Multiresolution Applications in Computer Graphics: Curves, Images, and Video

by

Adam Finkelstein

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by_____

(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Multiresolution Applications in Computer Graphics: Curves, Images, and Video

by Adam Finkelstein

Chairperson of Supervisory Committee: *Assistant Professor David H. Salesin*
Computer Science and Engineering

This dissertation presents some new representations and algorithms for multiresolution curves, multiresolution images, and multiresolution video, and it demonstrates several of their applications.

The multiresolution curve representation, which is based on wavelets, conveniently supports a variety of operations: smoothing a curve; editing the overall form of a curve while preserving its details; changing its fine details while maintaining its overall sweep; and approximating a curve for scan conversion.

For images, we use multiresolution analysis for "content-based image querying"—searching an image database using a query image that is similar to an intended target. The image-querying algorithm is remarkably effective, and fast enough to be performed on a database of 20,000 images at interactive speeds as a query is sketched.

Finally, we describe "multiresolution video," which allows for varying spatial and temporal resolutions in different parts of a video sequence. The representation supports a variety of applications: interactive viewing of large, multiresolution scientific data sets at different levels of detail; constant-speed display of video in the presence of varying bandwidth or CPU load; enhanced video scrubbing; and "video clip art" editing and compositing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The first thanks go to my family for all their love and encouragement.

Also, I would like to thank my advisor, David Salesin, whose guidance, support, and insight made this work possible, and whose aesthetic sense is excellent and (I hope) contagious. I'm indebted to Tony DeRose for teaching me graphics and for so much advice along the way. Thanks also to the other members of my thesis committee for their contributions, and especially Richard Ladner and Ronen Barzel for their good humor. We are all indebted to Ed Lazowska in his dual role as technology cheerleader and the man who makes it happen.

Chuck Jacobs was instrumental in the work described in Chapters 4 and 5 of this dissertation. Thanks, Chuck, for all your help, great ideas, for the many late nights of these projects, and also for your friendship.

Thanks, Erin Wilson and Glenn Fleishman, for forgiving me all the times I was late, and for your support and encouragement. Thanks to my many friends in GRAIL, C110 and C112, above all, Mike Salisbury.

Thanks also go to Frankye Jones, Chris Cunnington, Scott Dakins and Alicen Smith for the million things that had to happen right away and did. And thanks to the lab staff for the best-maintained network in the West; we all really appreciate it.

Finally, thanks, all you ultimate frisbee people: Umatata, Biddy, Drives, Rest, Lemmings, and everyone at pick-up. This is how my sanity was preserved.

This research was funded in part by an Intel Graduate Fellowship, for which I am deeply grateful.

Chapter 1

INTRODUCTION

1.1 Motivation

Many computer graphics applications represent a function using a discrete set of samples. For example a photograph is typically represented on a computer by a finite collection of colored dots called *pixels*, or collectively, an *image*. The overall appearance of these pixels when viewed from a reasonable distance is that of the original photograph, in the same vein as the effect achieved by the Impressionist painters of the 19th century[PJ79]. Of course, it was impossible for these (or any) artists to achieve an *exact* reproduction of the scenes they painted; when scrutinized up close, their paintings exhibit many tiny brush strokes of unmixed colors, rather than the small details present in the original scene. In order to capture a scene perfectly, an artist would have to paint an infinite number of infinitesimally small brush strokes. Likewise, on a computer with finite storage capacity, an image must be represented by a finite collection of samples, and will not, in general, be represented exactly.

The samples may be used to reconstruct a function that is defined everywhere on its domain by treating them as coefficients for a corresponding set of basis functions defined over the domain. Thus, given the samples, one can evaluate the function everywhere, not just at the locations of the samples. For example, the pixels in an image usually correspond to box basis functions—squares of constant color in the image—so that the image has color not just at the centers of the pixels, but everywhere in between.

The size of the intervals between samples dictates the *resolution* of the representation—

the degree to which it is able to describe (*resolve*) fine details in the function it represents. Tiny details of frequencies higher than half the sampling rate (the Nyquist limit [FvDFH90]) cannot be resolved. When we zoom in to an image there is no more fine detail to be seen; the image usually looks either blocky or blurry, depending on the nature of our basis functions. In the case of an Impressionist painting, when we look more closely we just see brush strokes, even though there is paint everywhere on the canvas. There is an obvious tradeoff between resolution and storage requirements; more detail can be accommodated by increasing the number of samples, at the expense of increased storage.

The typical arrangement for a discrete set of samples deploys them at even intervals. In the case of an image, for example, the pixels are usually organized in a regular rectangular grid. We call this a *uniresolution* representation—a finite, discrete set of samples spaced uniformly over the domain of the function they represent. Uniresolution representations are ubiquitous for a number of reasons. They are simple, easy to understand, and easy to implement. Furthermore, they often make use of implicit data structures such as arrays, so there is no storage overhead beyond the actual data being represented. Finally, these representations usually have a predictable size, which is advantageous for storage and transmission.

However, uniresolution representations suffer from a number of liabilities:

- We must make an *a priori* decision about the resolution. It may be difficult to know in advance how much fine detail will be present in the function we wish to represent. A conservative guess that overestimates the amount of detail present wastes space.
- Once we have decided on a specific resolution, the representation cannot characterize higher-resolution detail. To achieve greater resolution we have to switch to a different (higher-resolution) representation.
- If the representation describes fine detail anywhere, the representation pays for that detail everywhere. Consider an image of a painting hanging on a wall. If the image has enough resolution to clearly portray the details of the painting, then the large

regions of constant color on the wall around it must also be represented at this high resolution, even though they might very well be represented by many fewer samples.

