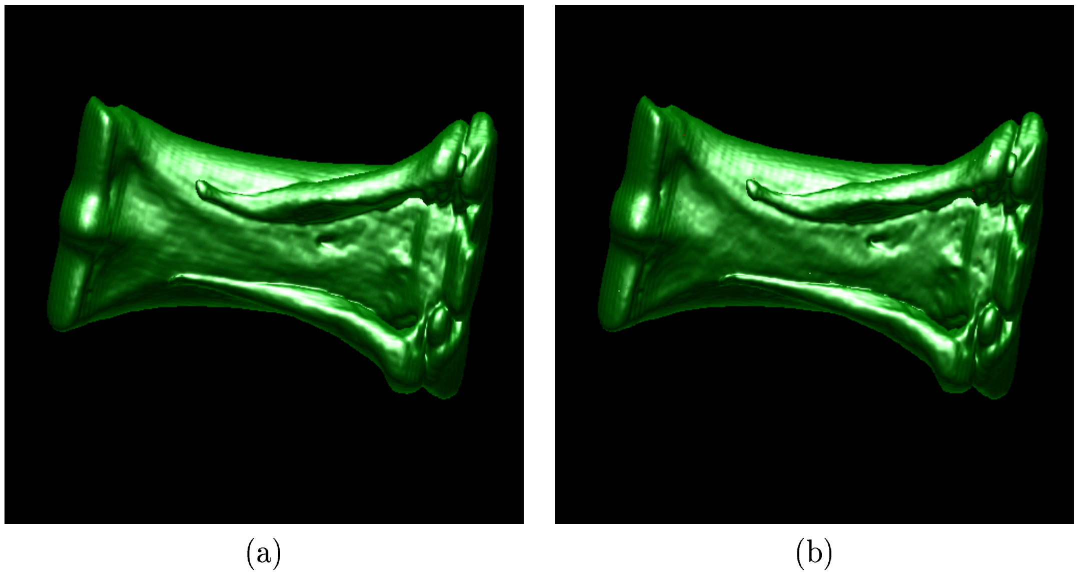


Figure 2.9: A second view of the equine metacarpal data set: (a) using tricubic sampling and trilinear interpolation; (b) using tricubic sampling and tricubic interpolation. Again, the banding virtually disappears.

have shown how a rotation-resampling approach can be used to reduce the complexity of ray-isosurface intersection calculations.

Future work includes developing error metrics to characterize and quantify the error introduced during the rotation/resampling phase. Better methods are needed to identify grid cells containing isosurfaces, more efficient ones than our current linear-search scheme. This method could by applied to other volume visualization techniques, *e.g.*, Levoy's method, and possibly to the rendering of vector fields.

If a data set has the same value over large regions, methods to smoothly approximate these regions by high-degree schemes do not improve smoothness: they require many unnecessary coefficients to be stored. Adaptive methods, such as an octree-based approach, could possibly be used to determine regions that hardly vary, and one could use low-degree schemes in such regions.

Additional work must be done to improve the ray tracing phase of the algorithm: There is a high degree of correlation among ray-isosurface intersections for adjacent pixels, which we do not consider at this point.

Chapter 3

Improving the spline model

3.1 Introduction

In this chapter, we will present a C^2 continuous spline scheme that significantly improves upon the work discussed in Chapter 2. We will also demonstrate a better method for visualizing the artifacts from Chapter 2. Section 3.2 briefly discusses the the central difference approximation; Section 3.3 discusses our method for generation C^2 continuous interpolating splines; Section 3.4 shows tests using 2-d data sets; Section 3.5 discusses 3-d results; and Section 3.6 is the conclusion.

3.2 Improvements to Catmull-Rom Interpolation

Central-differencing is a technique that can be used to approximate the gradient for discrete samples when there is no differentiable function defined. For closely spaced, repeating data values, it can give misleading results. We describe a simple example: construct a one-dimensional spline with control points $P_0, ..., P_n$, where $P_i =$

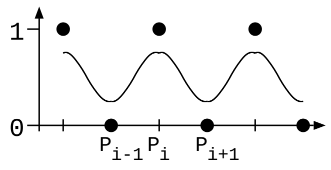


Figure 3.1: An example of the difficulties with central differencing: This curve is a uniform cubic B-spline . The central difference is $D_i = P_{i+1} - P_{i-1}$, which is zero for all D_i. If the gradient is computed from the central differences, it will always be zero.

$(i \mod 2)$, *i.e.*, $P_0 = 0, P_1 = 1, P_2 = 0, \ldots$; see Figure 3.1. For each $i = 1, ..., n-1$, compute the central differences, $D_i = P_{i+1} - P_{i-1}$. One can see that $D_i = 0$ for all i. In Chapter 2, we defined a spline over the data values P_i to compute the curve, and another spline over the central differences D_i to compute the normal to the curve (see section 2.6). For this example, the second spline will always show a gradient of zero. While this is a contrived example, it is demonstrative of the problems using central differences. The result is that the normals, calculated from splining the central differences, can be inaccurate – in fact, smoother than they should be.

When a ray emanating from a pixel intersects the isosurface, the Phong lighting model [And95] is applied, and the pixel is assigned the resulting color. The only way to see surface variation is through the surface highlights. However, the highlights are directly related to the normals, and since the normals are overly smooth, the highlights are similarly overly smooth. The result is that the applied lighting model may not reflect the bumps in the surface.

3.3 Constructing Interpolating Uniform Cubic B-spline Curves

We solve the problems resulting from the use of central differences in two parts. First, we use a C^2 continuous B-spline instead of the C^1 continuous spline described in chapter 2. Second, we take the mathematically correct normal by computing the partial derivative of the trivariate spline. We will not discuss taking the partial derivative, as this is well discussed in the literature.

In the univariate case, given a list $Q = \{Q_0, Q_1, ..., Q_{n-1}\}$ of n scalar values, we wish to derive a list $P = \{P_{-1}, P_0, ..., P_n\}$ of control points, that define a uniform cubic B-spline curve that interpolates all Q_i. The interpolation occurs at the uniformally spaced knots. The reason that there are two more data values in P is that a uniform cubic B-spline of m control points defines $m - 3$ Bézier segments. Since we want to include all $n - 1$ segments of Q, $m - 3 = n - 1$, which implies $m = n + 2$. Note that these two new data values appear at either end of the control polygon, *i.e.*, P_{-1} and P_n.

P_{-1} and P_n must be given some appropriate value. These "end conditions" have been extensively studies in the literature and their full discussion is beyond the scope of this thesis (see Farin [Far97]). For the applications and data sets that we are concerned with, we have found that $P_{-1} = P_n = 0$ works well. This will likely not be the case if the values in Q have large magnitude.

In the trivariate case, we are given a list $Q = \{Q_{0,0,0}, ..., Q_{n-1,n-1,n-1}\}$ of scalar values and must generate a list $P = \{P_{-1,-1,-1}, ..., P_{n,n,n}\}$. P has $(n+2)^3 - n^3 = 6n^2 + 12n + 8$ boundary values that should be given appropriate values.

3.3.1 The Formal Definition of B-spline Curves

The recursive definition of a B-spline curve (from [Joy98]) is as follows:

Given

n control points	$P_0, \ldots, P_{n-1}$,
a degree	k, and
knots	$t_0, \ldots, t_{n+k-1}$,

the implied B-spline curve is defined as follows:

$$
\begin{aligned}
&\text{if} \\
&\qquad t \in [t_i, t_{i+1}) \text{ and } i \in [k-1, n) \\
&\text{then} \\
&\qquad P(t) = P_i^{k-1}(t), \\
&\text{where} \\
&\qquad P_i^j(t) = \begin{cases} (1-\tau_i^j(t))P_{i-1}^{j-1}(t) + \tau_i^j(t)P_i^{j-1}(t) & \text{if } j > 0 \\ P_i & \text{otherwise} \end{cases} \\
&\text{and} \\
&\qquad \tau_i^j(t) = \tfrac{t-t_i}{t_{i+k-j}-t_i}
\end{aligned}
\tag{3.1}
$$

A uniform B-spline has uniformly spaced knots, *i.e.*, $t_i = i$. Figure (3.2) shows a uniform cubic B-spline curve.

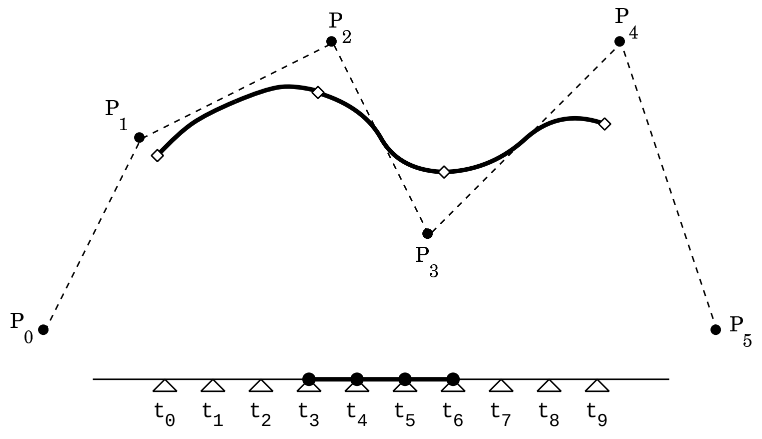


Figure 3.2: A uniform cubic B-spline curve. Here, $k = 4$ and there are 3 segments defined for $t \in [t_3, t_6)$.

A cubic B-spline is defined as follows:

if

$$t \in [i, i+1)$$

then

$$P(t) = P_i^3(t), \tag{3.2}$$

where

$$P_i^j(t) = \begin{cases} (1 - \frac{t-i}{k-j})P_{i-1}^{j-1}(t) + \frac{t-i}{k-j}P_i^{j-1}(t) & \text{if } j > 0 \\ P_i & \text{otherwise} \end{cases}$$

We can expand and simplify this equation for evaluation at a knot i:

$$\begin{aligned}
P_i^3(i) &= (1 - \tfrac{i-i}{4-3})P_{i-1}^2 + \tfrac{i-i}{4-3}P_i^2 = P_{i-1}^2 \\
&= \left(1 - \tfrac{i-(i-1)}{4-2}\right)P_{i-2}^1 + \tfrac{i-(i-1)}{4-2}P_{i-1}^1 = \tfrac{1}{2}P_{i-2}^1 + \tfrac{1}{2}P_{i-1}^1 \\
&= \tfrac{1}{2}\left(\left(1 - \tfrac{i-(i-2)}{4-1}\right)P_{i-3} + \tfrac{i-(i-2)}{4-1}P_{i-2}\right) + \tfrac{1}{2}\left(\left(1 - \tfrac{i-(i-1)}{4-1}\right)P_{i-2} + \tfrac{i-(i-1)}{4-1}P_{i-1}\right) \\
&= \tfrac{1}{2}\left(\tfrac{1}{3}P_{i-3} + \tfrac{2}{3}P_{i-2}\right) + \tfrac{1}{2}\left(\tfrac{2}{3}P_{i-2} + \tfrac{1}{3}P_{i-1}\right) \\
&= \tfrac{1}{6}P_{i-3} + \tfrac{2}{3}P_{i-2} + \tfrac{1}{6}P_{i-1}.
\end{aligned} \tag{3.3}$$

For the following analysis and discussion, we make several definitions and simplifications. First, we define the knots to be $-1 \ldots n$ so that the evaluation of the function $P(i)$ corresponds to the value Q_i, and makes the mapping of our problem definition one-to-one. Second, define the "neighborhood" of a knot i to be the non-zero support of the equation evaluated at i, or the terms P_{i-1}, P_i, and P_{i+1}. Lastly, it follows that the value of a uniform cubic B-spline as a knot i is given by:

$$P(i) = \frac{1}{6}P_{i-1} + \frac{2}{3}P_i + \frac{1}{6}P_{i+1}. \tag{3.4}$$

In the bivariate case, the neighborhood has nine (3×3) terms:

$$\begin{aligned}
P(i,j) = \; & \tfrac{1}{36}P_{i-1,j-1} && +\tfrac{1}{9}P_{i,j-1} && +\tfrac{1}{36}P_{i+1,j-1} \\
& \tfrac{1}{9}P_{i-1,j} && +\tfrac{4}{9}P_{i,j} && +\tfrac{1}{9}P_{i+1,j} \\
& \tfrac{1}{36}P_{i-1,j+1} && +\tfrac{1}{9}P_{i,j+1} && +\tfrac{1}{36}P_{i+1,j+1}
\end{aligned} \tag{3.5}$$

In the trivariate case, the neighborhood has 27 ($3 \times 3 \times 3$) terms:

$$P(i,j,k) = \sum_{I=-1}^{I=1} \sum_{J=-1}^{J=1} \sum_{K=-1}^{K=1} W_I W_J W_K P_{i+I,j+J,k+K}, \quad \text{where} \quad W_{-1} = \tfrac{1}{6}, W_0 = \tfrac{2}{3}, W_{+1} = \tfrac{1}{6}. \tag{3.6}$$

3.3.2 Solution Methods

Given a set of values $Q_0, ..., Q_{n-1}$, several solution methods exist to derive a set of values $P_{-1}, ..., P_n$. We will briefly discuss the methods that we do not use, then continue in the next section with an examination of the method that we do use.

The first method uses an analytical matrix solution [AR87]. Equations (3.4), (3.5), and (3.6) are linear, and can be solved by using standard matrix solution techniques. Each row and column in the matrix refers to a specific entry in the one-, two-, or three-dimensional grid. The matrix for the one-dimensional case can be solved in $O(n)$ time, where n is the number of linear equations, because it can be set up as a tridiagonal matrix. The matrix for the three-dimensional case has 27 non-zero terms per row and cannot be solved in $O(n)$ time. Also, the space cost for representing a sparse matrix is large with respect to the number of non-zero entries in that matrix. The ray-tracer is executing on a parallel machine, so the matrix solver must also execute in parallel, lest it become the bottleneck in the rendering pipeline. We have not had the time to write a fast parallel matrix solver nor have we found one that works under our parallel environment.

The second method is a variation of a scheme from Boehm *et al.* [BFK84]. Instead of solving for the entire three-dimensional grid, it is possible to break it into three steps (two steps for two dimensions). We will describe the two-dimensional version

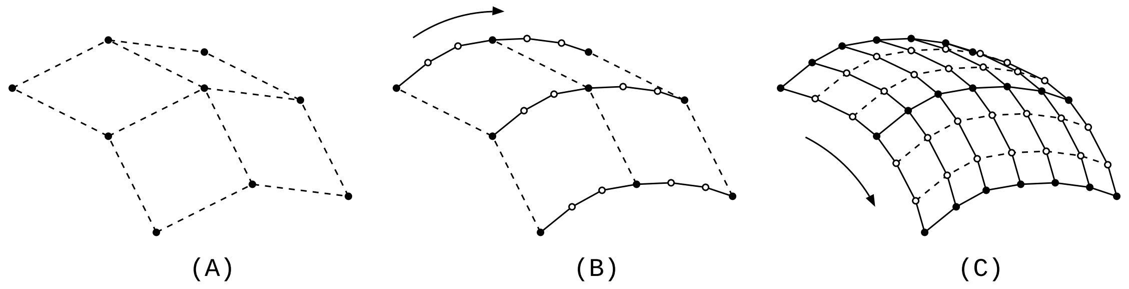


Figure 3.3: The Boehm *et al.*tensor-product interpolating surface scheme.

(Figure 3.3 shows the steps graphically).

One starts with the base grid and for each row, computes a C^2 continuous cubic spline using the matrix method. The matrix is a diagonally dominant, tridiagonal, and can be solved in $O(n)$ time, where n is the number of linear equations. One creates two new points on the curve between each pair of original control points. Then one repeats the process for the $3n-2$ columns. This produces a C^2 continuous surface. In trivariate case, this has a significant disadvantage: when given a grid of size $n \times n \times n$, the resulting grid is of size $(3n-2) \times (3n-2) \times (3n-2)$. Further, since the resampling process requires the final grid to be twice the size of the original grid, given a grid of size n^3, the final grid size will be $216n^3$. Given that the simpler data sets that we wish to visualize are 64^3 to 128^3 voxels at four bytes per voxel, this would require 216MB to 1728MB; this is too high a cost.

The last method is mentioned briefly by Farin [Far97]. This is an iterative scheme, where the control points $P_{-1}, P_0, ..., P_n$ are adjusted until the curve's approximation of $Q_0, Q_1, ..., Q_{n-1}$ is below some predefined threshold. This algorithm is simple to implement and lends itself towards a parallel implementation, so it is the method we use and will discuss in the next section.

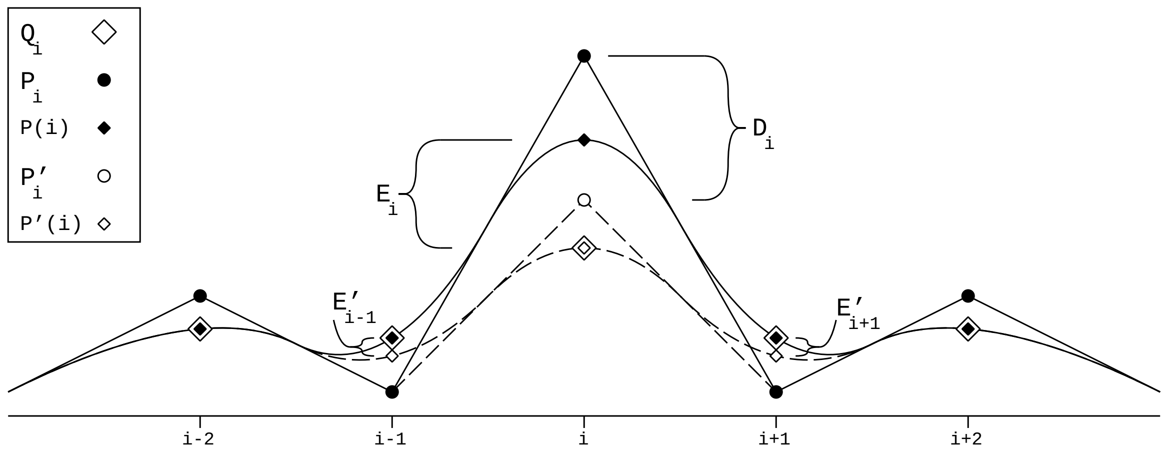


Figure 3.4: The curve in the neighborhood of knot i. Initially, the (solid) curve does not interpolate Q_i. $E_i \neq 0$ and $E_{i\pm1} = 0$. P_i is moved to P'_i, and the (dashed) curve interpolates Q_i: $E'_i = 0$ and $E'_{i\pm1} \neq 0$, but $E'_{i\pm1} \ll E_i$.

3.3.3 Computing the Control Points

We now discuss an iterative approach to computing the control points for an interpolating cubic B-spline curve. We will describe a one-dimensional algorithm, but the extension to two and higher dimensions is straight-forward.

Let the control points of the uniform cubic B-spline curve be $P_{-1,\dots,n}$, the curve be $P(i)$, as defined in equation 3.4, and the set of points be $Q_{0,\dots,n-1}$ where we assume that $P(i) = Q_i$.

The first step is to give P_i an initial values: assign $P_i = Q_i$ for $i \in [0, n-1]$; assign P_{-1} and P_n some appropriate value (as defined earlier). The second step is to move control points so the final curve interpolates the data points (see Figure 3.4). For all $i \in [0, n-1]$, we find a new control point P_i' that will cause the curve $P(i)$ to interpolate Q_i. We solve for P_i' using equation (3.4):

$$P_i' = \frac{3}{2}\left(Q_i - \frac{1}{6}P_{i-1} - \frac{1}{6}P_{i+1}\right). \tag{3.7}$$

The error in the local approximation is the differences between the point to be interpolated, Q_i, and the curve at $P(i)$. This value is

$$\begin{aligned} E_i &= P(i) - Q_i \\ &= \tfrac{1}{6}P_{i-1} + \tfrac{2}{3}P_i + \tfrac{1}{6}P_{i+1} - Q_i, \end{aligned} \tag{3.8}$$

which will be zero if the curve passes through Q_i.