These limitations can be alleviated by using a *multiresolution* representation—one that accommodates differing amounts of detail in different parts of the function being represented. For example, a multiresolution representation for images would permit us to capture arbitrarily fine details in the numbers of the clock face in Figure 5.5 (on page 83) while maintaining only a few samples in the large regions of constant color. The Impressionist painters understood this principle and made use of it in their work. For example, in *Le Promenade,* 1875, Claude Monet used small brush strokes to paint a woman's face, but used much larger brush strokes to paint the sky behind her. For the artist, this variation saved time and effort, and also served to draw the viewer's attention to the woman's face. The scale difference between the small brush strokes and the large brush strokes in this painting is limited by the sizes of the brushes and the dexterity of the artist. So, while this painting is multiresolution in nature, the fidelity is still limited. However, with computers it is possible to achieve an arbitrary degree of fine detail using a multiresolution representation.

In addition to alleviating some of the limitations listed above, multiresolution representations often yield other advantages over uniresolution representations:

1. **Control.** Multiresolution representations facilitate manipulation of data at different levels of detail.

2. **Feature detection.** Multiresolution representations can facilitate identification of salient features in the functions they represent.

3. **Compression.** Multiresolution representations lead naturally to compression schemes that maintain only salient features and ignore areas of little or no detail.

4. **Refinement.** Multiresolution representations permit addition of fine detail to a function without converting the function to a higher resolution.

This dissertation advocates the use of multiresolution representations and demonstrates the application of such representations to three areas of computer graphics: curves, images, and video—functions in one, two, and three dimensions, respectively.

1.2 Overview of dissertation

The remainder of this chapter provides an overview of the dissertation and a summary of its contributions. Chapter 2 provides some background on wavelet analysis, a mathematical tool that is employed in the two subsequent chapters. Chapters 3, 4, and 5 describe multiresolution representations for curves, images, and video, respectively. Note that each of these applications will leverage several of the four advantages of multiresolution representations listed above.

Chapter 3 introduces *multiresolution curves*, a representation that provides control of curves at different levels of detail. In the chapter we first develop wavelets appropriate for open B-spline curves. Next, we demonstrate how these multiresolution curves support a variety of operations including smoothing, editing at different levels of detail, and compression appropriate for scan conversion.

Chapter 4 shows how wavelet analysis of images can be used for "content-based image querying"—searching an image database using a query image that is similar to an intended target. The method relies on the feature detection and compression aspects of the wavelet representation. First we discuss the data structures and algorithms required to implement the image querying application. Then we describe how to tune some of the parameters of the method using a large database of images. Finally, we report the results of several experiments testing how well the image querying metric works in practice, and how robust it is with respect to various distortions in the images.

Chapter 5 presents *multiresolution video*, a representation for video that accommodates different amounts of detail in different parts of the video sequence. The chapter begins by describing a data structure for multiresolution video that compresses video in areas of spatial

and temporal coherence. It then describes a number of algorithms for creating, viewing and editing video clips.

Finally, Chapter 6 draws some conclusions and outlines several general areas for future work. Each of Chapters 3, 4, and 5 contains a section called "Extensions" that describes possible extensions to the particular work described in that chapter, whereas the final chapter describes future work that is broader in nature.

1.3 Related Work

Because this dissertation explores the use of multiresolution representations in a variety of distinct applications, the work relating to these applications falls into disparate groups. Therefore, discussions of work relating to these specific problems has been relegated to the "Related Work" sections of the chapters to which they correspond. This section outlines the more general context of the use of multiresolution techniques in computer graphics. It is not intended to provide an exhaustive bibliography, but rather to give a number of examples that illustrate the variety of applications for which multiresolution techniques have proven beneficial.

For the reasons motivated in Section 1.1, images are an ideal arena for use of multiresolution representations. Tanimoto and Pavlidis [TP75] developed a hierarchical structure for images called *image pyramids* and applied them to image processing. Williams [Wil83] showed how a similar multiscale structure for images called *MIP maps* can be used to antialias texture maps. More recently, Berman *et al*. [BBS94] demonstrated that a multiresolution representation for images based on wavelets could be used for efficient painting and compositing of images with different levels of detail in different locations. Related to these multiscale image representations is the user interface metaphor introduced by Perlin and Fox [PF93] in which the user is allowed to zoom in and out of an infinite desktop to store and view information at many different scales.

An important problem in image synthesis is the creation of images that have interesting

detail at different scales. Pioneered by Mandlebrot [Man83], *fractals* often have the property that they are self-similar at diffent scales. Perlin and Velho [PV95] generated images with arbitrarily fine detail using procedural multiscale textures. Heeger and Bergen [HB95] showed how a new image may be created by replicating the texture found in an example image, based on a multiresolution analysis of the texture.

Multiresolution analysis has also been employed to accelerate computationally expensive tasks, by focusing the work on areas that need more calculation and performing less work in less difficult areas. For example, Liu *et al*.[LGC94] used a wavelet basis for solving spacetime constraints problems in which physically-based motion for animations is generated based on user-directed goals. Gortler *et al*.and Christensen *et al*.[GSCH93, CSS+96] accelerated global illumination calculations, also by using a wavelet basis. Furthermore, Meyers [Mey94] employed wavelets to reduce the computational complexity of tiling a surface based on planar contours.

Surfaces also present an ideal domain for applying multiresolution techniques because of the large amount of information involved and because they can be difficult to manipulate. Forsey and Bartels [FB88] introduced *hierarchical B-splines*, a representation that permits arbitrary refinement of a surface and manipulation at different levels of detail. Lounsbury *et al*. [LDW93] developed wavelet analysis for surfaces of arbitrary topology and used the wavelet representation to automatically generate approximations based on viewing distance. Eck *et al*. [EDD+95] extended Lounsbury's analysis to meshes of arbitrary connectivity. Schröder and Sweldens [SS95a] developed wavelets for efficiently representing functions over spherical domains. Finally, Certain *et al*.[CPD+96] show that the color and shape information of a surface can be efficiently represented in separate wavelet bases, as demonstrated with an interactive viewer designed for progressive transmission of meshes.

All of these projects share the benefits of a multiresolution representation: control, feature detection, compression, and refinement.

1.4 Contributions

The specific contributions of this dissertation are:

- *A multiresolution representation for smooth, open curves based on wavelets*

  We develop wavelets for open cubic B-spline curves. In contrast to closed curves, open curves present a challenge to traditional wavelet constructions. Because the basis functions near the endpoints of the curve behave differently from those in the interior, the functions cannot all be translates and scales of one another, as in the traditional formulation.

- *Algorithms for manipulating multiresolution curves*

  We provide algorithms for a number of operations on multiresolution curves: smoothing, editing at different levels of detail, and approximation for scan conversion. The wavelet representation directly facilitates smoothing and editing operations at a series of discrete levels of detail. We extend these operations to apply on a continuum of levels of detail. We also show how fine detail from one curve can be captured and applied to a different curve to change its character.

- *An image querying method based on a multiresolution representation of images*

  We form a distance metric for images suitable for searching a large database of images. We also describe an efficient organization for the image database and an algorithm to quickly rank the images it contains, given a query image. The image-querying metric we describe is remarkably effective, and fast enough to be performed on a database of 20,000 images at interactive speeds as a query is sketched.

- *A multiresolution representation for video*

  We present a representation for video that allows for varying spatial and temporal resolutions in different parts of a video sequence. The representation compactly

stores regions of the video with either spatial or temporal coherence, and provides a controlled degree of lossy compression.

- *Algorithms for creating, viewing and editing video in this representation*

  We describe algorithms for creating multiresolution video from uniresolution source material. We show how to efficiently display multiresolution video at arbitrary spatial scales and temporal rates. We provide algorithms for translating, scaling, and compositing multiresolution video clips. Finally we show how these algorithms support a number of applications, such as constant perceived-speed playback, enhanced video searching, and the assembly of multiresolution videos from video "clip-art" elements.

The material that appears in this dissertation is largely drawn from the the research described in three consecutive Proceedings of the ACM SIGGRAPH Conference [FS94, JFS95, FJS96], each of which is self-contained. A more general resource for the use of wavelets in computer graphics is the monograph by Stollnitz *et al*. [SDS96], several chapters of which also draw heavily from the first two of these SIGGRAPH papers.

Chapter 2

BACKGROUND ON WAVELETS

This chapter provides some background on wavelets and multiresolution analysis. Multiresolution analysis is a simple mathematical tool that has found a wide variety of applications in recent years, including signal analysis [Mal89], image processing [DJL92], and numerical analysis [BCR91]. Informally, wavelets are the basis functions for multiresolution analysis; they will be be described more precisely below.

In Chapter 3, multiresolution analysis will be used to formulate a multiresolution representation for open curves that facilitates a variety of display and editing operations. In Chapter 4, a multiresolution analysis of images will lead to a distance metric for images. Not all multiresolution representations are based on multiresolution analysis, which refers to the specific mathematical construction visited in this chapter. In Chapter 5, we will describe a representation for video that does *not* make use of multiresolution analysis and wavelets.

2.1 General formulation

Rather than presenting the classical multiresolution analysis developed by Mallat [Mal89], we present here a slightly generalized version of the theory, following Lounsbery *et al.* [LDW93], that is more convenient for our applications.[1]

Consider a discrete signal C^n, expressed as a column vector of samples $[c_1^n, \ldots, c_m^n]^{\mathrm{T}}$. In

[1] The more general theory described here differs from Mallat's original formulation by relaxing his condition that the basis functions must be translates and scales of one another.

our curve application described in Chapter 3, for example, the samples c_i^n will be the curve's control points in $\mathbb{R}^2$.

Suppose we wish to create a low-resolution version C^{n-1} of C^n with a fewer number of samples m'. The standard approach for creating the m' samples of C^{n-1} is to use some form of linear filtering and subsampling on the m samples of C^n. This process can be expressed as a matrix equation

$$C^{n-1} = A^n C^n \tag{2.1}$$

where A^n is an $m' \times m$ matrix.