The distance that the control point P_i is moved to $P_i{}'$ to reduce the error to zero, *i.e.*, properly interpolate Q_i, is given as

$$\begin{aligned} D_i &= P_i{}' - P_i \\ &= \tfrac{3}{2}\left(Q_i - \tfrac{1}{6}P_{i-1} - \tfrac{1}{6}P_{i+1}\right) - P_i \\ &= \tfrac{3}{2}\left(Q_i - \tfrac{1}{6}P_{i-1} - \tfrac{1}{6}P_{i+1} - \tfrac{2}{3}P_i\right) \\ &= -\tfrac{3}{2}\left(\tfrac{1}{6}P_{i-1} + \tfrac{2}{3}P_i + \tfrac{1}{6}P_{i+1} - Q_i\right) \\ &= -\tfrac{3}{2}\left(P(i) - Q_i\right) \\ &= -\tfrac{3}{2}E_i. \end{aligned} \tag{3.9}$$

The new error in the local approximation is now zero:

$$E'_i = P'(i) - Q_i = 0. \tag{3.10}$$

The error in the neighboring approximations are:

$$\begin{aligned} E_{i+1} &= P(i+1) - Q_{i+1} \\ &= \tfrac{1}{6}P_i + \tfrac{2}{3}P_{i+1} + \tfrac{1}{6}P_{i+2} - Q_{i+1}. \end{aligned} \tag{3.11}$$

After moving P_i, the new error in the neighboring approximations is

$$\begin{aligned} E'_{i+1} &= P'(i+1) - Q_{i+1} \\ &= \tfrac{1}{6}P_i' + \tfrac{2}{3}P_{i+1} + \tfrac{1}{6}P_{i+2} - Q_{i+1} \\ &= \tfrac{1}{6}(P_i + D_i) + \tfrac{2}{3}P_{i+1} + \tfrac{1}{6}P_{i+2} - Q_{i+1} \\ &= \tfrac{1}{6}P_i + \tfrac{1}{6}D_i + \tfrac{2}{3}P_{i+1} + \tfrac{1}{6}P_{i+2} - Q_{i+1} \\ &= \tfrac{1}{6}D_i + \left(\tfrac{1}{6}P_i + \tfrac{2}{3}P_{i+1} + \tfrac{1}{6}P_{i+2} - Q_{i+1}\right) \\ &= -\tfrac{1}{4}E_i + E_{i+1}. \end{aligned} \tag{3.12}$$

By moving a control point P_i to P_i', the local approximation error decreases by E_i, but the two neighboring approximation errors increase by $\frac{1}{4}E_i$. After one iteration, the overall error will decrease by $\sum_i E_i$ and increase by $\sum_i 2 \times \frac{1}{4}E_i$, or a total decrease of $\sum_i \frac{1}{2}E_i$. Hence, the error will decrease by half each iteration, which means that the process will converge.

The algorithm is thus:

```
Input:
    a list of data values $Q_{0,\ldots,n-1}$
    a number of iterations $N$
Output:
    a list of control values $P_{-1,\ldots,n}$
Temporary:
    a temporary list of data values $T_{0,\ldots,n-1}$
Algorithm:
    copy $Q_{0,\ldots,n-1}$ to $P_{0,\ldots,n-1}$
    assign $P_{-1}$ and $P_n$ some appropriate value
```

```
for j = 1 to N
    for i = 0 to n-1
        $T_i = \frac{3}{2}\left(Q_i - \frac{1}{6}P_{i-1} - \frac{1}{6}P_{i+1}\right)$
    copy $T_{0,\ldots,n-1}$ to $P_{0,\ldots,n-1}$
```

In the bi- and trivariate cases, T and P are the same array. We do this to save on memory. This causes some artifacts that can be seen after the first and second iterations, but disappears after the third.

For the bi- and trivariate cases, we use a iteration limit of $N = 4$. To select this parameter, we created a series of pictures of the skull, with iteration limits of one to ten. The difference of two images is the absolute value of the difference of each color channel (red, green, blue) for each pixel. We found that the differences at $N = 4$ to be very small, with about 40 visible pixels in the differenced image (of a 500 x 500 pixel image). At this time, no formal error analysis has been performed. Clearly, there is a trade-off here: image quality for time. If the error is too high, a larger N can be used. Tables (3.1) and (3.2) detail a run of ten iterations on a curve of seven points. Figure 3.5 shows the final results of the run graphically.

3.3.4 The "Overshooting" Artifact and Error

One of the artifacts of most interpolation schemes is that an interpolating curve can "overshoot". Figure 3.6 shows Catmull-Rom and interpolating cubic B-spline curves. The overshooting in the Catmull-Rom spline is localized due to its local slope estimates. In contrast, interpolation based on uniform cubic B-spline curves can lead to global overshooting artifacts. These are well known artifacts of these methods.

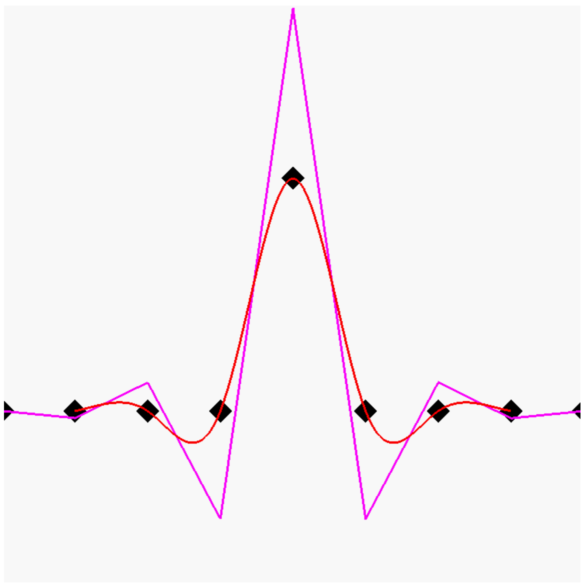


Figure 3.5: The interpolating cubic B-spline curve: The uniform cubic B-spline is red and it's control polygon is purple. The center seven diamonds, from left to right, are Q_0 through Q_6. The vertices of the control polygon, from left to right, are P_{-1} through P_7.

		Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	
		0.000	0.000	0.000	1.000	0.000	0.000	0.000	
i	P_{-1}	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000
1	0.0000	0.0000	0.0000	−0.2500	1.5000	−0.2500	0.0000	0.0000	0.0000
2	0.0000	0.0000	0.0625	−0.3750	1.6250	−0.3750	0.0625	0.0000	0.0000
3	0.0000	−0.0156	0.0937	−0.4218	1.6875	−0.4218	0.0937	−0.0156	0.0000
4	0.0000	−0.0234	0.1093	−0.4453	1.7109	−0.4453	0.1093	−0.0234	0.0000
5	0.0000	−0.0273	0.1171	−0.4550	1.7226	−0.4550	0.1171	−0.0273	0.0000
6	0.0000	−0.0292	0.1206	−0.4599	1.7275	−0.4599	0.1206	−0.0292	0.0000
7	0.0000	−0.0301	0.1223	−0.4620	1.7299	−0.4620	0.1223	−0.0301	0.0000
8	0.0000	−0.0305	0.1230	−0.4630	1.7310	−0.4630	0.1230	−0.0305	0.0000
9	0.0000	−0.0307	0.1234	−0.4635	1.7315	−0.4635	0.1234	−0.0307	0.0000
10	0.0000	−0.0308	0.1235	−0.4637	1.7317	−0.4637	0.1235	−0.0308	0.0000

Table 3.1: The iterative interpolating cubic B-spline method: this table shows how the control points are moved to interpolate the initial points $(Q_{-1}, ..., Q_7)$. Values away from the center (P_3) alternate between negative and positive.

i	L_1	L_2	L_∞
0	0.666667	0.166667	0.333333
1	0.333333	0.024306	0.083333
2	0.166667	0.004774	0.041667
3	0.078125	0.001004	0.015625
4	0.036458	0.000214	0.007812
5	0.016927	0.000046	0.003255
6	0.007813	0.000010	0.001628
7	0.003621	0.000002	0.000692
8	0.001668	0.000000	0.000346
9	0.000773	0.000000	0.000148
10	0.000356	0.000000	0.000074

Table 3.2: This table details the error norms of $Q_i - P(i)$ from table 3.1. Note that the L_1 norm is decreasing at a rate slightly larger than discussed in section 3.3.3. This can be explained by noting that E_{-1} and E_7 "absorb" the error from E_0 and E_6.

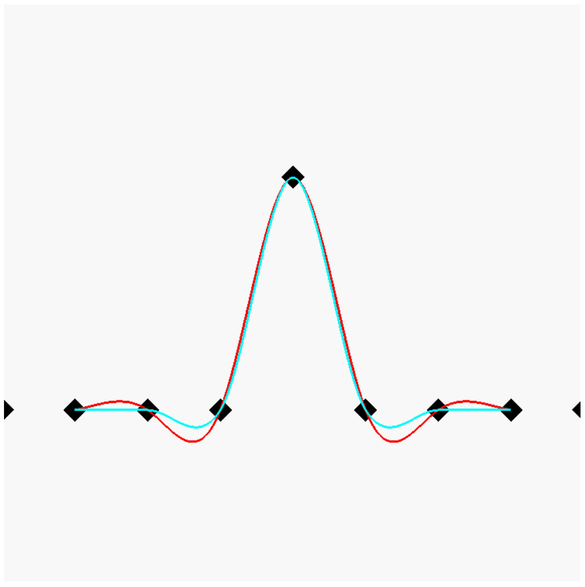


Figure 3.6: A comparison of linear splines: the blue curve is a C^1 Catmull-Rom spline and the red line is a C^2 interpolating cubic B-spline .

L	U	R G B
-10	0.100	1 1 1
0.100	0.200	0 1 0
0.200	0.300	0 0 1
0.300	0.400	1 0 0
0.400	0.500	0 1 0
0.500	0.600	0 0 1
0.600	0.700	1 0 0
0.700	0.800	0 1 0
0.800	0.900	0 0 1
0.900	1.000	1 0 0
1.000	10	1 1 1

Table 3.3: The isoslice look-up table. A value f is assigned a color C_i if $L_i <= f < U_i$. The first and last entries are to deal with the under and over-shoot features of interpolating curves.

3.4 Results with Two-dimensional Slices

This section discusses the results of a bivariate implementation. One of the issues with the prior method is the fact that it is not possible to visualize the surface directly – we could only see the highlights from surface features. Our solution is to extract a two-dimensional slice from a trivariate data set and render the slice. We found this method extremely useful for visualizing isocontours and the differences between the Catmull-Rom and interpolating cubic B-spline methods.

The images are generated first by defining a two-dimensional patch over the data set (either Catmull-Rom or interpolating uniform cubic B-spline). Then, for each pixel in the image, evaluate the function, obtaining a value f. Lastly, the color is assigned by checking in a look-up table (see tables 3.3, 3.4a, and 3.4b). The pixel is assigned color C_i if f is between L_i and U_i. The number of solution iterations for the interpolating uniform cubic B-spline is four.

L	U	R G B
0.00078125	0.0015625	0 1 1
0.0015625	0.003125	1 0 1
0.003125	0.00625	1 1 0
0.00625	0.0125	0 1 1
0.0125	0.025	1 0 1
0.025	0.050	1 1 0
0.050	0.100	0 1 1

(a)

L	U	R G B
0.900	0.950	0 1 1
0.950	0.975	1 0 1
0.975	0.9875	1 1 0
0.9875	0.99375	0 1 1
0.99375	0.996875	1 0 1
0.996875	0.9984375	1 1 0
0.9984375	1.000	0 1 1

(b)

Table 3.4: The isoslice look-up tables: (a) for small values and (b) for large values.

The contour is the edge between two colors. This allows us to show large families of isosurfaces in one image. This is very important because the nature of the artifacts of the different sampling algorithms change for isovalues from the very small (near zero) to very large (near one).

We have not done any analytical analysis on the contours and have relied on "eyeball" measurements. However, the pictures are fairly compelling without this analysis.

For the following series of images, we use the following conventions. "single" vs. "double" sampling refers to the sampling frequency, since the data in the source grid is resampled onto the target grid. A single-sampled image is one that is sampled onto a grid of the same spatial resolution. A double-sampled image is on that is one that is sampled onto a grid of double the spatial resolution. We note that the spatial extent of the target grid is larger than the source grid, and we do this such that the entirety of the source grid appears in the target grid. Also, iCS is an abbreviation for interpolating uniform cubic B-spline .

We note that, in contrast to the prior chapter, where different methods were used in resampling and rendering steps, when generating an image, we use the same

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

Table 3.5: The slab data set.

method for resampling and rendering, i.e., Catmull-Rom means that Catmull-Rom was for used for both sampling and rendering, and iCS means that iCS was used for both sampling and rendering.

The base data set is shown in Table 3.5. Images 3.7a, 3.7c, 3.8a, and 3.8c are created using the Catmull-Rom formulation. Images 3.7b, 3.7d, 3.8b, and 3.8d are created using the interpolating cubic B-spline (iCS) formulation.

The first pair of images, 3.7a and 3.7b, show the data sets, unrotated and sampled at the same rate (single sampling) as the original data set (11 x 11). There are clearly differences between images which are inherent to the formulation. These are the references images.

The second pair of images, 3.7c and 3.7d, show the data sets rotated by 0.2 radians ($\tilde{1}1.4$ degrees) and single sampled. These images look very little like unrotated versions. At this point, both are "equally" bad.

Images 3.8a and 3.8b comprise the third pair, which show the data sets rotated by 0.2 radians but are double sampled. One notices how "wavy" image 3.8a is and how poorly it reconstructs 3.7b. However, image 3.8c is very close to the unrotated version, image 3.7b.

The fifth and final images, 3.8c and 3.8d, show the data set rotated by 0.2 radians, double sampled, and the isovalues from tables 3.4a and 3.4b added to the color look-up

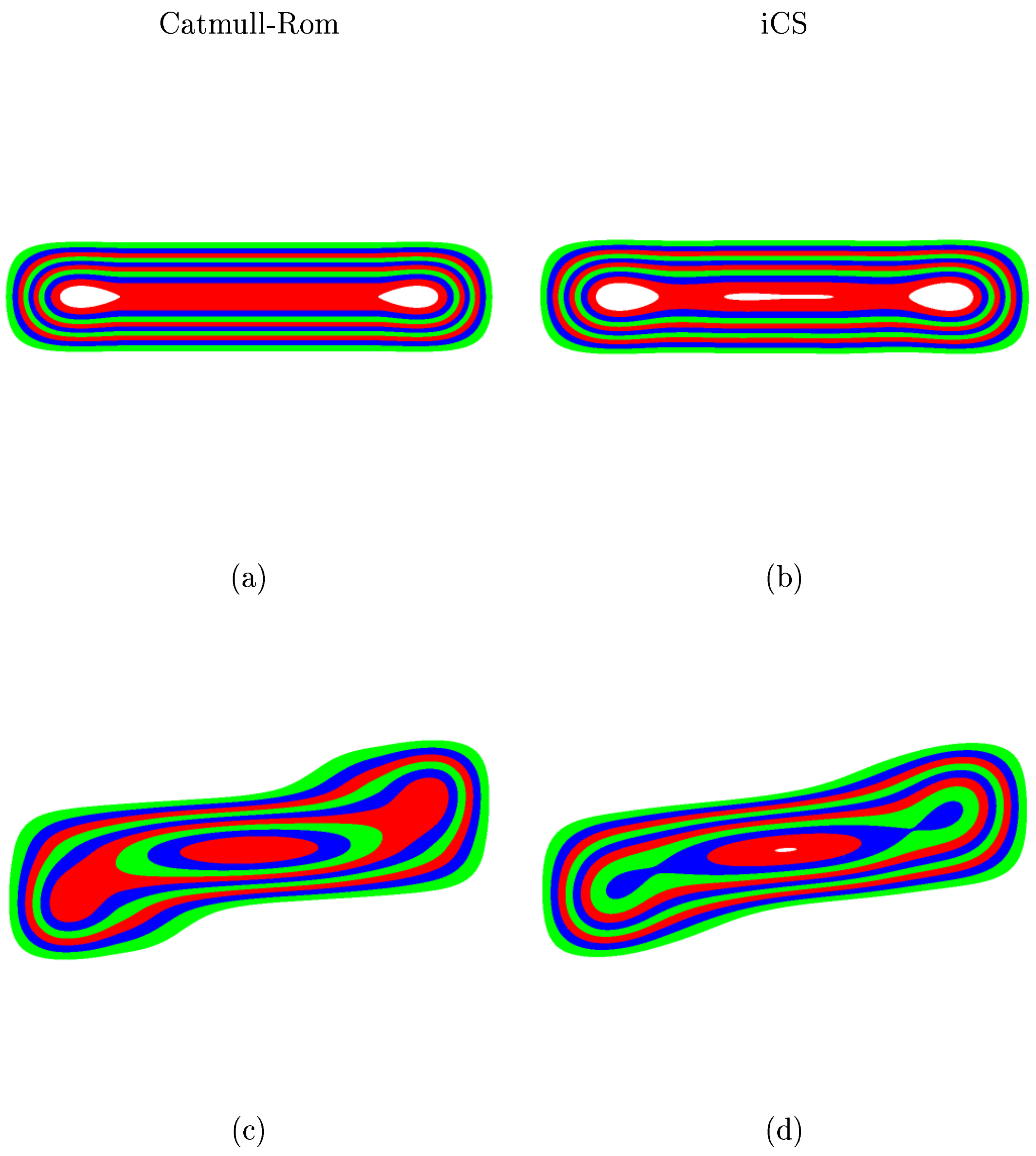


Figure 3.7: Images (a) and (c) are Catmull-Rom and (b) and (d) are iCS. Images (a) and (b) are the slab, unrotated, single sampled. Images (c) and (d) are the slab, rotated by 0.2 radians, and single sampled.

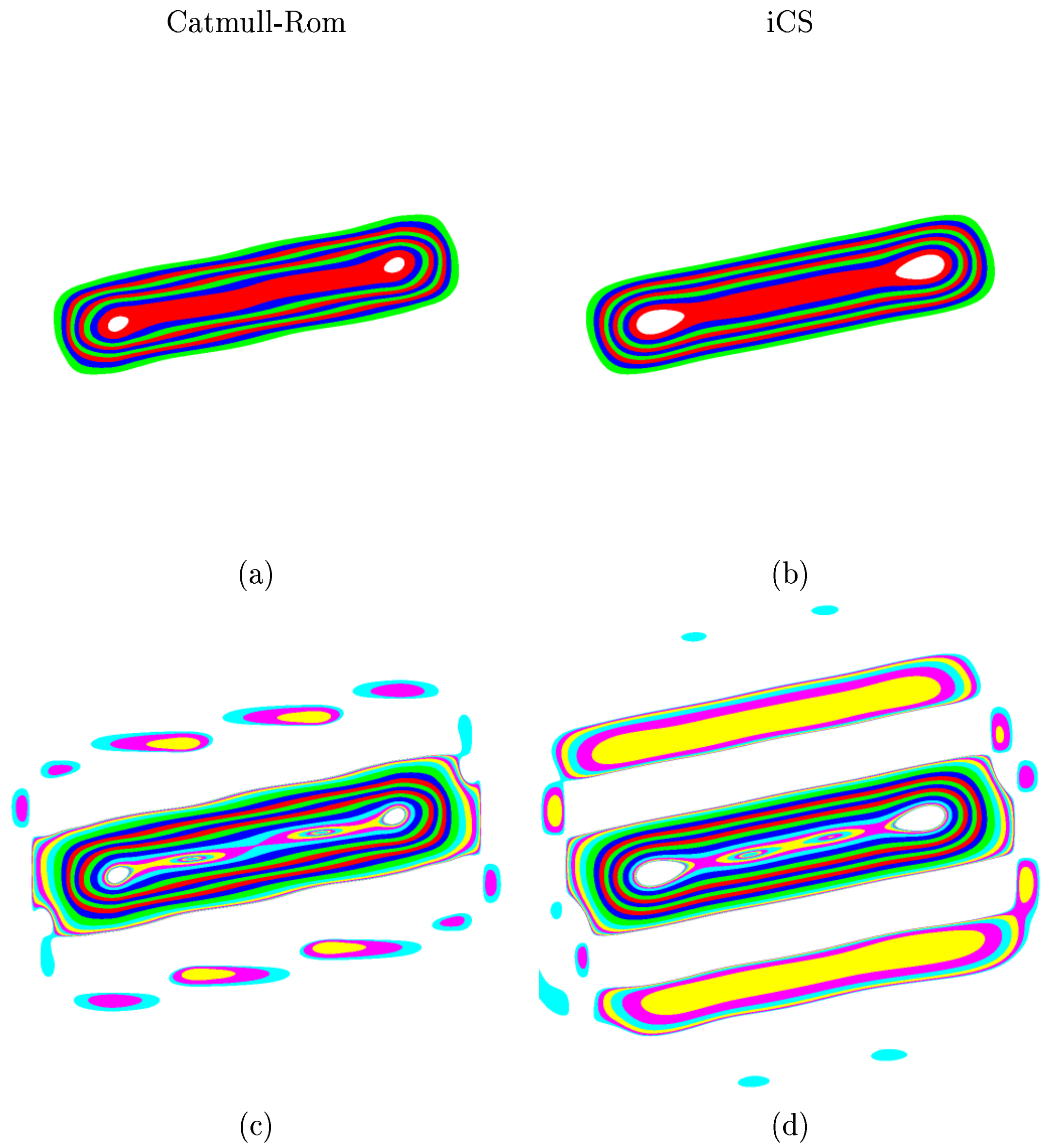


Figure 3.8: Images (a) and (c) are Catmull-Rom and (b) and (d) are iCS. Images (a) and (b) are the slab, rotated by 0.2 radians, double sampled. Images (c) and (d) are the slab, rotated by 0.2 radians, double sampled, with the color map structured to show the “echo” artifacts discussed in section 3.3.4.

table. These images show the overshoot artifacts of interpolating schemes. Note how much stronger the overshoot artifact is in the interpolating uniform cubic B-spline method.

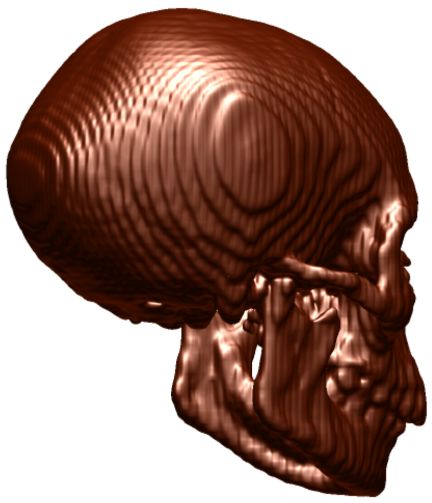
These images exhibit the superiority of the interpolating uniform cubic B-spline method over the Catmull-Rom method. The next step is to apply these methods to the trivariate case.

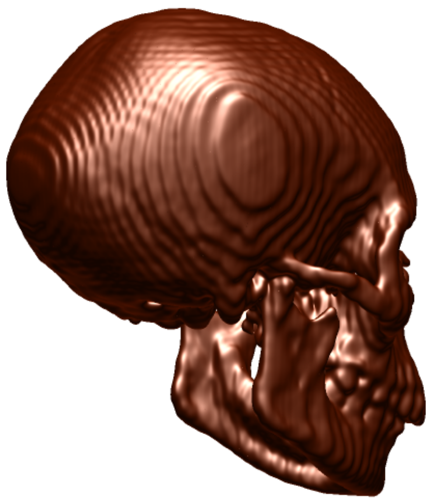
3.5 Results with Trivariate Scalar Field Data

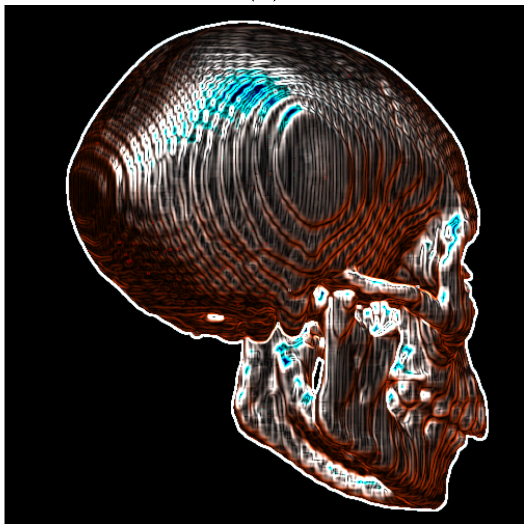
We show a few pictures demonstrating the results using the ray-tracer, see images 3.9a through 3.9d. The iterative solution step in three dimensions is the dominant step in the rendering pipeline. Table 3.6 shows representative times for rendering the skulls shown in images 3.9a through 3.9d. These images were rendered on a 16-processor SGI Origin 2000 system[1]. Note that the "Intervals" stage for the Catmull-Rom method is more complicated that the interpolating cubic B-spline stage, which accounts for the variation. Also, the "Solve" step performed only four iterations; the time for the Solve step is proportional to the number of iterations.

Lastly, we compute the normals to the surface analytically (differentiating the trivariate interpolation uniform cubic B-spline). This approach is much better than the original Catmull-Rom method (where we computed the normals by splining the central differences) and is considerably better than the current Catmull-Rom method (again, analytically). The only times where artifacts appear is for fairly small isovalues (less than 0.2), and they are still very hard to see, even under animations. The representation of isosurfaces greater than 0.2 is considerably better.

[1]Using only eight processors, with several users on the system.


(a)


(b)


(c)

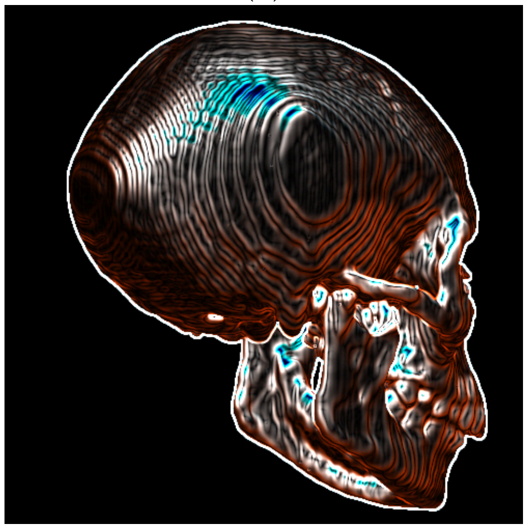

(d)

Figure 3.9: A comparison of Catmull-Rom sampling and raytracing with interpolating cubic B-spline sampling and raytracing: (a) Catmull-Rom; (b) interpolating cubic B-spline . Images (c) and (d) are the images from (a) and (b) respectively, filtered with a Sobel operator.

Step	Catmull-Rom	iCS
Rotate	1.51	1.59
Solve	n/a	13.70
Intervals	2.96	1.63
Ray-trace	3.28	3.20
misc.	0.42	0.38
Total	8.17	20.50

Table 3.6: Representative running times (in seconds) for Catmull-Rom and interpolating uniform cubic B-spline (iCS) methods.

3.6 Conclusions and Future Work

The interpolating cubic B-spline is a superior method to the Catmull-Rom spline for the purpose of resampling and isosurface extraction. The computational cost for this, however, is rather high. One avenue for future research is to find a method that can perform this curve fitting faster.

Another issue is the sparseness of the data: we are only interested in viewing a very small portion of the volume. For most data sets, the isosurface passes through a small fraction of the voxels, and this ratio goes up as the grid size increases. This means that most of the sampling and interval calculation is wasted. One solution may be to characterize the base data set, separating it into flat and non-flat regions. Evaluating the spline in a flat region is superfluous, since the function will always be the same. Similarly, the interval will be a single point in a flat region (see section 2.5.1. The "Sampling" and "Interval" steps can then skip over flat regions.

Part II

Multiresolution Techniques for Visualization

Chapter 4

Multiresolution Volume Visualization

In this chapter, we present a multiresolution technique for interactive texture-based volume visualization of very large data sets. This method uses an adaptive scheme that renders the volume in a region-of-interest at a high resolution and the volume away from this region at progressively lower resolutions. The algorithm is based on the segmentation of texture space into an octree, where the leaves of the tree define the original data and the internal nodes define lower-resolution versions. Rendering is done adaptively by selecting high-resolution cells close to a center of attention and low-resolution cells away from this area. We limit the artifacts introduced by this method by modifying the transfer functions in the lower-resolution data sets and utilizing spherical shells as a proxy geometry. It is possible to use this technique to produce viewpoint-dependent renderings of very large data sets.

4.1 Introduction

The capability of computing technology has steadily increased for more than four decades and continues to increase rapidly. These increased computing capabilities have enabled applications to scale accordingly in overall throughput and resulting data set sizes. However, current visualization techniques break down when operating in this environment due to the massive size of the data sets. New techniques are necessary to provide exploration of large, multidimensional data sets.

We combine hardware-assisted texture mapping and multiresolution methods for rendering large volumetric data sets. The general idea is to assign priorities to different regions of the volume and to render the high-priority regions with highest accuracy, while lower-priority regions are rendered with progressively less accuracy, and progressively faster.

We use an octree to decompose texture space and produce several coarser levels of the original data set. Each level is associated with a level in the octree and each level is half the resolution of the next level. The leaf nodes are associated with the original resolution, and the root node with the coarsest resolution. The interior nodes are created by subsampling the node's eight child nodes.

Rendering a volume involves traversing the octree and applying a selection filter to each node. Three results are possible: (1) The node (and its children) are skipped entirely; (2) the node is skipped, but its children are visited; or (3) the node is rendered, and the children are skipped. The selected nodes are then sorted and rendered in back-to-front order. We use spherical shells for proxy geometries for accuracy under perspective projections.

Section 4.2 provides a survey of related work. Section 4.3 discusses data issues for the multiresolution representation of textures, and Section 4.4 addresses the rendering of these textures. Section 4.5 shows results of the method on a number of data sets and gives performance results. Conclusions and future work are presented in Section 4.6.

4.2 Related Work

High-performance computer graphics systems are evolving rapidly. Silicon Graphics, Inc. (SGI) has been a primary developer of this rendering technology, introducing the RealityEngine graphics system [Ake93] in 1994 and the InfiniteReality graphics system [MBDM97] in 1998. SGI has also extended its graphics library OpenGL [SA98], [MB98] to take advantage of this hardware. These systems provided the initial capability for hardware-based rendering using solid textures.

Cabral *et al.* [CCF94] show that volume rendering and reconstruction integrals are generalizations of the Radon and inverse Radon transforms. They show that the Radon and inverse Radon transforms have similar mathematical forms, and by developing this relationship, show that both volume rendering and volume reconstruction can be implemented with hardware-accelerated textures. Thus, their algorithms execute many times faster than traditional software approaches.

Cullip and Neumann [CN94] discuss general implementation issues for hardware textures and are the first to generate pictures using this technique based on two different transfer functions. Their work illustrates the superiority of viewport- versus object-aligned sampling planes.

Wilson *et al.* [WVW94] and Van Gelder and Kim [VK96] develop the mathematics for generating texture coordinates. Van Gelder and Kim also introduce a quantized gradient method for shading. Here, a triangulated sphere describes quantized normals which, when coupled with a quantized set of material values, allows the construction of a look-up table. For each new scene and texture block, the current viewing and lighting parameters are applied to the look-up table, and the look-up table is applied to the texture map as it is transfered to the texture subsystem. They report interactive rates, both for orthographic and perspective projections. However, low gradient regions show traditional quantization artifacts.

Westerman *et al.* [WE98] show how to visualize isosurfaces resulting from rectilinear and unstructured grids. They use fragment testing to draw only those pixels that have a density value over a given threshold. Rectilinear grids are rendered by solid-texturing, which is shown to be much faster than the unstructured grid method. They also demonstrate how shade the texture-based isosurfaces with a technique that performs the shading as the texture map is transfered to the texturing subsystem.

Grzeszczuk *et al.* [GHY98] enumerate most methods for using hardware-accelerated texturing to provide interactive volume visualization. They also introduce a library for texture-based rendering called *Volumizer*, see [Eck98].

Massively parallel computers have been used to provide interactive volume visualization and isosurface extraction, see [OHA93], [HKP$^+$95], [HKW95], and [HH92]. Both ray-tracers and marching-cubes algorithms have been implemented, and both are very parallelizable. The overhead of data distribution and image composition is very high, and requires careful partitioning and tuning.

Our new method differs from these prior approaches in the sense that we allow adaptive rendering of a volume. Prior algorithms assume that the data is “uniformly complex” and “uniformly important.” This is not the case, for example, in an immersive environment, where data closer to the viewer has more visual importance

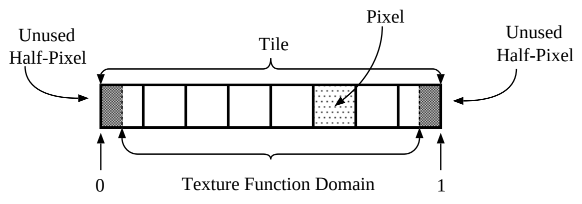


Figure 4.1: A one-dimensional texture tile containing eight pixels. The tile is one pixel larger than the texture function domain. We assume that the half-pixels at the edges are constant.

than data far away. Also, quality should be a “tunable” parameter: If a graphics supercomputer is not available or a user just wishes to quickly browse a data set, then the user will be satisfied with a poorer rendering quality.

4.3 Generating the Hierarchy

In hardware texturing algorithms, linear interpolation is used to interpolate the values at the centers of adjacent pixels. If we consider the one-dimensional example shown in Figure 4.1 and assume that the “tile” contains p pixels, then the texture function domain is the interval $[\frac{1}{2p}, \frac{2p-1}{2p}]$. If the unused half-pixels are clipped, a larger texture can be broken into a set of smaller textures or tiles, where interior edge pixels are duplicated between adjacent tiles. This technique is known in the literature as “bricking,” see [GHY98].

In Figure 4.2, we show a two-level texture hierarchy. The higher-resolution texture is denoted as level A, with tiles A^0 and A^1, and the lower-resolution one as level $\mathbf{B}$. The grey regions at the ends are unused, and the grey region shared by A^0 and A^1 indicates the “overlay pixel.” The image represented by A can be approximated by B. The image represented by B has the same number of pixels as A^0 or A^1, and half the number of pixels of A. We note that the natural relationship for two textures

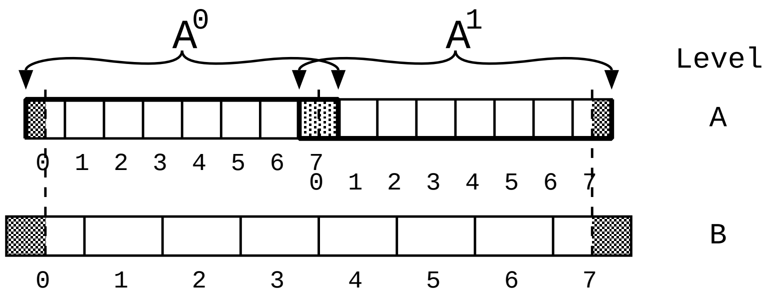


Figure 4.2: A texture hierarchy of two levels.

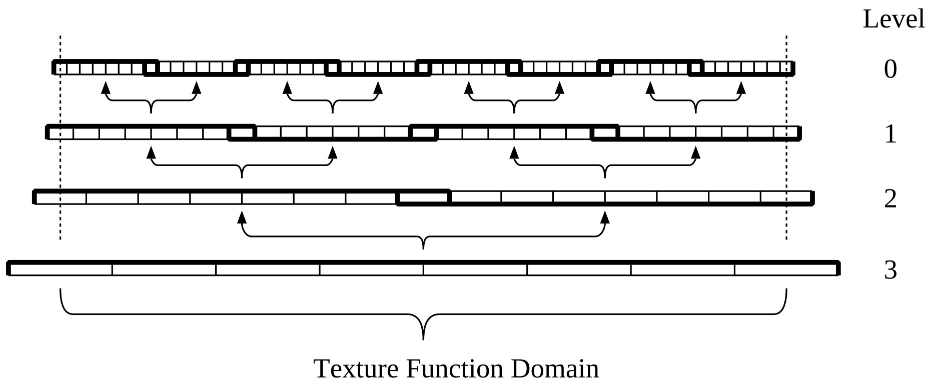


Figure 4.3: A texture hierarchy of four levels. Level 0 is the original texture, broken into eight tiles. Overlapping pixels are shared between the tiles. The dashed lines show the texture function domain for each level.

whose resolutions differ by a factor of two is using pixel-center alignment. In the binary tree arrangement defined by this one-dimensional texture B is the parent of A^0 and A^1.

4.3.1 The Multiresolution Texture Hierarchy

Figure 4.3 shows a one-dimensional texture hierarchy of four levels. The top level, level 0, is the original texture, broken into eight tiles; level 1 contains four tiles at half of the original resolution; level 2 contains two tiles at a quarter of the resolution; and level 3 has one tile at an eighth of the original resolution. The dashed vertical lines

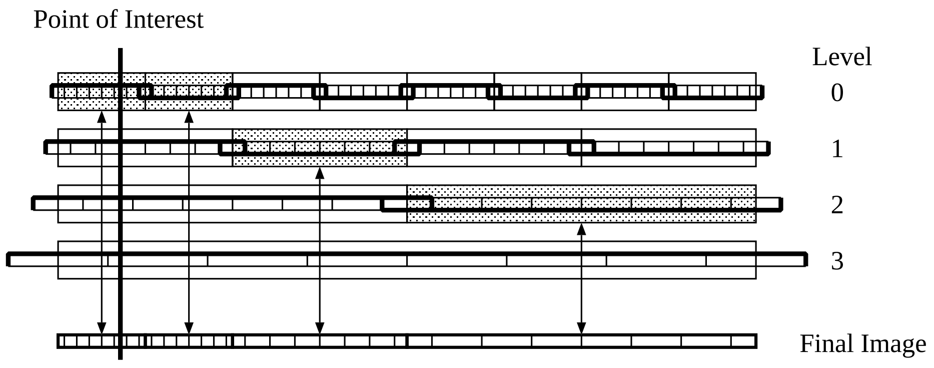


Figure 4.4: Selecting tiles from a texture hierarchy of four levels. The vertical line represents a point of interest p. Tile selection depends on the width of the tile and the distance from the point. (Selected tiles are shaded, with their position shown in the "Final Image" by double-headed arrows.)

on either side show the domain of the texture function over the entire hierarchy. The arrows denote the parent-child relationship of the hierarchy, defining a binary tree, rooted at the coarsest tile, level 3.

Figure 4.4 illustrates the logic for selecting tiles in a multiresolution environment: The thick vertical line denotes a point of interest, p, and tiles are selected if the distance from p to the center of the tile is greater than the width of the tile. Start with the root tile and perform this selection until all tiles meet this criterion, or no smaller tiles exist. (This is the case on the left side of Figure 4.4.)

Figure 4.5 shows a two-dimensional quadtree example. The original texture, level 0, has 256 tiles. The darker regions in each level show the portion of that level used to approximate the full image. The selection method is similar to the one-dimensional case: Select a node if the distance from the center of the node to the point p is greater than the length of the diagonal of the node. The original texture is divided into 256 tiles. The adaptive rendering uses 49 tiles, or approximately one-fifth the data of the original. While, in 2D, this only reduces the amount of data transmitted and

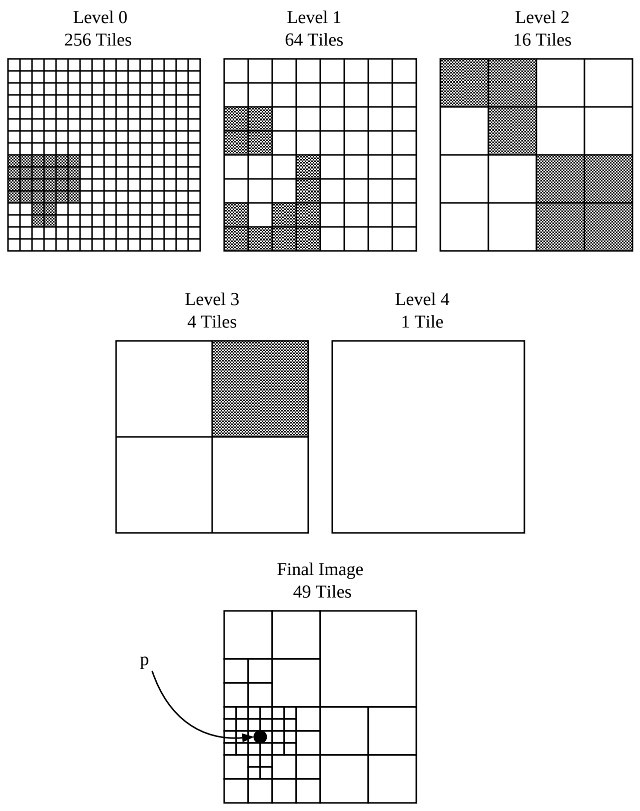


Figure 4.5: Selecting a tile set in two dimensions from a texture hierarchy of five levels. Given the point p, tiles are selected if the distance from the center of the tile to p is greater than the length of the diagonal of the tile. (Selected tiles are shaded.)

rendered, it will, generally, not reduce the rendering time. This formulation extends in a straightforward manner to three-dimensional textures; however, both the amount of data transmitted and rendered, and the rendering time will be reduced.