Since C^{n-1} contains fewer samples than C^n, it is intuitively clear that some amount of detail is lost in this filtering process. If A^n is chosen appropriately, it is possible to capture the lost detail as another signal D^{n-1} with $m - m'$ samples, computed by

$$D^{n-1} = B^n C^n \tag{2.2}$$

where B^n is an $(m - m') \times m$ matrix, which is related to matrix A^n. The pair of matrices A^n and B^n are called *analysis filters*. The process of splitting a signal C^n into a low-resolution version C^{n-1} and detail D^{n-1} is called *decomposition*. Usually m' is roughly half of m, so that C^{n-1} and D^{n-1} are roughly equal in size.

Note that C^{n-1} and D^{n-1} together have the same amount of information as C^n. If A^n and B^n are chosen correctly, then the original signal C^n can be recovered from C^{n-1} and D^{n-1} by using another pair of matrices P^n and Q^n, called *synthesis filters*, as follows:

$$C^n = P^n C^{n-1} + Q^n D^{n-1} \tag{2.3}$$

Recovering C^n from C^{n-1} and D^{n-1} is called *reconstruction*.

The procedure for splitting C^n into a low-resolution part C^{n-1} and a detail part D^{n-1} can be applied recursively to the new signal C^{n-1}. Thus, the original signal can be expressed as a hierarchy of lower-resolution signals $C^0, \ldots, C^{n-1}$ and details $D^0, \ldots, D^{n-1}$, as shown in Figure 2.1. This recursive process is known as a *filter bank*.

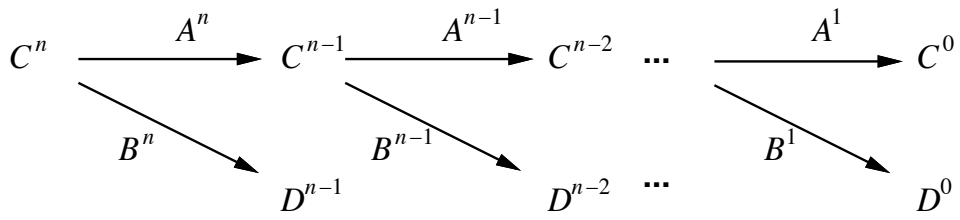


Figure 2.1: The filter bank.

Since the original signal C^n can be recovered from the sequence $C^0, D^0, D^1, \ldots, D^{n-1}$, this sequence can be thought of as a transform of the original signal known as a *wavelet transform*. Note that the total size of the transform $C^0, D^0, \ldots, D^{n-1}$ is the same as that of the original signal C^n, so no extra storage is required.[2]

Wavelet transforms have a number of properties that make them attractive for signal processing. First, if the filters A^j, B^j, P^j, and Q^j are constructed to be sparse, then the filter bank operation can be performed very quickly—often in $O(m)$ time, where m is the length of the vector C^n. Second, for many of the signals encountered in practice, a large percentage of the entries in the wavelet transform are negligible. Wavelet compression methods can therefore approximate the original set of samples in C^n by storing only the significant coefficients of the wavelet transform. Impressive compression rates have been reported for univariate signals as well as for images [DJL92]. Chapter 4 will capitalize on this compression technique in order to distill images down to extremely small "signatures" as part of an image querying method.

As suggested by the treatment above, all that is needed for performing a wavelet transform is an appropriate set of analysis and synthesis filters A^j, B^j, P^j, and Q^j. To see how to

[2] In truth, no extra storage is required in the ideal case where the samples in C^n are real numbers. However, dending on how these numbers are represented on a computer, more bits may be required for the samples in $C^0, D^0, D^1, \ldots, D^{n-1}$ than were used to represent the samples in C^n, or else some information may be lost. In practice, floating-point numbers typically have sufficient precision for most applications, but Section 5.2.5 describes how this issue was a factor in our decision not to use wavelets for multiresolution video.

construct these filters, we associate with each signal C^n a function $f^n(u)$ with $u \in [0,1]$:

$$f^n(u) \quad = \quad \Phi^n(u)\, C^n \tag{2.4}$$

where $\Phi^n(u)$ is a row matrix of basis functions $[\phi_1^n(u), \ldots, \phi_m^n(u)]$, called *scaling functions*. Thus, we treat the samples in C^n as coefficients for basis functions, as discussed in Section 1.1. In the curve-editing application of Chapter 3, for example, the scaling functions are the endpoint-interpolating B-spline basis functions, in which case the function $f^n(u)$ is an endpoint-interpolating B-spline curve.[3]

Scaling functions are required to be *refinable*; that is, for all j in $[1,n]$ there must exist an $m \times m'$ matrix P^j such that

$$\Phi^{j-1} \quad = \quad \Phi^j P^j \tag{2.5}$$

In other words, each scaling function at level $j-1$ must be expressible as a linear combination of "finer" level j scaling functions. As suggested by the notation, the refinement matrix in equation (2.5) is the same as the synthesis filter P^j of equation (2.3).

Next, let V^j be the linear space spanned by the set of scaling functions Φ^j. The refinement condition on Φ^j implies that these linear spaces are nested: $V^0 \subset V^1 \subset \cdots \subset V^n$. Choosing an inner product for Φ^j, the basis functions in V^j, allows us to define W^j as the *orthogonal complement* of V^j in V^{j+1}. That is, the space W^j whose basis functions $\Psi^j = [\psi_1^j(u), \ldots, \psi_{m-m'}^j(u)]$ are such that Φ^j and Ψ^j together form a basis for V^{j+1}, and every $\psi_i^j(u)$ is orthogonal to every $\phi_i^j(u)$ under the chosen inner product.[4]

The basis functions $\psi_i^j(u)$ are called *wavelets*, perhaps because they tend to look like waves that are localized in space (see Figure 3.1). We often use the term *wavelet analysis* interchangeably with "multiresolution analysis."

---

[3] For simplicity of notation, we often omit the explicit dependence on u when writing f^n and Φ^n.

[4] This orthogonality criterion holds for orthogonal wavelets such as Haar and Daubechies wavelets [Dau92], as well as semi-orthogonal wavelets such as Chui's B-spline wavelets [Chu92] and those developed in the following chapter. In contrast, bi-orthogonal wavelets such as the single-knot wavelets of Lounsbery *et al.* [LDW93] and wavelets found by the lifting scheme of Schröder and Sweldens [SS95a] relax this restriction. For a general discussion of these properties, see Stollnitz *et al.* [SDS96].

We can now construct the synthesis filter Q^j as the matrix that satisfies

$$\Psi^{j-1} = \Phi^j Q^j \tag{2.6}$$

Equations (2.5) and (2.6) can be expressed as a single equation by concatenating the matrices together:

$$\left[\Phi^{j-1} \mid \Psi^{j-1}\right] = \Phi^j \left[P^j \mid Q^j\right] \tag{2.7}$$

Finally, the analysis filters A^j and B^j are formed by the matrices satisfying the inverse relation:

$$\left[\Phi^{j-1} \mid \Psi^{j-1}\right] \left[\frac{A^j}{B^j}\right] = \Phi^j \tag{2.8}$$

Note that $\left[P^j \mid Q^j\right]$ and $\left[A^j \mid B^j\right]^{\mathrm{T}}$ are both square matrices. Thus,

$$\left[\frac{A^j}{B^j}\right] = \left[P^j \mid Q^j\right]^{-1} \tag{2.9}$$

from which it is easy to prove

$$\begin{aligned} A^j Q^j &= B^j P^j = \mathbf{0} \\ A^j P^j &= B^j Q^j = P^j A^j + Q^j B^j = \mathbf{1} \end{aligned} \tag{2.10}$$

where $\mathbf{0}$ and $\mathbf{1}$ are the matrix of zeros and the identity matrix, respectively. Equation (2.9) means that we can derive the analysis filters from the synthesis filters. The identities in (2.10) will be employed in Chapter 3.

2.2 Specializing the basis

The formulation above is very general. To construct wavelets for a specific basis we need to make several choices, as enumerated below:

1. *Choose the scaling functions $\Phi^j(u)$ for all levels j.*

   This choice determines the synthesis filters P^j.

2. *Select an inner product for any two functions f and g in V^j.*

   This choice determines the orthogonal complement spaces W^j.

3. *Select a set of wavelets $\Psi^j(u)$ that span W^j.*

   This choice determines the synthesis filters Q^j.

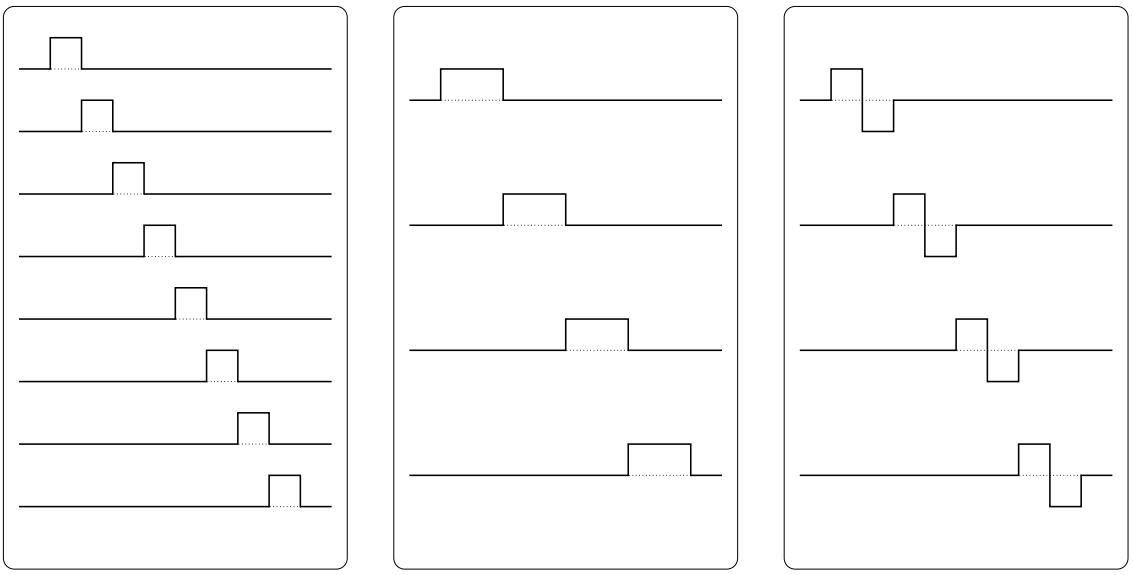
In the following section we will specialize multiresolution analysis for the simple case of the Haar basis. Chapter 4 will use these Haar wavelets to form a distance metric for images suitable for searching for images in a large database.