We generate a texture hierarchy by subsampling textures.[1] Subsampling chooses every other voxel when generating a lower-resolution data set. For example, if A is a linear array of $2n$ elements and B is a linear array of n elements that approximates A, then we generate the elements of B as $B_i = A_{2i}, i = 0, 1, \ldots, n-1$.

How much memory is wasted by breaking a volume into bricks? The waste is generated by the outer layer of voxels, which is shared by adjacent tiles. If a brick has size n (n^3 voxels) and is surrounded by a half-voxel layer of duplicate voxels, the effective size of a brick is $n-1$, and there are $n^3 - (n-1)^3$ "waste" voxels. The waste relative to the tile size is $O(n^2/n^3) = O(n^{-1})$, which means that as the tile size increases the relative waste decreases.[2] If we choose a tile size of 64^3, the tile contains 262,144 voxels, 250,047 effective voxels and 12,097 extra voxels.

4.4 Rendering

The rendering phase is divided into three steps: (1) Selecting tiles to render; (2) sorting the tiles; and (3) rendering the tiles using a proxy geometry.

4.4.1 Selecting Tiles

The first rendering step determines which tiles will be rendered. The general filtering logic starts at the root tile and performs a depth-first traversal of the octree. At each

[1]We have also tried several other methods, including averaging techniques, but the results are substantially better using subsampling for the data sets used in this study.

[2]The total waste still increases.

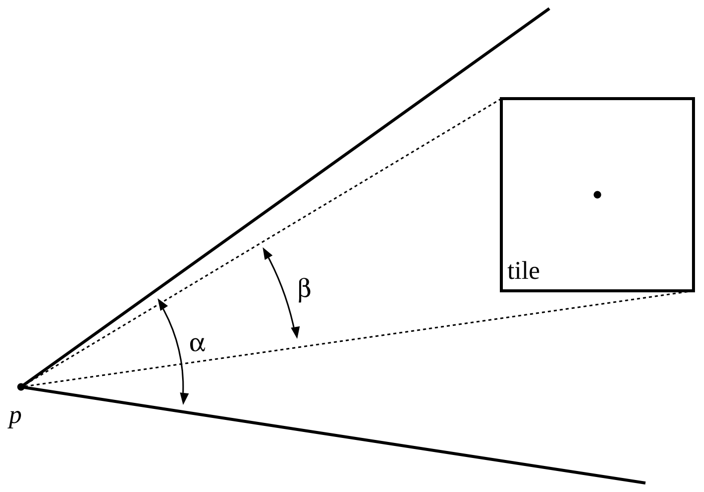


Figure 4.6: Field-of-view selection filter. The projected angle β of the tile is less than half the field-of-view angle α.

tile, we evaluate a selection filter, which returns one of three possible responses:

- Ignore this tile and all of its children. This response is used to cull the tree. For example, if a tile is not in the view frustum, then we can ignore the tile and its children.
- The tile satisfies all criteria. Render the tile and do not consider the children.
- The tile does not satisfy the criteria. Check the children of the tile.

Our primary selection filter is based on one of the following criteria:

- **Distance.** This filter selects a tile if the distance from the viewpoint to the center of the tile is greater than the diagonal length of the tile.
- **Field-of-View.** This filter selects a tile if it intersects the view frustum and the projected angle of the tile is less than half the view frustum's field-of-view angle, see Figure 4.6.

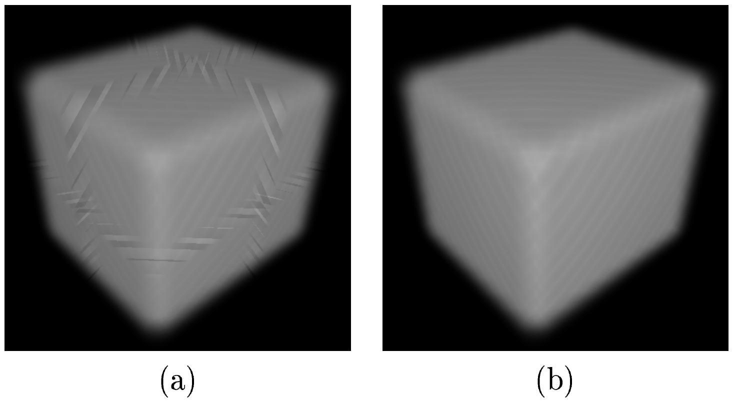


Figure 4.7: Differences between (a) viewport-aligned planes and (b) spherical shells on a constant texture. Note the artifacts at the “tile boundaries” visible in (a).

Tiles must be sorted and composited in back-to-front order. We order tiles with respect to a view direction such that, when drawn in this order, no tile is drawn behind a rendered tile. The order is fixed for the entire tree for orthogonal projections, and has to be computed just once for each new rendering, see [GHY98]. The order is not fixed for perspective projections, and it must be computed at each new node.

4.4.2 Proxy Geometries

Texture-based volume visualization requires proxy geometries on which to render the texture. Object-aligned planes and viewport-aligned planes are two traditional techniques, but they lead to serious artifacts under perspective projections. To deal with these artifacts, we use “spherical shells” – finely tessellated concentric spheres surrounding the viewpoint, culled to the view frustum. Figure 4.7 shows the differences when viewport-aligned planes and spherical shells are used as proxy geometries on a constant texture defined over a cube.

Object-aligned planes (OAP) are implemented with two-dimensional textures on polygons aligned with the xy-, yz-, and xz-planes of the volume. This is the fastest

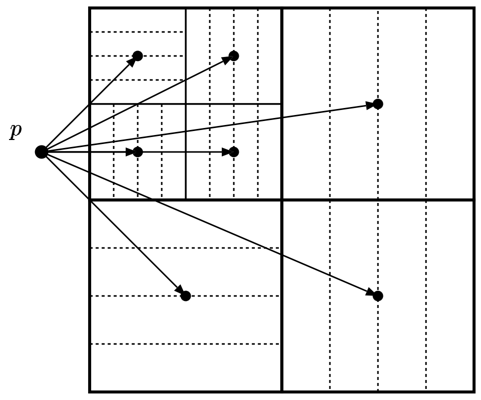


Figure 4.8: Multiresolution object-aligned planes. The proxy geometry is generated independently for each tile.

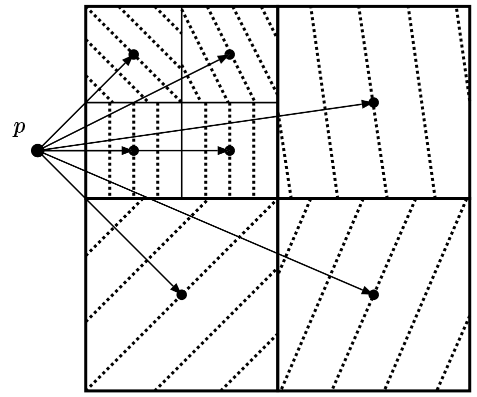


Figure 4.9: Multiresolution viewpoint-aligned planes. The proxy geometry is generated independently for each tile. Substantial artifacts can appear at the boundaries.

method, and it is supported by most contemporary graphics workstations. However, three sets of polygons must be maintained. If only one set is used, certain viewpoints will lead to an “edge-on situation,” and nothing will be rendered. Also, the light attenuation is not computed correctly as the projected distance between polygons is not constant. This error is worst for an angle of 45^o. Figure 4.8 illustrates multiresolution OAP in two dimensions. Here, OAPs planes must be generated independently for each tile, generating substantial artifacts at the boundaries of the tiles.

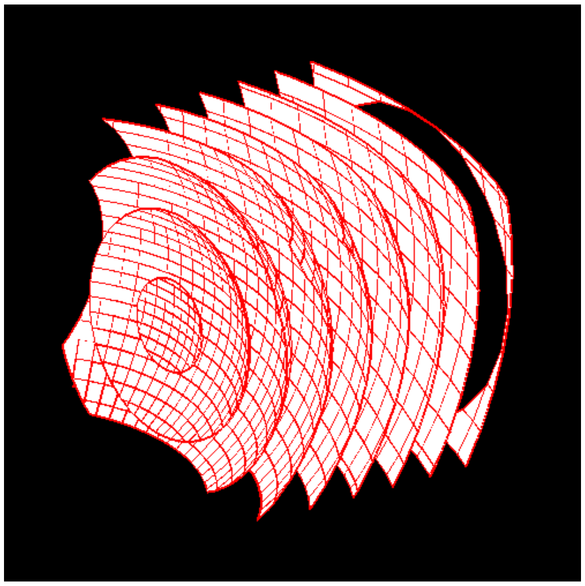


Figure 4.10: Spherical shells intersecting the voxelized data set. These shells provide a proxy geometry that can be adapted to the location of the viewpoint.

Viewport-aligned planes (VAP) are implemented with three-dimensional textures and polygons that are aligned parallel to the viewport. Only one texture set is required. All orthographic projections yield visually correct results. Three-dimensional texturing is currently only supported on high-end workstations. Rendering based on VAPs is generally less than half as fast as OAPs because (1) the Nyquist sampling theorem requires twice the number of polygons as OAPs requires and (2) the underlying computations are more complex. Figure 4.9 shows the use of multiresolution VAPs. This technique creates strong artifacts under perspective projections. The tiles are rendered with a differently oriented set of polygons, and these polygons do not meet at the tile borders. This creates an artifact reminiscent of the "cracking" artifact from multiresolution polygonal schemes, and it manifests itself as light-and-dark alternating bands at the tiles boundaries.

Viewpoint-centered spherical shells (VCSS) are implemented with three-dimensional

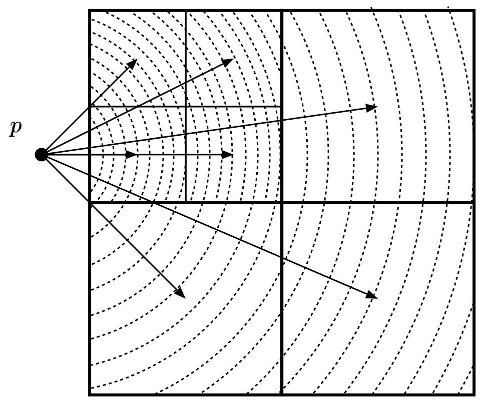


Figure 4.11: Multiresolution viewpoint-centered spherical shells. The differences between sampling on the boundaries of the tiles is relatively small.

textures and use concentric spherical shells centered at the viewpoint, culled to the view frustum. This technique does not produce artifacts under perspective projections, but it is slower than VAPs due to the increased geometric complexity required to approximate a spherical shell. In Figure 4.10, the viewpoint is on the left-hand side, almost touching the volume. The sample interval is exaggerated to show the structure – one shell every two voxels. Figure 4.11 illustrates multiresolution VCSSs. Using this approach, one can achieve continuity across tile faces.

4.4.3 Preserving Visual Properties

When rendering tiles at different levels of the hierarchy, the opacity properties of the tiles are different. The classical rendering algorithms depend on using the same sampling along rays for each pixel, see [Lev87]. But in the context of a multiresolution format, the the volume is sampled in different ways, and at varying resolutions. To preserve optical properties between tiles of different resolutions, we must modify the transfer functions for those tiles generated by subsampling the original texture.

Figure 4.12 shows an example where we have sampled a texture with spherical

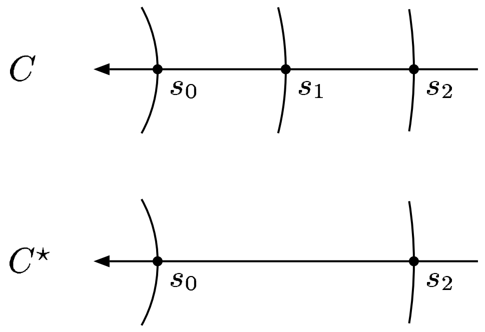


Figure 4.12: When the space is sampled at two different resolutions, the colors C and $C^\star$ should be the same.

shells at two different resolutions – one is half the resolution of the other. Each sample s_i has an associated color c_i and an opacity value α_i. The light emitted by s_i is a function of the incoming light, and the color and opacity properties of the sample itself. Following [Lev87], the color C resulting from the higher-resolution sampling is

$$C = \alpha_0 c_0 + (1 - \alpha_0) C_1, \tag{4.1}$$

where C_1 is the incoming color from samples $s_1, s_2, \ldots$ – that is

$$C_1 = \alpha_1 c_1 + (1 - \alpha) C_2, \tag{4.2}$$

where C_2 is the incoming color from samples $s_2, s_3, \ldots$ For the coarse resolution, the color $C^\star$ is given by

$$C^\star = \alpha^{\star 0} c_0 + (1 - \alpha^{\star 0}) C^{\star 2}, \tag{4.3}$$

where $C^{\star_2}$ is the color calculated as a result of the samples $s_2, s_4, s_6, \ldots$

By considering only the first three samples, the resulting colors C and $C^\star$ are given by

$$\begin{aligned} C = &\quad \alpha_0 c_0 + (1-\alpha_0)\alpha_1 C_1 + (1-\alpha_0)(1-\alpha_1)C_2 \text{ and} \\ C^\star = &\quad \alpha^{\star_0} c_2 + (1-\alpha^{\star_0})C^{\star_2}, \end{aligned} \tag{4.4}$$

and these quantities, in general, are different.

However, if we compute the total-accumulated opacities A and $A^\star$, we obtain

$$\begin{aligned} A = &\quad \alpha_0 + (1-\alpha_0)\alpha_1 + (1-\alpha_0)(1-\alpha_1)A_2 \text{ and} \\ A^\star = &\quad \alpha^{\star_0} + (1-\alpha^{\star_0})A^{\star_2}. \end{aligned} \tag{4.5}$$

Assuming that the accumulated opacities are equal at the even samples, it follows that $A_2 = A^{\star_2}$ and $A = A^\star$, i.e.,

$$\alpha_0 + (1-\alpha_0)\alpha_1 + (1-\alpha_0)(1-\alpha_1)A_2 = \alpha^{\star_0} + (1-\alpha^{\star_0})A_2, \tag{4.6}$$

Solving this equation for $\alpha_0^\star$, one obtains

$$\begin{aligned} \alpha^{\star_0} &= \alpha_0 + (1-\alpha_0)\alpha_1 \\ &= 1 - (1-\alpha_0)(1-\alpha_1). \end{aligned} \tag{4.7}$$

By assuming that $\alpha_1 = \alpha_0 + \epsilon$ (where ϵ is a very small number), we obtain the equation

$$\begin{aligned} \alpha^{\star 0} &= 1 - (1-\alpha_0)(1-\alpha_1) \\ &= 1 - (1-\alpha_0)(1-\alpha_0) + \epsilon(1-\alpha_0). \end{aligned} \tag{4.8}$$

Therefore, we modify the transfer function of the parent (coarser) texture by

$$\alpha^{\star} = 1 - (1-\alpha)^2 \tag{4.9}$$

for all opacity values in the subsampled texture to minimize the artifacts between the texture bricks. This formula is used when applying the transfer function to a level of the texture hierarchy.

4.5 Results

We have implemented our algorithm and applied it to several complex data sets. All data sets were rendered on an SGI Onyx2 computer system with .5 gigabytes of main memory, using a single 195Mz R10000 processor.

The first data set is a CT scan of a horse metacarpus. This data set consists of $128 \times 128 \times 108$ voxels. Figure 4.13 shows this data set. The primary feature of this data set is the "empty interior" of the bone, which can be visualized only from close inspection of one end. Figures 4.14a and 4.14b were generated using fixed tile sizes, while Figures 4.14c and 4.14d were generated using our multiresolution technique. In Figure 4.14a, we show the viewpoint and bricks associated with the rendering in

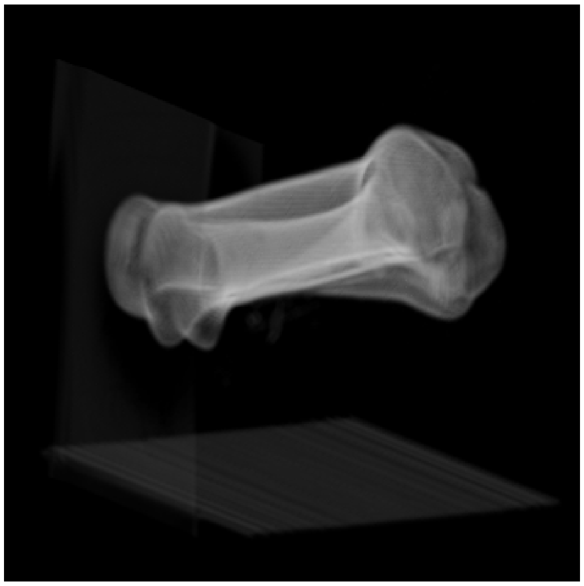


Figure 4.13: The horse metacarpus data set

Figure 4.14b. In Figure 4.14c, we show the bricks used for the adaptive rendering of the texture, resulting in the image in Figure 4.14d.

The second image is a rendering of a trebecular bone data set, shown in Figure 4.14e. The data set consists of 256^3 voxels, and the interesting features are inside the data set. Figures 4.14f and 4.15g–i show two views of this data set, one from just outside the data set, and one from the interior. The respective viewpoints and bricks associated with the views are shown in Figures 4.14f and 4.15h, while the resulting renderings are shown in Figures 4.15g and4.15i.

The third image is a rendering of data generated from an immersive auditory interface system, shown in Figure 4.15j. This data set has a “channel” in the middle of the volume. An image from the interior of the channel is shown in Figures 4.15k and 4.15l. Statistics about rendering times for the various algorithms applied to the three data sets are given in Table 4.1.

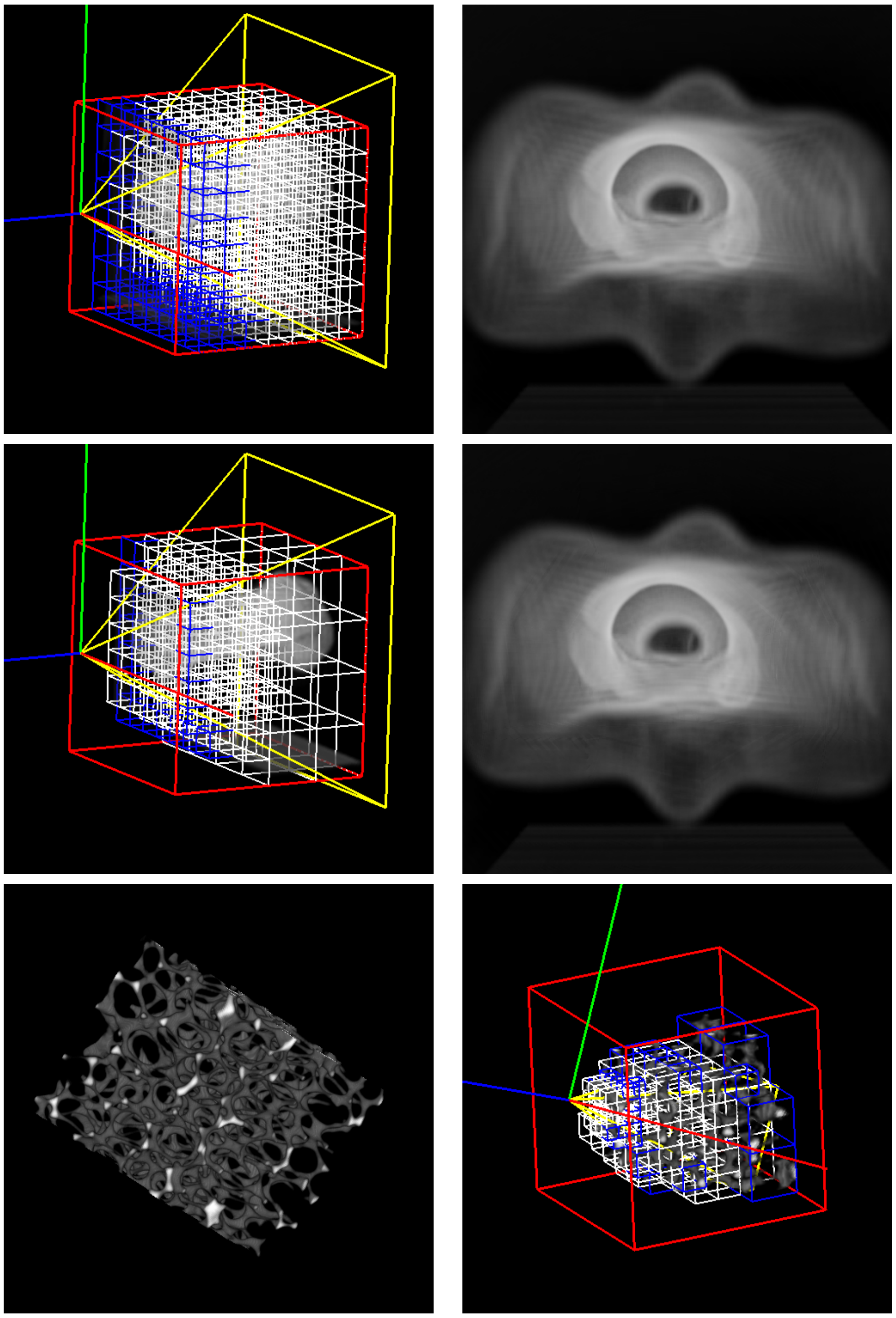


Figure 4.14: Multiresolution Texture-based Volume Rendering, plate 1.