Chapter 3 will specialize multiresolution analysis for the more difficult case of endpoint-interpolating uniform cubic B-spline curves. We will then show that this multiresolution curve representation leads to a number of useful display and editing operations.

In Chapter 5 we describe a representation for multiresolution video that is *not* based on wavelets, but rather a specialized data structure. In that chapter we will discuss our reasons for not using wavelets in the context of video.

2.3 A simple example: Haar wavelets

In this section we develop the simplest wavelet basis—Haar wavelets. We begin by picking a set of scaling functions that span V^j for all j. In this case we use the simple "box" functions shown in Figure 2.2 (a) and (b), whose widths are inversely proportional to 2^j. These functions are refinable; note that each scaling function in V^j can trivially be expressed as a linear combination of two scaling functions in V^{j+1}. The synthesis filter P^j describes these linear combinations for each of the scaling functions in V^j. In the case of Haar wavelets,



(a) basis for V^{j+1} (b) basis for V^j (c) wavelets in W^j

Figure 2.2: Haar basis.

P^j is a banded matrix of the form:

$$P^j = \frac{1}{2}\begin{bmatrix} \ddots & & & \\ 1 & 0 & 0 & \\ 1 & 0 & 0 & \\ 0 & 1 & 0 & \\ 0 & 1 & 0 & \\ 0 & 0 & 1 & \\ 0 & 0 & 1 & \\ & & & \ddots \end{bmatrix}$$

The exact dimensions of this matrix depend on j.

For an inner product we use the standard form $\langle f \mid g \rangle = \int f(u)\, g(u)\, du$. This determines the orthogonal complement spaces W^j.

Finally we choose wavelets—a set of basis functions that span W^j. In this case we find the functions shown in Figure 2.2(c). These are the Haar wavelets. Note that (give or take a constant factor) these are the only functions that span the complement space W^j and are

orthogonal to each other.

These wavelets can be expressed as a linear combination of narrow scaling functions at level $j + 1$, using a refinement matrix of the form:

$$Q^j = \frac{1}{2}\begin{bmatrix} \ddots & & \\ 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \\ & & \ddots \end{bmatrix}$$

The wavelets, combined with the basis functions for V^j, span the same space as the functions in Figure 2.2(a), namely V^{j+1}. Also note that the wavelets are orthogonal to the basis functions for V^j.

Together, the synthesis filters P^j and Q^j determine the analysis filters A^j and B^j by equation (2.9). In this case, with simple Haar wavelets, our analysis filters are:

$$A^j = P^{j\,\mathrm{T}}$$
$$B^j = Q^{j\,\mathrm{T}}$$

although this is not true in general, as will become obvious in Section 3.2.1.

With the analysis and synthesis filters in hand, we are ready to explore some applications of wavelets. The Haar wavelets described in this section will be used in Chapter 4. Chapter 3 will develop other wavelets.

Chapter 3

MULTIRESOLUTION CURVES

3.1 Introduction

In this chapter, we develop a multiresolution representation for curves and show how these *multiresolution curves* provide a single, unified framework for supporting several important operations:

- the ability to change the overall "sweep" of a curve while maintaining its fine details, or "character" (Figure 3.5);
- the ability to modify a curve's "character" without affecting its overall "sweep" (Figure 3.8);
- the ability to edit a curve at any continuous level of detail, allowing an arbitrary portion of the curve to be affected through direct manipulation (Figure 3.6);
- continuous levels of smoothing, in which undesirable features are removed from a curve (Figure 3.4); and
- curve approximation, or "fitting," within a guaranteed maximum error tolerance, for scan conversion and other applications (Figures 3.10 and 3.11).

The representation requires no extra storage beyond that of the original uniresolution representation (m B-spline control points), and the algorithms that use it are both simple and

fast, typically linear in m.

There are many applications for multiresolution curves. For computer-aided design, cross-sectional curves are frequently used to specify surfaces. In keyframe animation, curves are used to control parameter interpolation. For 3D modeling and animation, "backbone" curves are manipulated to specify object deformations. Curves are used to describe regions of constant color or texture for graphic design. Likewise, in font design, the outlines of characters are typically represented as curves. And in pen-and-ink illustrations, curves are the basic elements of the finished piece. In all of these situations, the editing, smoothing, and approximation techniques we describe can be powerful tools.

3.1.1 Related work

Some of the algorithms supported by multiresolution curves are new, such as the ability to edit a curve at any continuous level of detail and the ability to change a curve's character without affecting its overall sweep. However, the majority of applications described in this chapter have already been addressed in one form or another. Although the algorithms we describe compare favorably with most of this previous work, it is the convenience with which the multiresolution representation supports such a wide variety of operations that makes it so useful. We survey here some of these previous techniques.

Forsey and Bartels [FB88] employ hierarchical B-splines to address the problem of editing the overall form of a surface while maintaining its details. Their original formulation requires the user to design an explicit hierarchy into the model. In later work [FB91], they describe a method for recursively fitting a hierarchical surface to a set of data by first fitting a coarse approximation and then refining in areas where the residual is large. This construction is similar in spirit to the filter bank process used in multiresolution analysis, as described in Section 2.1. One significant difference is that in their formulation there are an infinite number of possible representations for the same surface, whereas the multiresolution curve representation is unique for a given shape and parameterization. One reason to

strive for a unique representation of a given shape is that it yields consistent behavior when it is manipulated, whereas different representations for the same shape might yield different behaviors. Fowler [Fow92] and Welch and Witkin [WW92] also describe methods in which editing can be performed over narrower or broader regions of a surface; however, in neither of these works is there an attempt to preserve the higher-resolution detail beneath the edited region.

Curve and surface smoothing algorithms that minimize various energy norms have also been studied; these are surveyed in Hoschek and Lasser [HL92]. One example is the work of Celniker and Gossard [CG91], in which a fairness functional is applied to hand-drawn curves, as well as to surfaces. Gortler and Cohen [GC94] also generate fair surfaces by minimizing an energy norm, but they accelerate the optimization using a B-spline wavelet basis similar to the one developed in this chapter. The method we describe is really a least-squares type of smoothing, which is much simpler but supports continuous levels of smoothing that behave quite reasonably and intuitively in practice.

Many schemes for approximating curves within specified error tolerances have also been explored [BC90, LM87, PS83, Sch88]. Most of this research has centered on various forms of knot removal for representing curves efficiently with non-uniform B-splines. In this chapter, we look at the practical concern of producing a small number of Bézier segments that approximate the curve well, since these segments are the standard representation for curves in PostScript [Ado90], the most common page description language. Our requirements are also somewhat different than those of most previous curve-fitting methods. In particular, for our application of scan conversion we do not require any continuity constraints for the approximating curve. Relaxing this condition can allow much higher compression ratios.

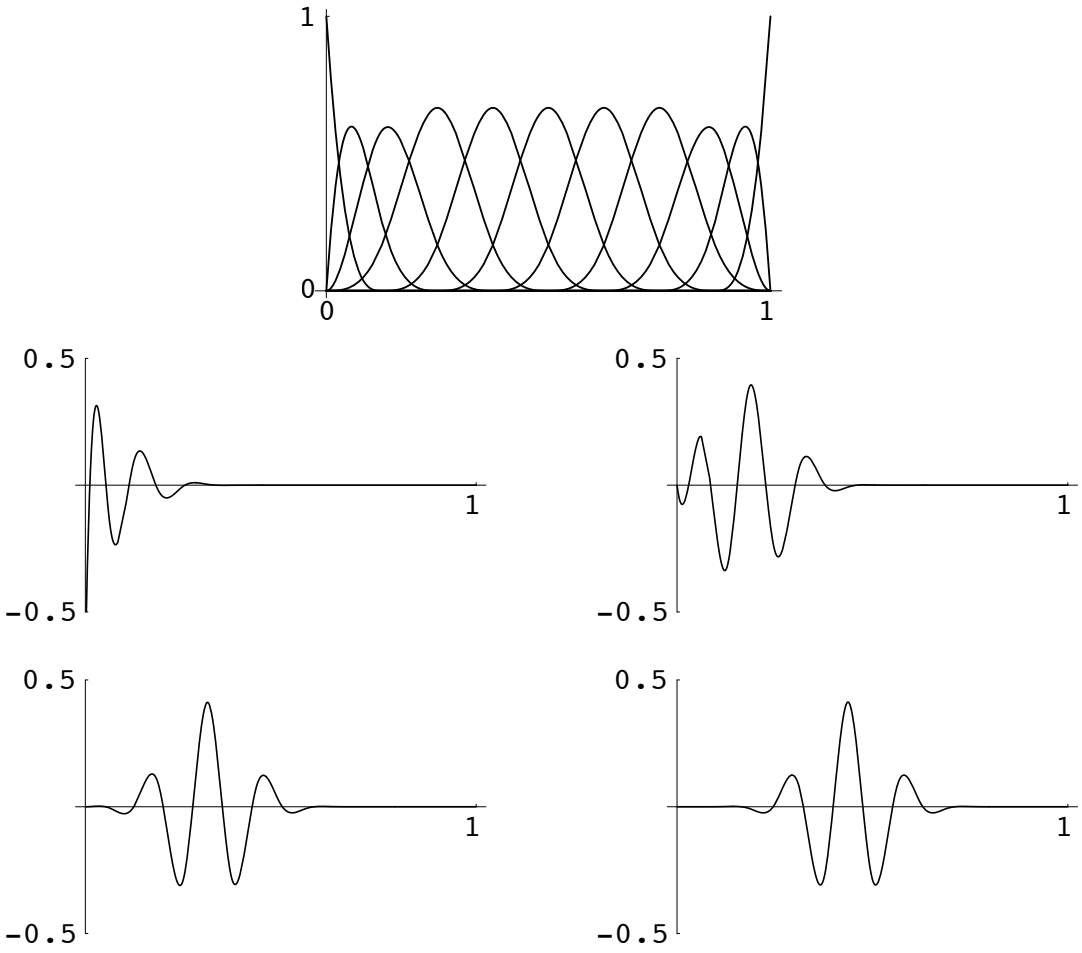


Figure 3.1: The B-spline scaling functions and the first four wavelets at level 3.

3.1.2 Overview

The next section specializes the theory of multiresolution analysis to develop a multiresolution representation for B-spline curves. Sections 3.3, 3.4, and 3.5 describe how this representation can be used to support efficient smoothing, editing, and scan conversion. Finally, Section 3.6 suggests some areas for extending this work.

3.2 Theory of multiresolution curves

To begin this section we apply the theory of wavelets and multiresolution analysis to endpoint-interpolating B-spline curves. Next we show how to perform analysis efficiently for these curves.