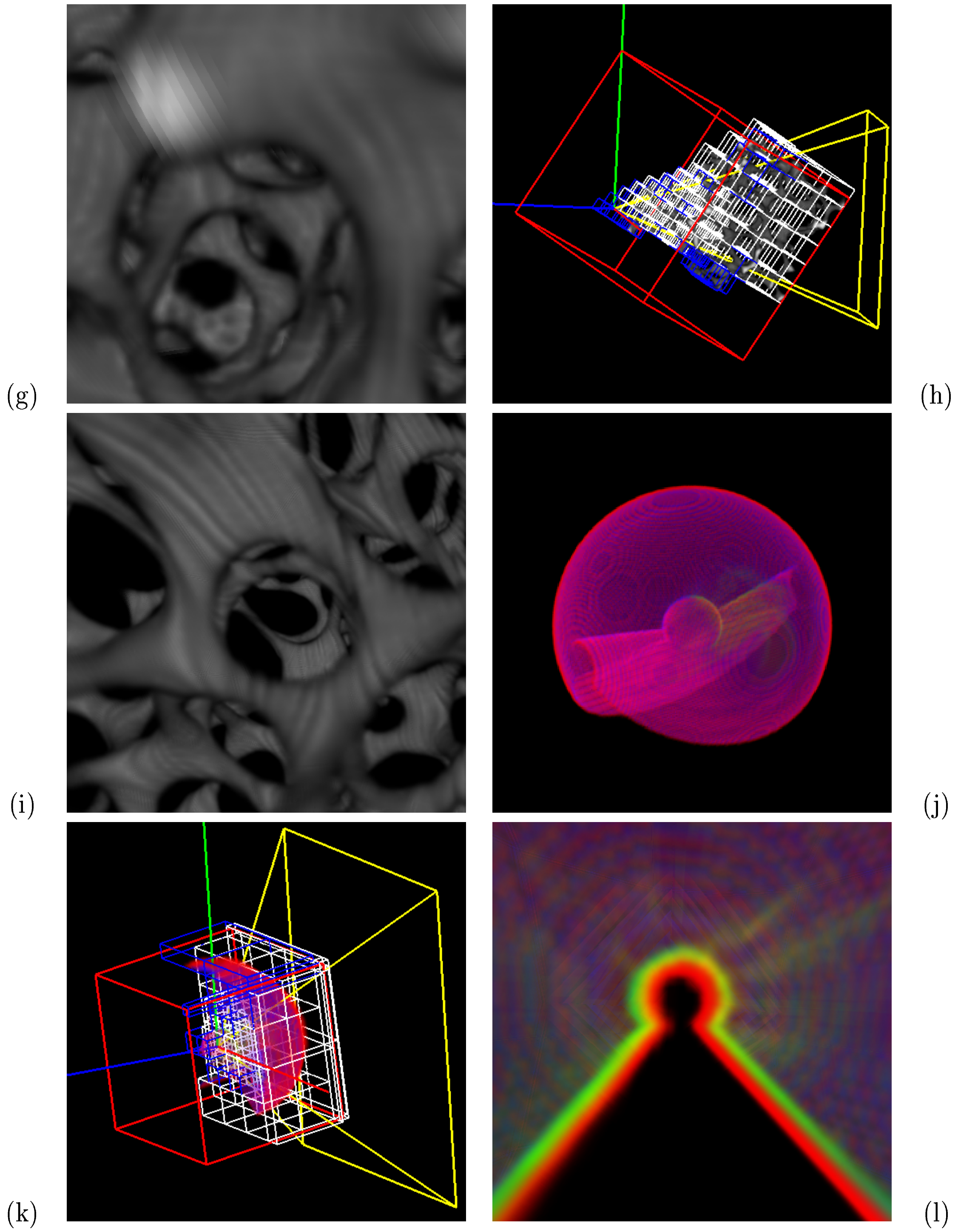


Figure 4.15: Multiresolution Texture-based Volume Rendering, plate 2.

Model	Horse Metacarpus	Immersive Auditory	Trebecular Bone
Data Set Size	$128^2 \times 108$	126^3	256^3
Tile Size	16^3	16^3	16^3
Number of Tiles (Fixed Tile Size)	391	237	1049
Number of Tiles (Multiresolution)	195	64	397
Time (OAP) (Fixed Tile Size)	0.848	0.419	2.25
Time (OAP) (Multiresolution)	0.424	0.123	0.853
Time (VAP) (Fixed Tile Size)	1.51	1.10	4.20
Time (VAP) (Multiresolution)	0.964	0.683	2.23
Time (VCSS) (Fixed Tile Size)	2.87	1.73	8.94
Time (VCSS) (Multiresolution)	1.530	0.830	3.93

Table 4.1: Rendering times for the various data sets. (All times are in seconds.)

4.6 Conclusions

We have described a new method for building and rendering a multiresolution texture hierarchy approximation for very large data sets. The approach utilizes a "bricking" strategy, where the displayed bricks are selected from an octree representation. Despite the fact that our overall system is limited by the amount of available texture memory, the algorithm produces very good results, and we expect that this approach will have a major impact on the huge volumetric data sets that are currently encountered in numerous applications. Future work will involve the extension of the technique to vector fields and the parallelization of our algorithm.

Chapter 5

Transfer Functions Using Indexed Textures

We extend the work discussed in Chapter 4 to use indexed texture maps, which allow for interactive modification of the opacity transfer function.

5.1 Introduction

In this chapter, we combine hardware-assisted texture mapping and color table look-up with multiresolution methods for rendering large volumetric data sets. Texture mapping is substantially faster than software-based approaches, and color look-up tables allow for easy manipulation of transfer functions. The multiresolution principle assigns priorities to different regions of the volume and renders "high-priority regions" with highest accuracy, while "low-priority" regions are rendered with progressively less accuracy and progressively faster.

We use an octree to decompose texture space and produce several coarser levels of an original data set. Each level is associated with a level in the octree, and each level is half the resolution of the next level. The leaf nodes are associated with the original resolution and the root node with the coarsest resolution. Interior nodes are created by subsampling a node's eight child nodes. Each value in the texture map is an index, not an RGBA tuple; the transfer function is applied as the texture map is transferred to texture memory.

Rendering a volume involves traversing the octree and applying a selection filter to each node. Three results are possible: (1) The node (and its children) are skipped entirely; (2) the node is skipped, but its children are visited; or (3) the node is rendered, and the children are skipped. The selected nodes are then sorted in back-to-front order. Finally, for each node, the algorithm builds and transfers the color look-up table, transfer the texture map, and render the proxy geometry.

Section 5.2 addresses the rendering of multiresolution textures. Section 5.3 discusses issues of static and indexed texture maps. Section 5.4 shows results for two data sets and lists performance results. Conclusions and future work are presented in Section 5.5.

5.2 Rendering

The rendering phase is divided into three steps: (1) selecting tiles to render; (2) sorting the tiles; and (3) rendering the tiles using a proxy geometry.

5.2.1 Selecting Tiles

We add the following selection filter to the filters discussed in Section 4.4.1:

- **Cone criterion.** Selects the highest-resolution tiles within the viewing frustum. Its primarily use is to determine the speedup factor of the Field-Of-View criterion.

5.2.2 Proxy Geometries

We use *viewpoint-centered spherical shells* (VCSSs) proxy geometries, as presented in 4.4.2.

5.2.3 Preserving Visual Properties

See section 4.4.3 for a complete discussion of this topic. For the purpose of this chapter, we note that the correction formula is used when generating the texture look-up table from the transfer function.

5.3 Static and Index Texture Maps

Static texture maps are “pre-shaded” texture maps. The data is transformed by the transfer function, from a byte to an RGBA tuple, then stored in a texture map. Opacity correction is also applied during this step. This technique is useful since one can freely choose the transfer function. For example, opacity may be a function of the gradient. Transferring a static texture to the texture subsystem is very fast as no further transformations are necessary. However, the overall process is very slow since the transfer function is implemented in software, which requires that all data be touched for each new transfer function.[1] Lastly, static texture maps require two

[1]The results can be cached, for each transfer function, but at a very high memory cost.

or four times as much memory as an index texture map. The static texture map uses a byte for each RGBA (red, green, blue, alpha) value, or four bytes per texel, while index texture maps use one or two bytes per texel. For these reasons, this technique is inappropriate for situations where one may use a large number of different transfer functions.

Index texture maps are stored directly in the texture maps, interpreted as an index. Opacity correction is deferred until building the look-up table. Though performed in software, this step has very little overhead as it is performed once and mostly involves moving memory. Transferring an index texture map to the texture subsystem is more involved. A texture look-up table is built and transferred to the texture subsystem; the texture look-up table translates the index for a value for each of the RGBA channels. This is performed in hardware and is very fast. The texture look-up table is quite small, 256 to 65536 entries[2], and can be updated easily and quickly in software. Subsequently, the texture map is transferred and the texture subsystem translates that index to an RGBA tuple for each texel.

5.4 Results

We have implemented our algorithm and applied it to several complex data sets. All data sets were rendered on an SGI Onyx2 computer system with 512MB of main memory, 2GB of swap, and 16MB of texture memory, using a single 195Mz R10000 processor.

Figures 5.1a to 5.1d show an Equine Metacarpus data set. This is a CT scan

[2]At four bytes per entry, this results in 1024 bytes to 256KB; compared to a 1024^3 static texture map, which requires 4GB.

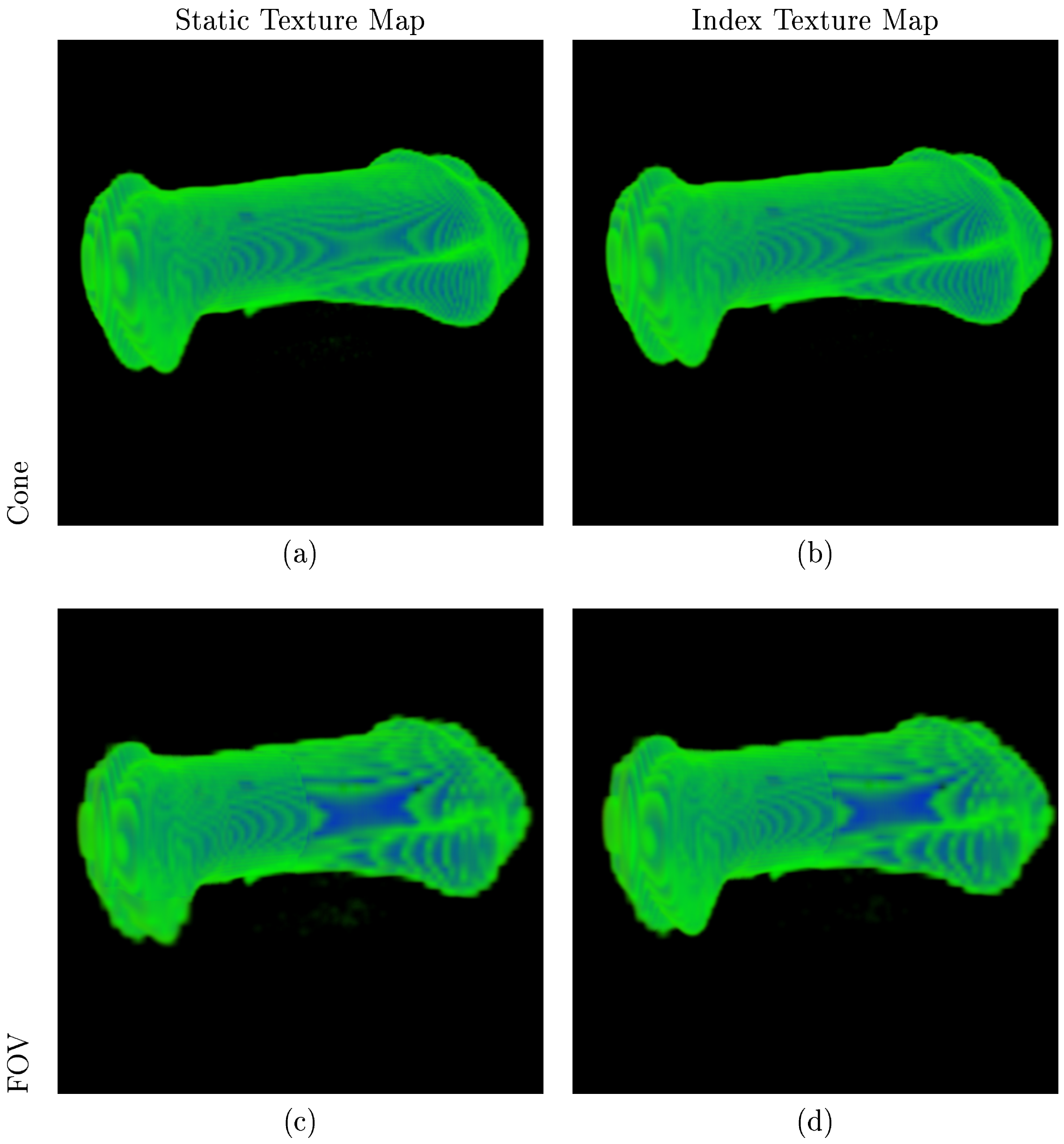


Figure 5.1: *Equine Metacarpus* data set. Image size is 500^2 pixels.

	Static Texture Maps		Index Texture Maps	
Data Set Size	$108^2 \times 126$			
Tile Size	16^2 texels			
Tiles at Level 0	448			
Tile Time	25.3 sec.		5.1 sec.	
Memory Used	8 MB		4 MB	
Filter	Cone	FOV	Cone	FOV
Number Of Tiles	432	148	433	150
Rendering Time	10.72 sec.	5.1 sec.	9.94 sec.	4.93 sec.

Table 5.1: Rendering statistics for the *Equine Metacarpus* data set. (Times are in seconds.)

data set consisting of $108^2 \times 126$ voxels. Each tile contains 16^3 texels. The number of tiles for levels 0 to 3 are: 448, 64, 8, and 1; for a total of 521 tiles. The memory requirements for static textures, with $16^3 \times 4 = 16384$ bytes per tile, is 8MB. The memory requirements for index textures, with $16^3 \times 2 = 8192$ bytes per tile, is 4MB. Images 5.1(a) and 5.1(c) were rendered using static texture maps, while images 5.1(b) and 5.1(d) were rendered using index texture maps. Images 5.1(a) and 5.1(b) use tiles from the Cone filter; images 5.1(c) and 5.1(d) use tiles from the FOV filter. While the rendering times are roughly the same, index texture maps require 5.1 seconds to generate all tiles ("Tile Time"), and is done only once. Static texture maps require 25.3 second to generate all tiles, and this must be done every time the transfer function is changed. Statistics concerning the rendering times for this data set are provided in Table 5.1.

Figures 5.2a to 5.2d and 5.3e to 5.3h show a Raleigh-Taylor instability data set. This data set is a single time step from a simulation and consists of $500^2 \times 768$ voxels. Each tile contains 64^3 texels. The number of tiles for levels 0 to 4 are: 1053, 175, 36, 8, and 1 (total: 1273 tiles). For index textures, this requires $1273 \times 64^3 \times 2 = 637$ MB of memory. Images 5.2(a), (c), 5.3(e), and (g) are rendered using tiles selected by the

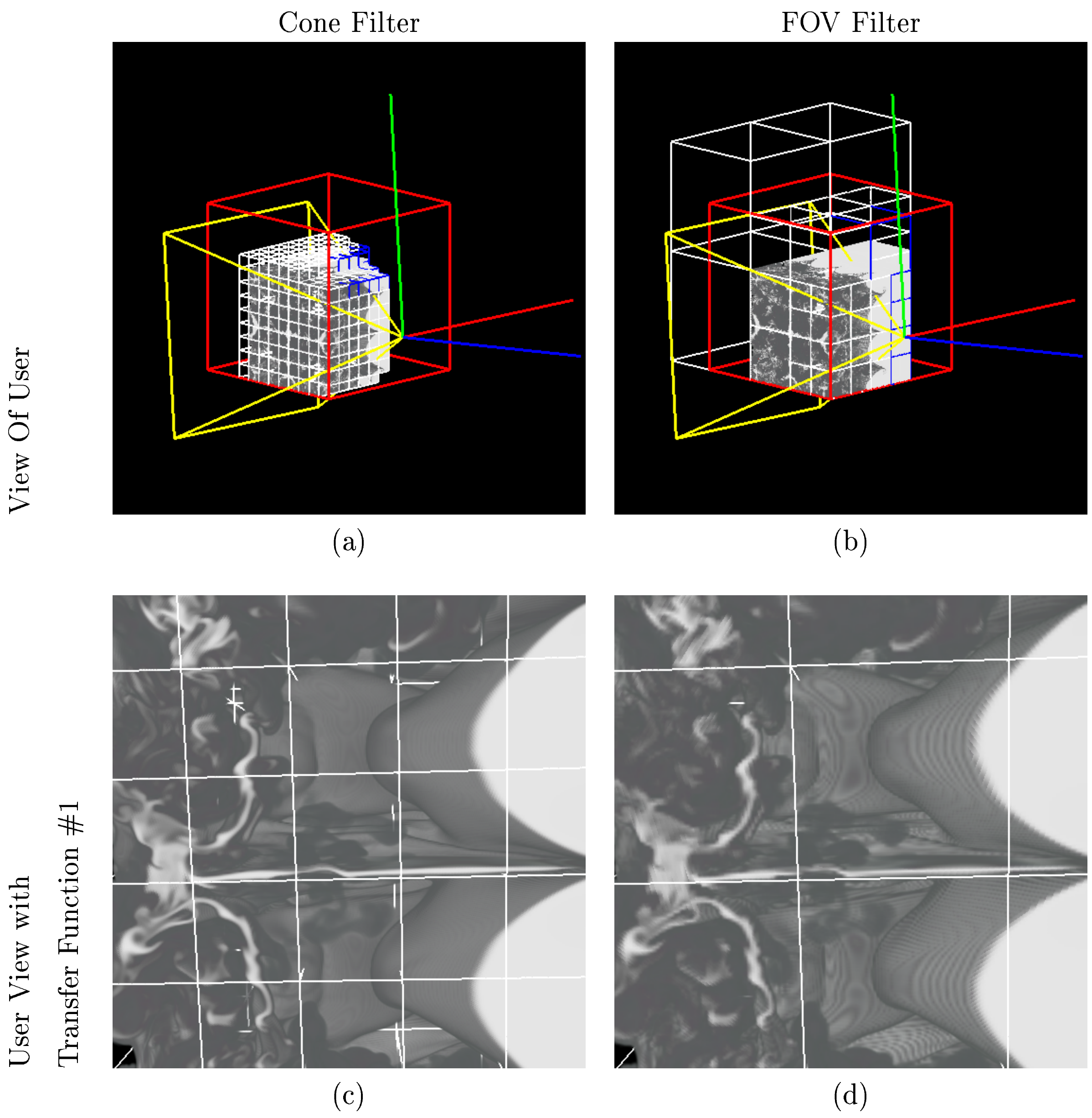


Figure 5.2: *Raleigh-Taylor instability* data set, plate 1. Images (a), (c), (e), and (g) rendered with tiles selected by the **Cone** filter. Images (b), (d), (f), and (h) are rendered with tiles selected by the **FOV** filter. Image size is 500^2 pixels. Images(a) and (b) show the viewing frustum, the yellow pyramid, with respect to the data.

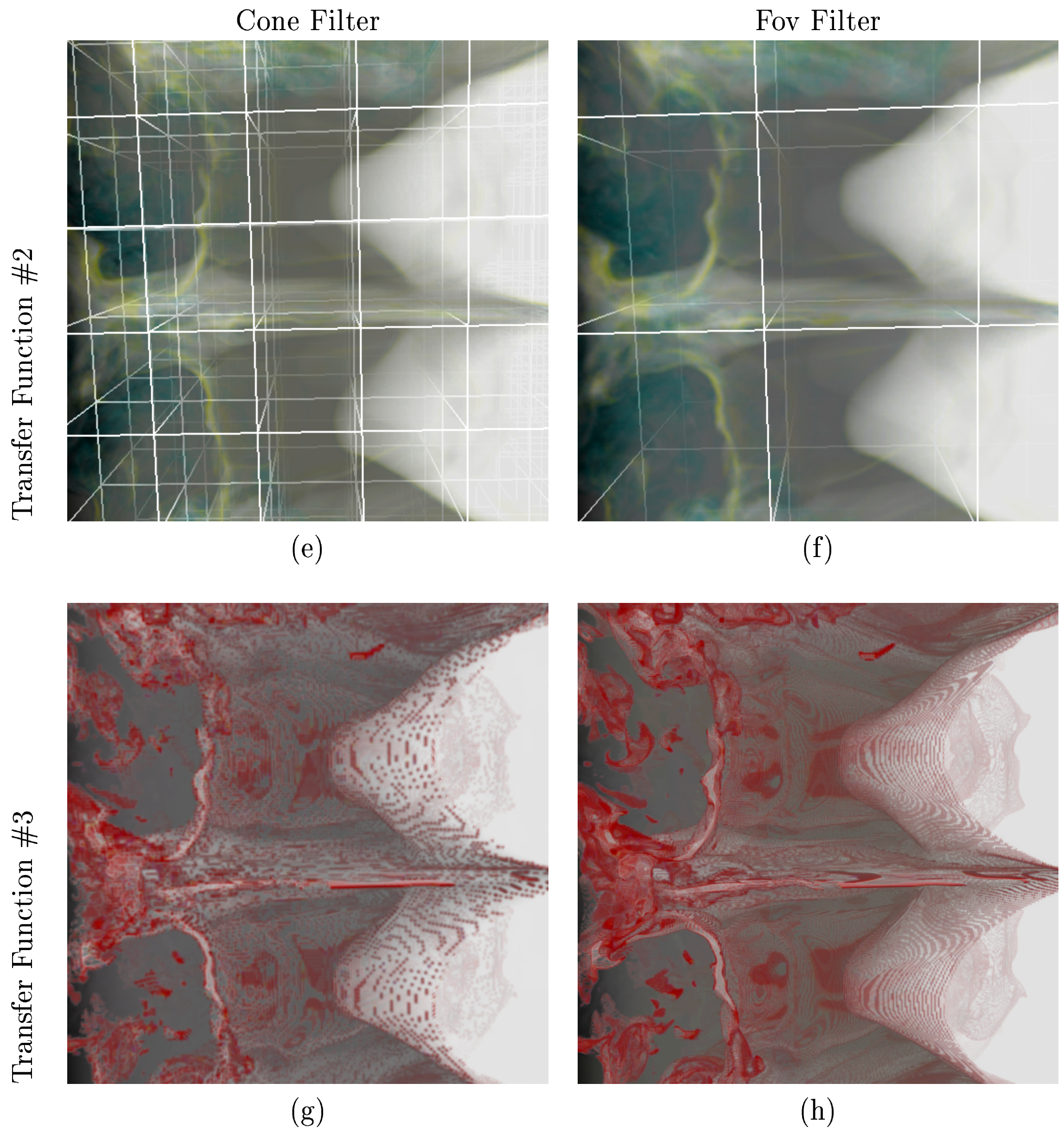


Figure 5.3: *Raleigh-Taylor instability* data set, plate 2. Images (a), (c), (e), and (g) rendered with tiles selected by the **Cone** filter. Images (b), (d), (f), and (h) are rendered with tiles selected by the **FOV** filter. Image size is 500^2 pixels.

Data Set Size	$500^2 \times 768$	
Tile Size	64^3 texels	
Tiles at Level 0	1053	
Tile Time	1049 sec.	
Memory Used	637 MB	
Filter	Cone	FOV
Number of Tiles	593	47
Rendering Time	25.4 sec.	2.47 sec.

Table 5.2: Rendering statistics for the *Raleigh-Taylor instability* data set. Times are in seconds.

Cone filter, requiring 593 tiles. Images 5.2(b), (d), 5.3(f), and (h) are rendered using tiles selected by the FOV filter, requiring only 47 tiles. The white grid lines show the tile boundaries. Images 5.2(c), 5.3(e), and (g) and images 5.2(d), 5.3(f), and (h) are rendered in perspective. Images 5.2(a) and 5.2(b) are rendered orthographically. The yellow pyramid is the viewing frustum of the perspective projections, and the white and blue boxes show the tiles boundaries. Three different transfer functions are shown: images 5.2(a) to 5.2(d) use a transfer function called "#1"; images 5.3(c) and 5.3(d) use a transfer function called "#2"; and images 5.3(e) and 5.3(f) use a transfer function called "#3".

The fixed-resolution images, 5.2(c), (e), and (g), each require approximately 25 second to render, while the multiresolution images, 5.2(d), 5.3(f), and (h), each require about 2.5 seconds to render. Statistics concerning rendering times for this data set are given in Table 5.2.

We were not able to render a static texture map version of the *Raleigh-Taylor instability* data set. However, the Equine Metacarpus data shows the timing issues. We should also note that the *Raleigh-Taylor instability* data set "Tile Time" also includes a reasonable amount of time lost to swapping. However, the render time includes the time to change the transfer function. If we extrapolate the time (from the Equine Metacarpus Static vs. Index "Tile Time") to regenerate the tiles for a static texture map version, the "Tile Time" should be about 5245 seconds (about 88 minutes); this is clearly not a reasonable amount of time if one wishes to experiment with the transfer function.

5.5 Conclusions

We have described a new method for building and rendering a multiresolution texture hierarchy approximation for very large data sets where the transfer function can be modified interactively. The approach utilizes a "bricking" strategy, where the displayed bricks are selected from an octree representation, and index texture maps, where the transfer function applied with a color look-up table. The color look-up table is significantly faster than prior methods and provides a better basis for exploring very large data sets. Despite the fact that our overall system is limited by the amount of available texture memory, the algorithm produces very good results, and we expect that this approach will have a major impact on the huge volumetric data sets that are currently encountered in numerous applications. Future work involves removing the artifacts between tiles of different resolution and rendering pre-segmented biological data sets.

Chapter 6

Multiresolution Cutting-Planes

We present a multiresolution technique for interactive texture-based rendering of arbitrarily oriented cutting-planes for very large data sets. This method uses an adaptive scheme that renders the data along a cutting-plane at different resolutions: higher resolution near the point-of-interest and lower resolution away from the point-of-interest. The algorithm is an extension of work in chapters 4 and 5.

6.1 Introduction

In this Chapter, we combine hardware-assisted texture mapping and multiresolution methods for rendering cutting-planes of large volumetric data sets. The general idea is to assign priorities to different regions of the volume and to render the high-priority regions with highest accuracy, while lower-priority regions are rendered with progressively less accuracy, and progressively faster.

We use an octree to decompose texture space producing several coarser levels of the original data set. Each level is associated with a level in the octree and each level is half the resolution of the next level. The leaf nodes are associated with the original resolution, and the root node is associated with the coarsest resolution. Interior nodes are created by subsampling the eight child nodes. Each node contains two texture tiles, called *high* and *low*. The *high* tile stores the node's copy of the data; the *low* tile stores portion of the parent's *high* tile that covers the same area as the node.

Rendering a cutting-plane involves traversing the octree and applying a selection filter to each node, building a selected node tree. Three results are possible: (1) the node (and its children) are skipped entirely; (2) the node is skipped, but its children are visited; or (3) the node is rendered and the children are skipped. The selected node tree forms an incomplete octree with the leaves being the nodes selected for rendering. The second step is to balance the selected node tree: all adjacent nodes must differ by no more than one level of resolution. The final step is to render each node, blending the *high* and *low* tiles when the node is adjacent to a lower-resolution node.

This technique reduces the amount of data accessed to produce a rendering. This is important in "data exploration" or visual steering applications, where a user does not know the point-of-interest or would just like to browse the data. Another application is progressive visualization: often, a data set is too large to be placed on one computer system, and portions are distributed across a network of machines. It is not always practical to wait for all systems to finish rendering. With our technique, an initial approximation is first rendered. As higher-resolution data is received, a higher-quality approximation is rendered. This continues until all the data is received or the user changes viewing parameters.

Section 6.2 discusses construction of the texture hierarchy, and Section 6.3 covers how to process and render the texture hierarchy. Section 6.4 shows results for two data

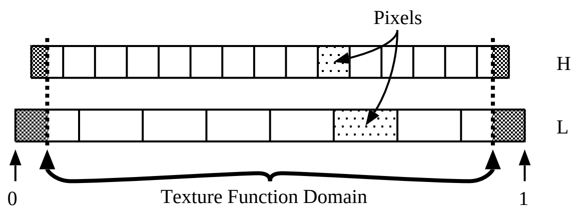


Figure 6.1: A node with one-dimensional tiles, *high(H)* and *low(L)*.

sets and provides performance results. Conclusions and future work are presented in Section 6.5.

6.2 Generating The Texture Hierarchy

6.2.1 ***High*** and ***Low*** Texture Tiles

In hardware texturing, linear interpolation is used to interpolate the values at the centers of adjacent texels. To allow for blending within a node, each node contains two texture map tiles (Figure 6.1). The *high* tile is the normal data associated with that node. The *low* tile is that part of the parent's *high* tile that is covered by the child node. The size ratio *high* to *low* is defined as $|high| = |low| * 2 - 1$. Thus, one of the tiles must have odd size. If the size of a texture tile must be a power of two, then this relationship will incur some memory overhead. Our system uses a power-of-two size for the *low* tile, and the size for the *high* tile is calculated accordingly.

6.2.2 The Multiresolution Texture Hierarchy

Figure 6.2 shows a texture hierarchy consisting of two levels. The higher-resolution level is denoted as level A, with nodes A^0 and A^1, and the lower-resolution level as B. The image represented by A can be approximated by B. The *high* and *low* tiles in B

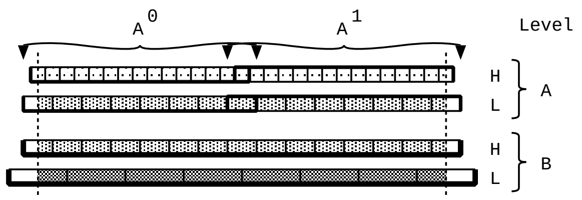


Figure 6.2: A texture hierarchy of two levels.

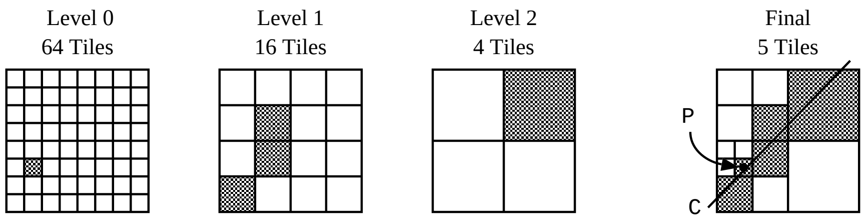


Figure 6.3: Selecting a set of tiles from a 2D hierarchy of four levels (level three not shown).

are the same size as the *high* and *low* tiles in A^0 or A^1, and half the total size of the *high* and *low* tiles in A. We note that the natural relationship for two textures whose resolutions differ by a factor of two is using texel-center alignment. In the binary tree arrangement defined by this one-dimensional texture, B is the parent of A^0 and A^1. Also, note the correspondence between the *low* tile of the children to the *high* tile of the parent.

Figure 6.3 shows a two-dimensional quadtree example. The original texture, level 0, contains 64 nodes. The dark regions show the portion of the level used in rendering the cutting-plane. Nodes are selected when the distance from the center of the node to the point p is greater than the diagonal length of the node, and when the node intersects the cutting-plane c. The selected nodes are shaded. The original texture, divided into 64 nodes, requires 64 time units to transfer. The multiresolution rendering uses five nodes, requiring five time units which implies a speed-up factor of about 13.

This technique extends to three-dimensional textures. Approximations are generated by performing low-pass transforms on the textures. The amount of memory "wasted" over the prior technique [LJH99] is the storage of the *low* tile with each node; since each *low* tile is $\frac{1}{8}$ the size of the *high* tile, the additional memory overhead is $\frac{1}{8}$. For the above example, if all nodes rendered required some portion of both the *high* and *low* tiles, the five nodes would contain $5 \times (1 + \frac{1}{8}) = 5.625$ (*high*) tiles, so the speed-up would only be about 11.37. However, only those nodes the require rendering of both *high* and *low* tiles would require transmitting this additional data.

6.3 Rendering

The rendering phase is divided into the following steps: (1) selecting nodes to be rendered and building the selected node tree; (2) balancing the selected node tree; (3) computing the blending ratios; and (4) rendering the nodes.

6.3.1 Selecting Nodes

The first rendering step determines which nodes will be rendered. The general filtering logic starts at the root node and performs a depth-first traversal of the octree. Our primary selection filter is based on one of these two criteria:

- **Cutting-Plane.** This filter selects a node when it intersects the cutting-plane.
- **Multiresolution Cutting-Plane.** This filter selects a node when it intersects the cutting-plane and the distance from the node center to the point-of-interest (on the cutting-plane) is smaller than the diagonal length of the node. We note

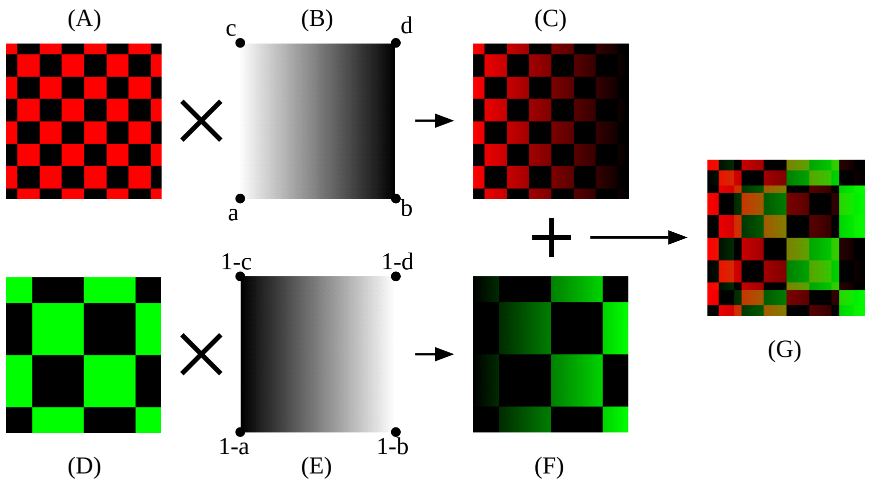


Figure 6.4: Blending red and green checker board patterns.

that this selection criteria is arbitrary and is chosen just to show the essential process.

6.3.2 Blending

Texturing is performed by modulating the color of the proxy geometry by the texture; the color is white and constant across a polygon. However, to blend two images, we can change the polygon color to implement bilinear filtering. In Figure 6.4, image (G) is created by performing a per-pixel affine combination of images (A) and (D). Image (B), with ratios of $a = c = 1$ and $b = d = 0$, multiplies (A) and produces (C). Image (E) multiplies (D) and produces (F). Images (B) and (E) sum to unity. Adding (C) and (F) produces (G): a transition from red checks on the left to green checks on the right. We obtain a smooth transition, provided (A) and (D) are two different resolutions of the same image. We note that our current hardware does not allow this to be performed in one-pass, thus we implement this as a two-pass scheme. However, we believe that the next generation of graphics hardware will include functionality to

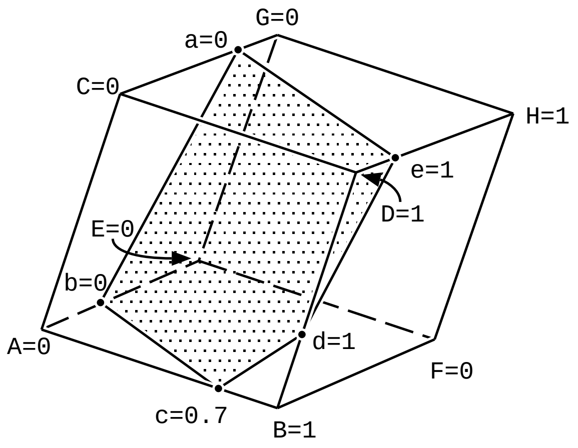


Figure 6.5: Cutting-plane clipped against the faces of an intersecting node.

allow us to implement this in a one-pass scheme.

6.3.3 Neighborhoods and Balancing

The blending algorithm described in section 6.3.2 requires that all selected nodes in a 26-neighborhood (across node faces, edges, and corners) have resolutions that differ by at most one level in the octree. Blending within a node can only blend between two texture resolutions: the high-resolution texture is blended into the low-resolution texture. Nodes have two textures tiles, *high* and *low*, so that a pair of nodes that differ by one level in the tree can be blended. Those that differ by two or more levels do not share any textures and cannot be blended.

After balancing the tree, we examine the neighbors of all selected nodes. The nodes adjacent to a node of lower-resolution must be blended such that the textures match. Considering a corner of a given node, whenever one of the seven adjacent nodes exists and has lower-resolution, that corner must blend to the *low* tile; otherwise, it must use the *high* tile.

	Mandrill (Fig. 6.6)		Visible Female (Fig. 6.7)	
Data set resolution	$256^2 * \text{RGB}$ (2D)		$500^2 * 250 * \text{RGBA}$ (3D)	
Data set size	192K		238MB	
Tile resolution (high/low)	$15^2/8^2$		$32^3/16^3$	
Tile size (high/low)	1024/256 bytes		128K/16K bytes	
Level 0 nodes	324		2601	
Rendered nodes: fixed/MR	324	41	443	50
Bytes transmitted	405K	51K	56MB	7MB
Rendering time	-	-	2.0 sec.	0.37 sec.

Table 6.1: Timing results for *Mandrill* and *Visible Female* data sets.

Figure 6.5 shows a cutting-plane clipped to an intersecting node. A to H are the blend ratios associated with the node: corners B, E, and H are adjacent to lower-resolution nodes, so that the blend ratio is one; the other corners have a blend ratio of zero, selecting the *low* and *high* tile of the node, respectively. The values a to e are the blend ratios associated with the clipped cutting-planes vertices. Ratios on an edge are linear combinations of the ratios at the ends of that edge, and are proportional to the position of the point along the edge.

For rendering, we first define the RGB value for each clipped cutting-plane vertex to the ratio (a to e in Figure 6.5), download the *low* texture tile, and draw the polygon. The color values will be interpolated across the polygon, multiplying the texture and producing the first weighted image. Next, we download the *high* texture tile, define the RGB value for each clipped cutting-plane vertex to one minus the ratio, and draw the polygon, producing the second weighted image. Finally, by adding the first and second images, we produce the desired blended result.

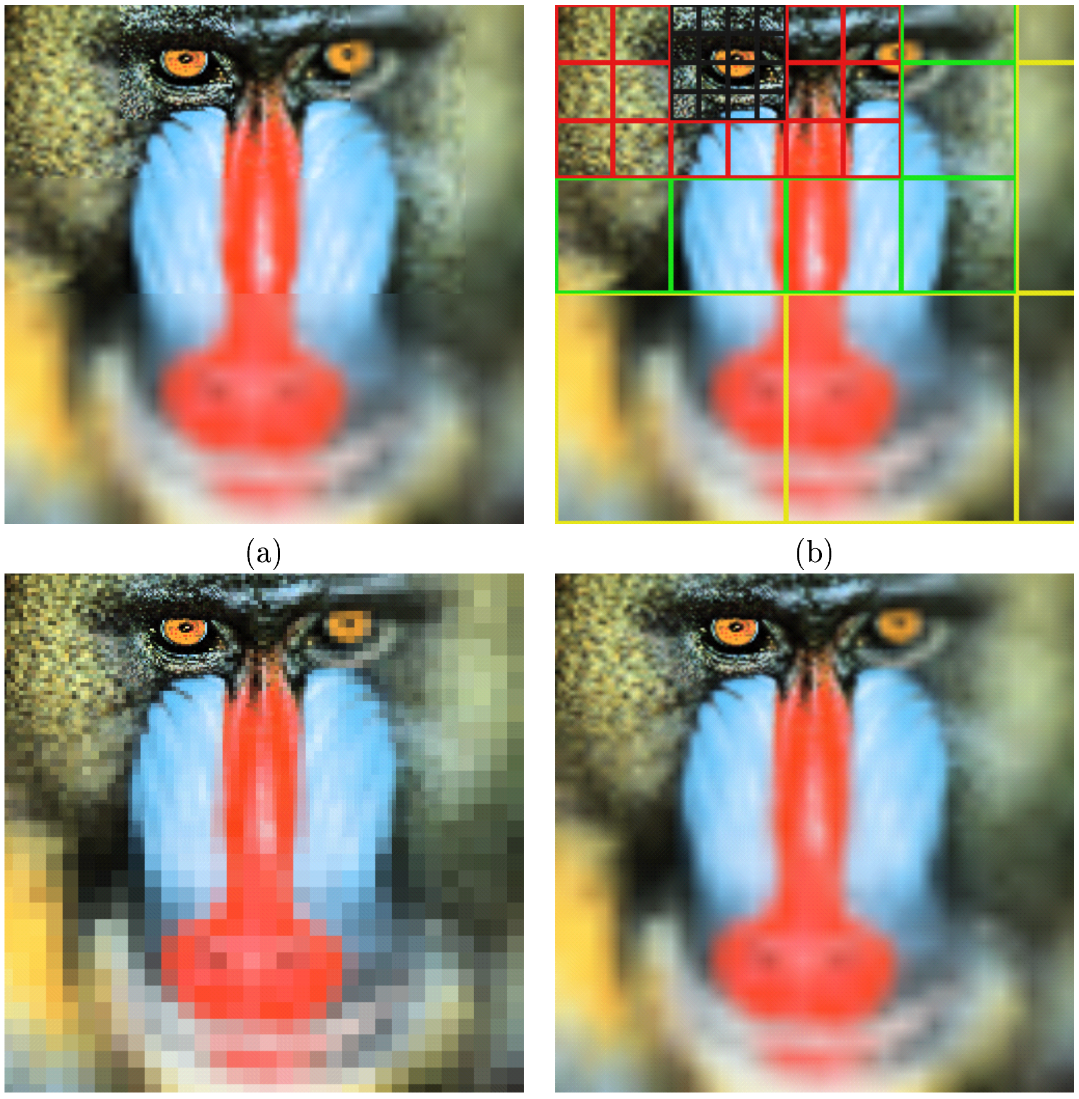

(a) (b) (c) (d)

Figure 6.6: Multiresolution Mandrill: (a) without blending; (b) node boundaries high-lighted; (c) blended nearest-neighbor; and (d) blended, bilinear filtering.

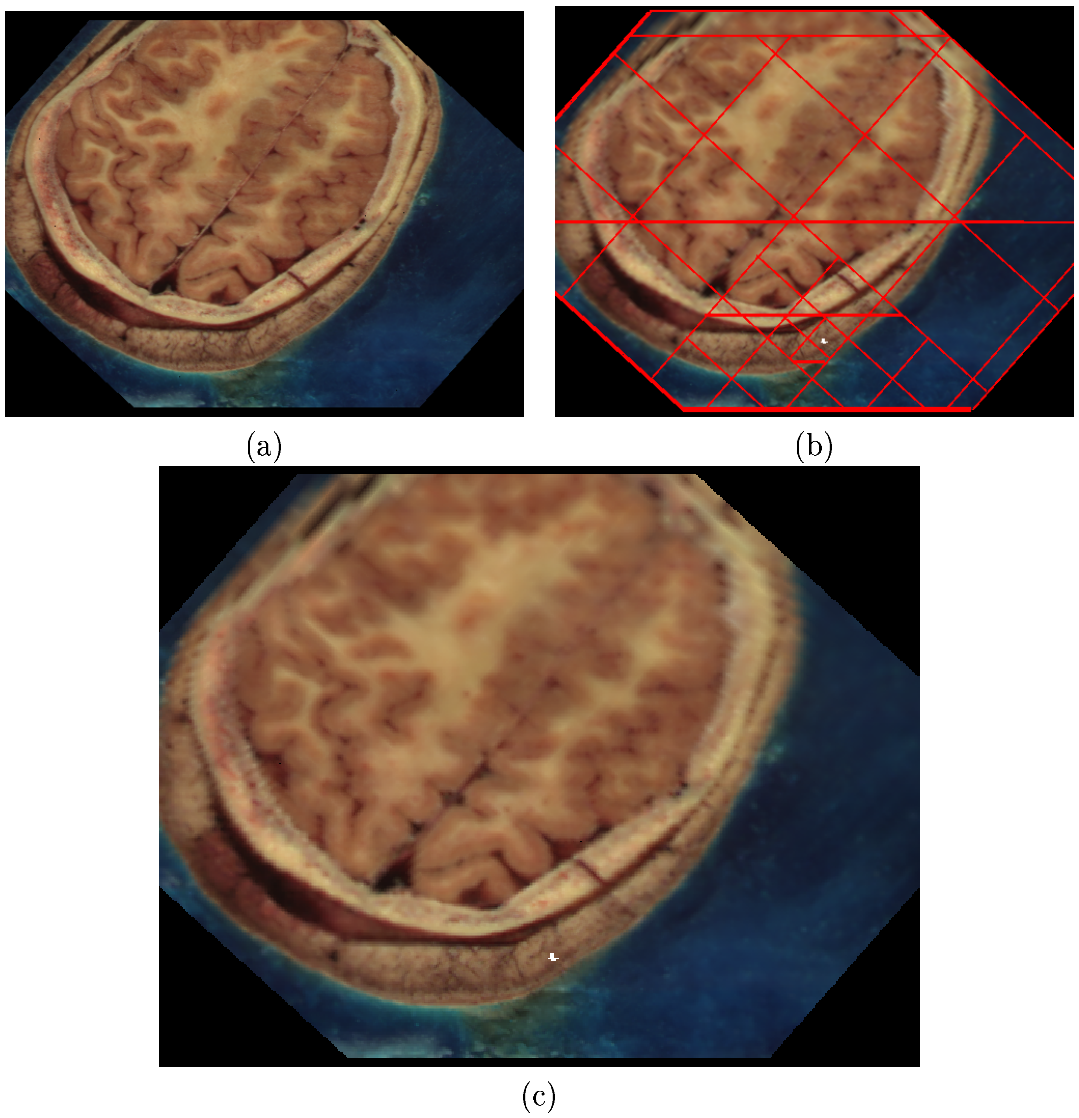


Figure 6.7: Multiresolution cutting-plane of the Visible Female data set: (a) fixed resolution; (b) blending with node boundaries high-lighted; and (c) MR, blending.

6.4 Results

We have implemented the algorithm and applied it to parts of the Visual Female data set. The data sets were rendered on an SGI Onyx2 computer system with 512MB of main memory and 16MB of texture memory, using a single 195MHz R10K processor.

For comparison, Figure 6.6 shows a multiresolution image of a Mandrill. This image is used to point out the artifacts when not blending across different levels of resolution. Image 6.6(b) shows the nodes and node boundaries: the resolution is shown by the node's boundary color, from highest to lowest resolution: black, red, green, and yellow; notice the artifacts at the node boundaries in image 6.6(a). Image 6.6(c) shows the blending result, with nearest-neighbor filtering. One notices that the pixel sizes blend smoothly across the nodes. Image 6.6(d) shows the final result. One notices how the image is free of the boundary artifacts and smoothly blends high-resolution nodes to low-resolution nodes.

Figure 6.7 shows a multiresolution view of the Visible Female data set. The 443 nodes of the Visible Female represent the highest-resolution nodes that intersect the cutting-plane (the other 2158 are never considered). The performance results shown in Table 6.1 are for a single frame, at 20 frames per second. The 1.1GB/sec required for the non-multiresolution approach exceeds the SGI InfiniteReality Engine's maximum transfer rate for textures of 320MB/sec by a factor of about 3.5, while the 140MB for the multiresolution approach has capacity to spare.

6.5 Conclusions

We have presented an algorithm for interactive rendering of multiresolution cutting-planes. We use hardware-based texturing, multiresolution techniques, and image blending to render a smooth approximation of a cutting-plane. We have shown that our algorithm can produce a reasonable approximation while using less data. Despite the fact that our overall system is limited by the amount of available texture memory, the algorithm produces very good results, and we expect that this approach will have a major impact on the exploration of massive volumetric data sets that are currently generated in numerous applications.

Future work includes error analysis, implementing this technique in our multiresolution volume visualization system, extend it to visualizing vector fields, and developing more sophisticated selection criteria. The selection criteria discussed in this chapter are overly simple – a much more sophisticated mechanism is required for progressive rendering, e.g., for web applications over a slow modem.

Chapter 7

Efficient Error Calculation

Multiresolution volume visualization is necessary to enable interactive rendering of large data sets. Interactive manipulation of a transfer function is necessary for proper exploration of a data set. However, multiresolution techniques require assessing the accuracy of the resulting images, and re-computing the error after each change in a transfer function is very expensive. We extend our existing multiresolution volume visualization method (see [LJH99], [LHJ00], and [LDHJ00]) by introducing a method for accelerating error calculations for multiresolution volume approximations. Computing the error for an approximation requires adding individual error terms; in this case, one error value must be computed once for each original voxel and its corresponding approximating voxel. For byte data, we observe that the set of error pairs can be quite large, yet the set of *unique* error-pairs is small. Instead of evaluating the error function for each original voxel, we construct a table of the unique combinations and the number of their occurrences. To evaluate the error, we add the products of the error function for each unique error pair and how often this error pair occurs. This dramatically reduces the amount of computation time involved and allows us to re-compute the error associated with a new transfer function quickly.

7.1 Introduction

When rendering images from approximations, it is in most applications necessary to know how close the generated image is to the original data. For multiresolution volume visualization, it is not possible to compare the images generated from original data to all possible images generated from approximations. The reason for using the approximations is to substantially reduce the amount of time required to render the data. If we assume that there is a reasonable amount of correlation between the data and the resulting imagery, we can compute an error value between the approximations and the original data (in 3D object space), and use that value to estimate the error in the 2D imagery.

However, the time required to evaluate the error over an entire data hierarchy can be very expensive. This is especially important when it is necessary for a user to interactively modify a transfer function: each change in the transfer function requires re-computing the error for the entire hierarchy.

We introduce a solution based on the observation that, for many data sets, the range size of (integer) data sets is often many orders smaller that the domain size (physical extension) – and that, instead of evaluating an error function for each original voxel, it suffices to count the frequencies of unique pairs of error terms. In the case of 8-bit integer (byte) data, there are only 256^2 combinations (each term is a single byte, or 256 potentially different values). To compute the error, instead of adding individual error terms, we add the products of the error (for a unique pair of error terms) by the number of occurrences of that unique pair.

For example, a typical 512^3 voxel data set, with one byte per voxel, contains 2^{27} bytes. To compute the error, a naive method would evaluate an error function for each of the original 2^{27} voxels. However, there are only 256^2 unique pairs of error terms. Thus, to compute the error, we evaluate the error function for each unique pair of error terms, or 2^{16} times – which is 2^{11} times faster than the naive method. This algorithm requires that the entire data set be examined for unique pairs of error terms - this is a preprocessing cost that can be performed off-line.

This work is a direct extension of earlier work [LJH99], [LHJ00], and [LDHJ00]. We first review the generation of a volume hierarchy in section 7.2, discuss the error criteria we use in section 7.3, cover some basic optimizations of the general approach in section 7.4, show the performance in section 7.5, and discuss directions for future work in section 7.6.

7.2 Generating the Hierarchy

First, we review certain aspects of our multiresolution data representation and how it influences the decisions on how to evaluate and store error. The underlying assumption is that data sets are too large to fit into texture memory, and are often too large even to fit into main memory. Given a volumetric data set, we produce a hierarchy of approximations. Each level in the approximation hierarchy is half the size of the next level. Each level is broken into constant-sized tiles – tiles that are small enough to fit in their entirety in texture memory, see Section 4.3.

Figure 7.1 illustrates a 1D multiresolution hierarchy of three levels and seven nodes of a one-dimensional function with tile size of 15 texels. Each node in the tree contains one tile. Level 0 is the original data and is broken into four tiles. Level 1

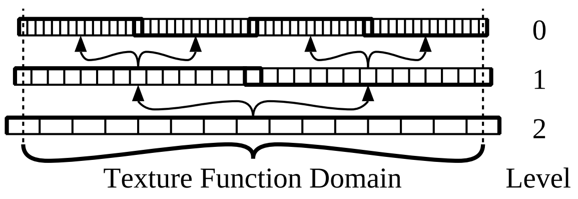


Figure 7.1: Texture Hierarchy of Three Levels.

is the first approximation and is broken into two tiles; and Level 2 is the final and coarsest approximation (of Level 0) and contains just one tile. The coarsest tile is contained in the root of the tree; the arrows show the parent-child relationship of the four nodes. This tree forms a binary tree (and, in three-dimensions, an octree), and a Level i texel approximates $(2^i + 1)^d$ level 0 texels, where $d = 1$ for a binary tree, see Figure 7.1, and $d = 3$ for an octree.

Our multiresolution volume visualization system uses hardware-accelerated 3D texturing for rendering a volume, and represents that volume with index textures. A transfer function is applied by first building a look-up table, transferring it into the graphics system, and then transferring the texture tile. The translation of index values to display values (luminance or color) is performed in hardware.

7.3 Error Calculation

We calculate error on a per-node basis: when a node meets the error criterion, it is rendered; otherwise, its child (higher-resolution) nodes are considered for rendering. We currently assume a piece wise constant function implied by a set of voxels, but use trilinear interpolation for the texture. This simplifies the error calculation.

We use two error norms: the L-infinity and root-mean-square (RMS) norms. Given two sets of function values, $\{f_i\}$ and $\{g_i\}$, $i = 0, \ldots, n-1$, the L-infinity error norm is defined as $l_\infty = max_i \{|x_i|\}$, and the RMS is defined as $E_{rms} = \sqrt{\frac{1}{n}\sum_i (x_i^2)}$, where $x_i = T[f_i] - T[g_i]$ and $T[x]$ is a transfer function. For the purposes of this discussion, we will assume that $T[x]$ is a simple scalar function, mapping density to grey-scale luminance; the issue of error in color space is beyond the scope of this discussion. We evaluate the error function once for each Level-0 voxel.

A data set of size 512^3 contains 2^{27} voxels, with the same number of pairs of error terms for each level of the hierarchy. However, when using byte data, we observe that, though there are 2^{27} pairs of values, there are only 2^{16} (256^2) unique pairs of error values. This means, on average, that each unique pair is evaluated 2^{11} times. Our solution is to construct a table Q, with elements $Q_{a,b}$, that stores the number of occurrences for each (f_i, g_i) pair, where a and b are the table indices corresponding to f_i and g_i, respectively. Thus

$$Q_{a,b} = \sum_i \begin{pmatrix} 1 & \text{if } f_i = a \text{ and } \& g_i = b \\ 0 & \text{otherwise} \end{pmatrix}.$$

This table is created only once, when the data is loaded. Each internal node of the octree, i.e., each approximating node, contains a Q table that counts the unique error terms that the node "covers" of the original data.

To calculate the L-infinity error value for a node, we search the for the largest value, with the requirement that this value corresponds to a real pair of values:

$$l_\infty = max_{a,b|Q_{a,b} \neq 0} \{|T[a] - T[b]|\}.$$

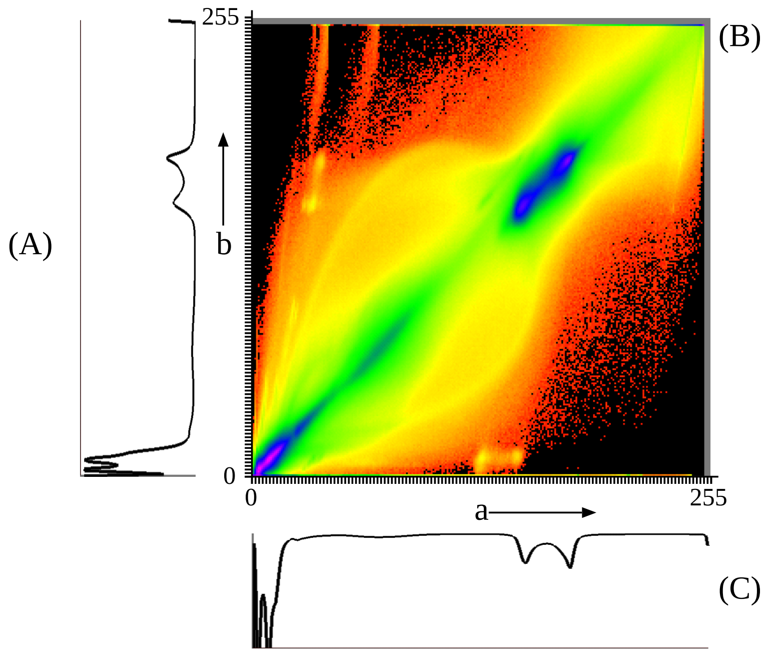


Figure 7.2: The Visible Female CT data set. Image (B) in the upper-right corner shows the union of all $Q_{a,b}$ tables for the Level-1 approximation. Graph (A), on the left, shows the Level-0 histogram. Graph (C), on the bottom, shows the Level-1 histogram.

We compute the RMS error for a node as

$$E_{rms} = \sqrt{\frac{1}{n} \sum_{a,b} (T[a] - T[b])^2 \times Q_{a,b}},$$

where $n = \sum_{a,b} (Q_{a,b})$. For a set of t nodes, we compute the value

$$E_{rms} = \sqrt{\frac{1}{N} \sum_{t} \left(\sum_{a,b} (T[a] - T[b])^2 \times Q_{a,b} \right)},$$

where $N = \sum_t \left(\sum_{a,b} Q_{a,b} \right)$.

Image 7.2(B) in the upper, right-hand corner of Figure 7.2 shows a graphical representation of the union of all Q tables for the Level-1 approximation of the Visible Female CT data set. This image consists of 256×256 pixels, with a on the horizontal axis and b on the vertical axis; each pixel corresponds to a $Q_{a,b}$ element. The colors are assigned by normalizing the logarithm of the number of occurrences, then linearly mapping these into a rainbow color sequence, where zero maps to red and one maps to violet: $pixel_{a,b} = RainbowColorMap\left[ln\left(Q_{a,b}\right)/ln\left(Max_{a,b}\left\{Q_{a,b}\right\}\right)\right]$. Graphs 7.2(A) and 7.2(C) show the Level-0 and Level-1 histograms, respectively.

Figure 7.2 shows that the approximation is generally good: most of the error terms are along the diagonal. The "lines" in the upper-left corner of the image may correspond to high gradients present in the data set. The Visible Female CT data set was produced such that sections of the body fill a 512^2 image: these different sections are scanned with different spatial scales. We have not accounted for these different spatial scales in our version of the data set, thus there are several significant discontinuities in data values in the data set.

7.4 Optimizations

Evaluating a table is still fairly expensive when interactive performance is required. We currently use three methods to reduce the time needed to evaluate the error.

First, one observes that the order of the error term calculations, $x_i = |f_i - g_i| = |g_i - f_i|$ and $x_i = (f_i - g_i)^2 = (g_i - f_i)^2$, does not matter. If we order the indices in $Q_{a,b}$ such that $a \leq b$, or $a = min(f_i, g_i)$ and $b = max(f_i, g_i)$, we obtain a triangular matrix that has slightly more than half the terms of a full matrix (32896 vs. 65536); this roughly halves the evaluation time and storage requirements.

Second, one observes that, typically, there is a strong degree of correlation between approximation and original data. This means that the values of $Q_{a,b}$ are large when a is close to b (i.e., near the diagonal); and small, often zero, when $a \ll b$, (i.e., far away from the diagonal). We have observed that the number of non-zero entries in a Q table decreases (i.e., the table is becoming more sparse) for nodes closer to the leaves of the octree. This happens since the nodes closer to the leaf nodes correspond to high-resolution approximations and thus a better approximation; the correlation is strong, and the non-zero values in the Q table cluster close to the diagonal. Also, the nodes closer to the leaves cover a progressively smaller section of the domain. We perform a column-major scan, i.e., traverse the table first by column, then by row. Thus, the Q tables become sparse, and we can terminate the checking of elements if we remember the last non-zero entry for a row. We maintain a table, L_{row}, that contains the index of the last non-zero value for a row. It may even be possible to perform a run-length encoding on a row to skip over regions of zero entries. Figure 7.2 shows that there are large, interior regions with zero entries.

Third, one can use "lazy evaluation" of the error. When we re-calculate the error for all nodes in a hierarchy and few of the nodes are rendered, much of the error evaluation is wasted. Thus, for each new transfer function, we only re-calculate the error value of those nodes that are being considered for rendering (see [LHJ00]).

7.5 Results

Performance results of static vs. index texture transfer function and error time were obtained on an SGI Origin2000 with 10GB of memory, using one (of 16) 195MHz R10K processor. However, images were produced on an SGI Onyx2 Infinite Reality with 512MB or memory, using one (of four) 195MHz R10K processor. This was

Image	Lvl.	Voxels	Nodes	Nodes	Mem.	Time	Error	
			in Lvl.	Rndrd.	(MB)	(sec.)	l_∞	E_{rms}
7.3a)	0	$512^2 \times 1734$ =454M	2268	1560	390	83.4	0.000	0.00000
7.3b)	1	$257^2 \times 868$ =57M	350	263	65	8.73	0.305	0.00678
7.3c)	2	$129^2 \times 435$ 7.2M	63	49	13	2.38	0.305	0.00917
7.3d)	3	$65^2 \times 218$ 921K	16	16	4	0.563	0.305	0.01059

Table 7.1: Visible Female CT data set multiresolution statistics. Abbreviations: "Lvl." for level, "Rndrd" for rendered, and "Mem" for memory.

done for the following reasons: we are interested in the time required to process the hierarchy, free of memory limitations. The logical memory used during a visualization is 2.4GB (0.5G of this is mapped), and thrashing completely dominates (by a factor of 10 or more) the transfer function and error calculation times. The SGI Origin2000 does not have a graphics subsystem, while the SGI Onyx2 does.

We have used the Visible Female CT data set, consisting of $512^2 \times 1734$ voxels in our experiments. Figure 7.3 shows different-resolution images of this data set: 7.3(a) shows the original data, and 7.3(b) to 7.3(d) are progressively 1/2 linear (1/8 total) size. Table 7.1 shows performance statistics for four (of six) levels of the hierarchy. The "Voxels" column shows the total size of the approximation in voxels (the upper entry is the linear dimensions and the lower entry is the total), and the next column shows the total number of nodes associated with that level. The "Nodes Rendered" column shows how many nodes where used to render that level – the number of nodes rendered is actually less than one would expect: many regions of the data set are constant, so there is no error when approximating these regions. Each tile contains 64^3 bytes = 256K. When transferring n tiles, the total memory transferred

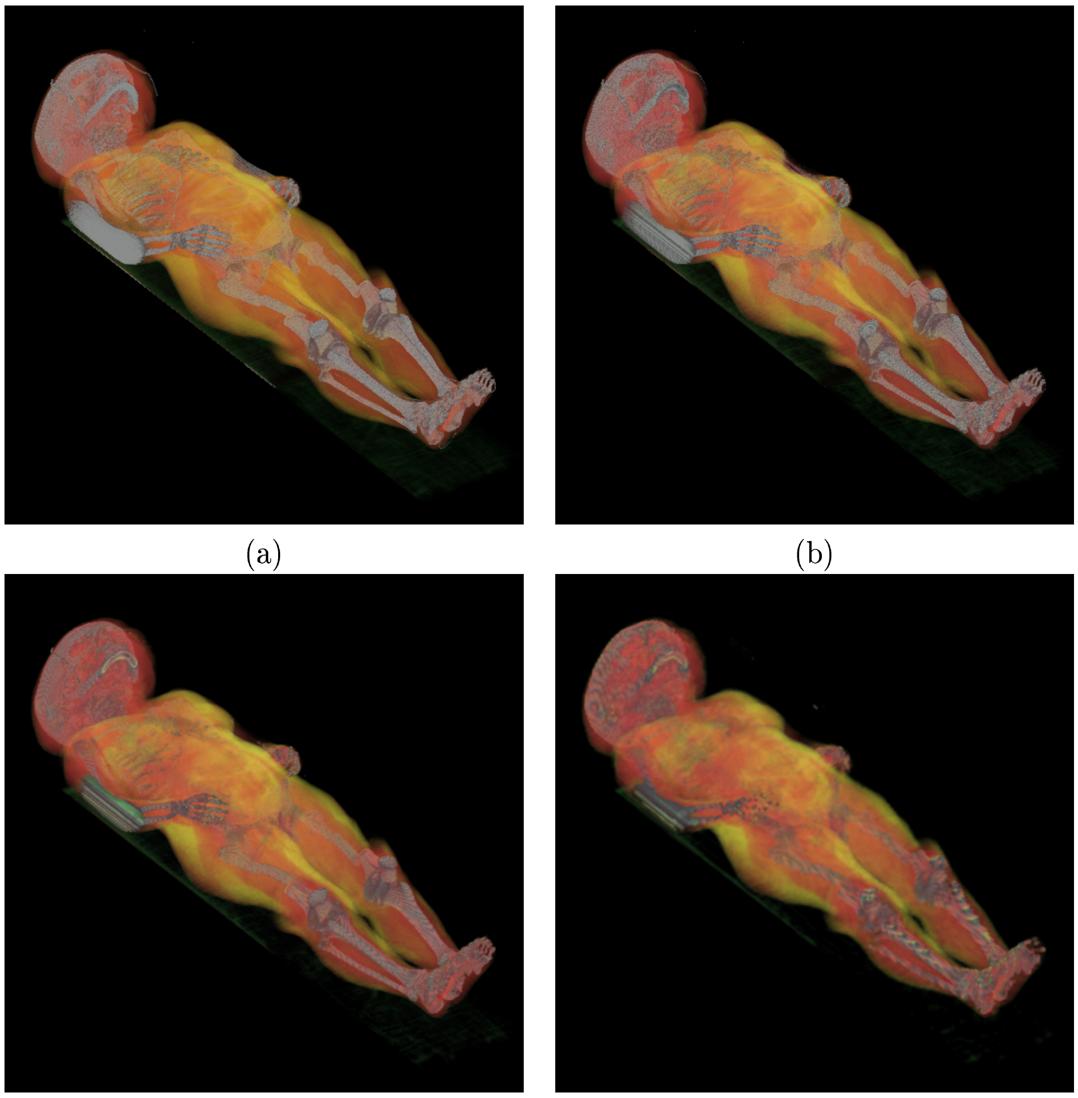


Figure 7.3: Visible Female CT data set rendered at four different resolutions (see Table 7.1). The transfer function shows bones in white, fat in yellow, muscle in red, and internal organs in green. Note the different spatial scales for different sections of the data set.

Step	Index Texture
Compute Texture Hierarchy	1 min 46 sec
Compute Error Table	5 min 35 sec
Calculate Error	1.23 sec

Table 7.2: Performance statistics for Visible Female CT data set.

to the graphics subsystem for n tiles is 256K$\times n$, shown in the "Memory" column. The "Time" column shows the time, in seconds, required to produce the rendering for that level; and the l_∞ and E_{rms} columns show the error values associated with that rendering.

Table 7.2 shows the times for various stages of our system – we note that the "Compute Texture Hierarchy" and "Compute Error Tables" stages are only performed once for a data set, and only the "Calculate Error" step is performed for each new transfer function. The time for "Calculate Error" is the times needed to to re-compute the error for all 432 approximating nodes (with Q tables). The time per node is approximately 0.0028 seconds – and when coupled with a lazy evaluation scheme, error calculation is insignificant in terms of the other parts of the rendering pipeline (see "Time" column in Table 7.1).

7.6 Conclusions and Future Work

We believe that there are several significant directions to continue this work. The first is to extend this technique to color images or more general vector data hierarchies. The second is to consider data ranges other than bytes: data sets with 12 and 16 bits per voxel are common. However, a table would contain 4096^2 (2^{24}) or 65536^2 (2^{32}) entries. These tables would be larger than the actual volume data per node, and possibly require more time for evaluation. Could some quantizing approach work? Since these nodes should have a high degree of correlation, will these tables be sufficiently sparse to compress? A third direction is to apply our technique to time-varying data. The error between nodes in different time steps would be expressed in the same manner and could be encoded in a table.

Chapter 8

Conclusions and Future Work

In this dissertation, two major research directions have been introduced and discussed.

In part one, we showed that the tricubic B-spline provides an effective method for producing smoother (C^2) isosurface renderings. However, the resample time is long and thus poses a significant hurdle to achieve interactivity. We propose two different methods that might produce the same results. The first technique is to use hardware-based texturing method for rendering isosurfaces and the proposed OpenGL cubic interpolation of texture values. The second technique is to extend methods from subdivision surfaces, where a surface is subdivided until it is sufficiently simple or flat. We use the interpolating uniform cubic B-spline. Subdivision methods for one and two dimensions are well understood, and are easily extended to three dimensions.

In part two, we developed the idea of multiresolution techniques for hardware-based texture-based volume visualization and arbitrarily-oriented cutting planes. These techniques will be significantly improved over the next few years as more sophisticated graphics hardware will become available for PC platforms. Commodity PC graphics cards have surpassed dedicated graphics systems (SGI InfiniteReality, for example) in terms of speed of basic operations, and will surpass general workstation functionality with the next generation of PC graphics cards. For example, two recent introductions, ATI's Radeon and FireGL's FireGL graphics cards support volumetric textures. This means that all of the techniques introduced and discussed in part two of this thesis, thus far only executable on SGI's InfiniteReality engine, can be implemented on PCs. At the time of this writing, these cards were available for $400 and $2000, respectively. The SGI Onyx2 InfiniteReality, machine upon which most of this research was performed, costs an estimated $125K, while a similarly equipped PC with a FireGL card costs an estimated $4K.

We believe feel that extending this work to progressive rendering and steering will have the most impact in the next years. One of the original motivations for this research was the fact that data set sizes are continuing to increase. At the same time, fewer manufacturers are producing large-scale rendering systems, and centralized data repositories require users to connect remotely to the data. Parallel computers still offer a solution, but not for the general user for routine rendering. Thus, data exploration must be done with desk-side workstations, both to fetch the data and to render it. We believe that it is timely to investigate client-server mechanisms for rendering.

We propose a rendering environment where users, using commodity graphics equipped PCs, connect remotely to a data repository to explore a large data set. Super computers, connected locally to the data repository, can render higher quality images and ship these images back to the user. People are already quite comfortable with this relationship: they use word-processors but prefer the printed results. For example, a medical doctor can visit a patient at home and can review or annotate a recent MRI or CT scan of that patient. Later, the doctor can request a higher (or full) resolution image of the same viewpoint. Similarly, an emergency room doctor should be able to see basic patient data when seeing the patient - not waiting for the full CT scan to arrive from another hospital.

Acknowledgements

I would like to thank the following people for their time and input: Martin Bertram, Falko Küster, Greg Schussman, Björn Heckel, Ben Jordan, Liz Jurrus, Fritz Barnes, Joerg Meyer, Gerik Scheuerman, Gunther Weber, Oliver Kreylos, Mark Duchaineau, Randy Frank, Sam Uselton, Ken Joy, Bernd Hamann, Nelson Max, Kwan-Liu Ma, and Hans Hagen.

This work was supported primarily by Lawrence Livermore National Laboratory under contract B504965; also by the National Science Foundation under contract ACI 9624034 (CAREER Award) and through the National Partnership for Advanced Computational Infrastructure (NPACI); the Office of Naval Research under contract N00014-97-1-0222; the Army Research Office under contract ARO 36598-MA-RIP; the NASA Ames Research Center through an NRA award under contract NAG2-1216; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; and the North Atlantic Treaty Organization (NATO) under contract CRG.971628 awarded to the University of California, Davis. We also acknowledge the support of ALSTOM Schilling Robotics, Chevron, Silicon Graphics, Inc. and ST Microelectronics, Inc.

Bibliography

[Ake93] Kurt Akeley. RealityEngine Graphics. In Lynn Valastyan and Laura Walsh, editors, *Proceedings of Siggraph 93*, volume 27, pages 109–116. ACM, August 1993.

[And95] Andrew S. Glassner. *Principles of Digital Image Sythesis.* Moran Kaufmann Publishers, Inc., San Francisco, Ca, 1995.

[AR87] H. Anton and C. Rorres. *Elementary Linear Algebra with Applications.* John Wiley, New York, NY, 1987.

[BFK84] Wolfgang Boehm, Gerald Farin, and Jürgen Kahmann. A survey of curve and surface methods in CAGD. *Computer Aided Geometric Design*, 1(1):1–60, 1984.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, October 1994. ISBN 0-89791-741-3.

[CLL$^+$88] Harvey E. Cline, William E. Lorensen, Sigwalt Ludke, Carl R. Crawford, and Bruce C. Teeter. Two algorithms for the reconstruction of surfaces from tomograms. *Medical Physics*, 15(3):320–327, June 1988.

[CN94] Timothy J. Cullip and Ulrich Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical Report TR93-027, Department of Computer Science, University of North Carolina - Chapel Hill, May 1 1994.

[CR74] Edwin Catmull and Raphael Rom. A Class of Local Interpolating Splines. In Robert E. Barnhill and Richard F. Riesenfeld, editors, *Computer Aided Geometric Design*, pages 317–326. Academic Press, 1974.

[Eck98] George Eckel. *OpenGL Volumizer Programmer's Guide.* Silicon Graphics Computer Systems, Mountain View, Ca, 1998.

[Far97] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design.* fourth edition, Academic Press, Boston, Ma, 1997.

[FN80] Richard Franke and Greg Nielson. Smooth interpolation of large sets of scattered data. *International Journal for Numerical Methods in Engineering*, 15(11):1691–1704, 1980.

[FN95] Richard Franke and Gregory M. Nielson. Scattered data interpolation and applications: A tutorial and survey. In H. Hagen, H. Mueller, and G.M. Nielson, editors, *Focus on Scientific Visualization*, pages 131–159. Springer-Verlag, New York, 1995.

[GHY98] Robert Grzeszczuk, Chris Henn, and Roni Yagel. *Siggraph '98 "Advanced Geometric Techniques for Ray Casting Volumes" course notes.* ACM, July 1998.

[Gla95] Andrew S. Glassner. *Uniform Sampling and Reconstruction*, pages 331–368. Morgan Kaufmann, San Francisco, CA, 1995.

[Har71] R. L. Hardy. Multiquadric equations of topography and other irregular surfaces. *Journal of Geophysical Research*, 76:1906–1915, 1971.

[Har90] R. L. Hardy. Theory and applications of the multiquadric-biharmonic method: 20 years of discovery 1968–1988. *Computers and Mathematics with Applications*, 19:163–208, 1990.

[HH92] Charles D. Hansen and Paul Hinker. Isosurface Extraction SIMD Architectures. In *Visualization'92*, October 1992.

[HKP$^+$95] Charles D. Hansen, Michael Krogh, James Painter, Guillaume Colin de Verdiere, and Roy Troutman. Binary-Swap Volumetric Rendering on the T3D. In *Cray Users Group Conference*, Denver, Co., March 1995.

[HKW95] Charles D. Hansen, Michael Krogh, and William White. Massively Parallel Visualization: Parallel Rendering. In Bailey, David H., Bjørstad, Petter E., Gilbert, John E., Mascagni, Michael V., Schreiber, Robert S., Simon, Horst D., Torczon, Virginia J. and Layne T. Watson, editors, *Proceedings of the 27th Conference on Parallel Processing for Scientific Computing*, pages 790–795, Philadelphia, PA, USA, February15–17 1995. SIAM Press.

[HTF97] Bernd Hamann, Issac J. Trotts, and Gerald E. Farin. On Approximating Contours of the Piecewise Trilinear Interpolant Using Triangular Rational-Quadratic Bézier Patches. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):215–227, July 1997.

[Jai89] Anil K. Jain. *Fundamentals of Digital Image Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Joy98] Kenneth I. Joy. On-line geometric modeling notes. http://graphics.cs.ucdavis.edu/CAGDNotes/CAGD-Notes.html, 1998.

[Lac95] Philippe Lacroute. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In Stephen N. Spencer, editor, *Proceedings of the 1995 Parallel Renering Symposium*, pages 15–22, New york, 30–31 October 1995. ACM Press.

[Lac96] Philippe Lacroute. Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, September 1996.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In M. C. Stone, editor, *SIGGRAPH '87 Conference Proceedings (Anaheim, CA, July 27–31, 1987)*, volume 21, pages 163–170. Computer Graphics, Volume 21, Number 4, July27–31 1987.

[LDHJ00] Eric C. LaMar, Mark Duchaineau, Bernd Hamann, and Kenneth I. Joy. Multiresolution Techniques for Interactive Texturing-based Rendering of

Arbitrarily Oriented Cutting-Planes. In W. C. de Leeuw and R. van Liere, editors, *Data Visualization 2000*, pages 105–114, Vienna, Austria, January 2000. Proceedings ofthe "Joint EUROGRAPHICS and IEEE TCVG Symposium of Visualization", Springer-Verlag.

[Lev87] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, February 1987.

[Lev90a] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[Lev90b] Marc Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, February 1990.

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):285–288, July 1991.

[LHJ99] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. High-quality rendering of smooth isosurfaces. *The Journal of Visualization and Computer Animation*, 10(2):79–90, April–June 1999.

[LHJ00] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution Techniques for Interactive Hardware Texturing-based Volume Visualization. In R. F. Erbacher, P. C. Chen, J. C. Roberts, and Craig M. Wittenbrink, editors, *Visual Data Exploration and Analysis*, pages 365–374, Bellingham, Washington, January 2000. SPIE – The International Society for Optical Engineering.

[Lit50] D.E. Littlewood. *A University Algebra*. William Heinemann, Ltd., London, England, 1950.

[LJH99] Eric C. LaMar, Ken Joy, and Bernd Hamann. Multi-resolution techniques for interactive hardware texturing-based volume visualization. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization 99*, pages 355–361. IEEE, ACM Press, October 25-29 1999.

[LL94] Philippe Lacroute and Marc Levoy. Fast Volume Rendering Using a Shear–Warp Factorization of the Viewing Transformation. In *Proceedings of Siggraph 94*, pages 451–458. ACM, July 1994.

[MB98] Tom McReynolds and Davis Blythe. *Siggraph '98 "Advanced Graphics Programming Techniques Using OpenGL" course notes*. ACM, July 1998.

[MBDM97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinite Reality: a Real-Time Graphics System. In *Proceedings of Siggraph 97*, pages 293–302. ACM, August 1997.

[NFHL91] Gregory M. Nielson, Thomas A. Foley, Bernd Hamann, and David A. Lane. Visualizing and modeling scattered multivariate data. *IEEE Computer Graphics and Applications*, 11(3):47–55, May 1991.

[NH91] Gregory M. Nielson and Bernd Hamann. The Asymptotic Decider: Removing the Ambiguity in Marching Cubes. In *Proceedings of Visualization '91*, pages 83–91, 1991.

[OHA93] Frank A. Ortega, Charles D. Hansen, and James P. Ahrens. Fast Data Parallel Polygon Rendering. In IEEE, editor, *Proceedings, Supercomput-*

ing '93: Portland, Oregon, November 15–19, 1993, pages 709–718. IEEE Computer Society Press, November 1993.

[SA98] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2).* Silicon Graphics Computer Systems, Mountain View, Ca, 1998.

[SFYC96] Raj Shekhar, Elias Fayad, Roni Yagel, and J. Fredrick Cornhill. Octree-based decimation of marching cubes surfaces. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization 96*, pages 335–342, Los Alamitos, October 27–November 1 1996. IEEE.

[SS92] Peter Schröder and Gordon Stoll. Data parallel volume rendering as line drawing. In *Proceedings of 1992 Workshop on Volume Visualization*, pages 25–32. ACM, 1992.

[TQ97] Tommaso Toffoli and Jason Quick. Three-dimensional rotations by three shears. *Graphical Models and Image Processing*, 59:89–96, 1997.

[VK96] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *1996 Volume Visualization Symposium*, pages 23–30. IEEE, October 1996. ISBN 0-89791-741-3.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently Using Graphics Hardware In Volume Rendering Applications. In *Proceedings of Siggraph 98*, pages 169–177. ACM, 19-24July 1998.

[Web90] Robert E. Webber. Ray tracing voxel data via biquadratic local surface interpolation. *The Visual Computer*, 6(1):8–15, February 1990.

[WV91] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. In Thomas W. Sederberg, editor, *Proceedings of Siggraph 91*, volume 25, pages 275–284. ACM, ACM, July 28–August 2 1991.

[WVW94] Orion Wilson, Allen Van Gelder, and Jane Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, June 29 1994.

[YK92] Roni Yagel and Arie Kaufman. Template-Based Volume Viewing. In A. Kilgour and L. Kjelldahl, editors, *Computer Graphics Forum (EUROGRAPHICS '92 Proceedings)*, volume 11, pages 153–167, September 1992.