3.2.1 Multiresolution endpoint-interpolating B-splines

The multiresolution framework described in Section 2.1 is quite general. In this section, we restrict our attention to multiresolution analysis for *endpoint-interpolating uniform cubic B-splines*—cubic B-splines defined on a knot sequence that is uniformly spaced everywhere except at its ends, where its knots have multiplicity 4. Figure 3.1 shows some examples of these B-spline scaling functions and their wavelets.

A construction similar to the one described here has also been independently proposed by Chui and Quak [CQ92]. Note that multiresolution constructions can be built for other types of splines as well, such as uniform B-splines [Chu92], and non-uniform B-splines with arbitrary knot sequences [LM92]. A construction applicable to subdivision surfaces is discussed by Lounsbery *et al*. [LDW93]. Sections 3.6 and 6.3 discuss as possible areas for future work the application of techniques described in this chapter to several of these other multiresolution constructions.

To construct *multiresolution curves* for the endpoint-interpolating cubic B-splines, we need to make several choices, as prescribed in section Section 2.2:

1. *Choose the scaling functions* $\Phi^j(u)$.

2. *Select an inner product*.

3. *Select a set of wavelets* $\Psi^j(u)$ *that span* W^j.

We elaborate on these choices for the remainder of this section.

Step 1: Choose the scaling functions

In our case, the scaling functions are the basis functions of endpoint-interpolating uniform cubic B-splines. We chose to work with these curves for a number of reasons:

- They describe open curves that interpolate their first and last control points.
- They have curvature continuity, so they are smooth.
- They are refinable, as required by equation (2.5).
- They can be described by simple cubic polynomials.
- The are well-understood and used in practice.

These curves are discussed in detail in many texts on computer-aided design [BBB87, Far92, HL92].

The i-th B-spline basis function ${}^dB_i$ of degree d over knot sequence $u_1, u_2, \ldots, u_n$ may be found by the Mansfield, de Boor, Cox recursion [Far92]:

$$ {}^dB_i(u) \quad = \quad \frac{u - u_{i-1}}{u_{i+d-1} - u_{i-1}} \; {}^{d-1}B_i(u) \quad + \quad \frac{u_{i+d} - u}{u_{i+d} - u_i} \; {}^{d-1}B_{i+1}(u) $$

Every recursion needs a base case; the degree 0 B-splines are:

$$ {}^0B_i(u) = \begin{cases} 1 & \text{if } u_{i-1} \leq u \leq u_i \\ 0 & \text{otherwise} \end{cases} $$

These piecewise-constant "box" basis functions are the scaling functions corresponding to the Haar wavelets. Thus, the B-spline wavelets that we are developing in this section generalize the Haar wavelets developed in Section 2.3. For the remainder of this chapter we will be working with the cubic (degree 3) B-splines:

$$ \Phi^j(u) \quad = \quad \begin{bmatrix} {}^3B_1(u) & {}^3B_2(u) & {}^3B_3(u) & \ldots & {}^3B_{2^j+3}(u) \end{bmatrix} $$

The P^j matrices shown in Figure 3.2 encode how each endpoint-interpolating cubic B-spline in V^{j-1} can be refined—expressed as a linear combination of narrower B-splines in V^j. These matrices have dimensions $(2^j+3) \times (2^{j-1}+3)$. The middle columns, for $j \geq 3$, are given by vertical translates of the fourth column, shifted down by 2 places for each column.

$$P^1 = \frac{1}{16}\begin{bmatrix} 16 & 0 & 0 & 0 \\ 8 & 8 & 0 & 0 \\ 0 & 8 & 8 & 0 \\ 0 & 0 & 8 & 8 \\ 0 & 0 & 0 & 16 \end{bmatrix}$$

$$P^2 = \frac{1}{16}\begin{bmatrix} 16 & 0 & 0 & 0 & 0 \\ 8 & 8 & 0 & 0 & 0 \\ 0 & 12 & 4 & 0 & 0 \\ 0 & 3 & 10 & 3 & 0 \\ 0 & 0 & 4 & 12 & 0 \\ 0 & 0 & 0 & 8 & 8 \\ 0 & 0 & 0 & 0 & 16 \end{bmatrix}$$

$$P^{j\geq 3} = \frac{1}{16}\begin{bmatrix} 16 & 0 & 0 & 0 & 0 & 0 & \\ 8 & 8 & 0 & 0 & 0 & 0 & \\ 0 & 12 & 4 & 0 & 0 & 0 & \\ 0 & 3 & 11 & 2 & 0 & 0 & \\ 0 & 0 & 8 & 8 & 0 & 0 & \cdots \\ 0 & 0 & 2 & 12 & 2 & 0 & \\ 0 & 0 & 0 & 8 & 8 & 0 & \\ 0 & 0 & 0 & 2 & 12 & 2 & \\ 0 & 0 & 0 & 0 & 8 & 8 & \\ 0 & 0 & 0 & 0 & 2 & 12 & \\ & & \vdots & & & & \ddots \end{bmatrix}$$

$$Q^1 = \frac{1}{3}\begin{bmatrix} 1 \\ -2 \\ 3 \\ -2 \\ 1 \end{bmatrix}$$

$$Q^2 = \frac{1}{2064}\begin{bmatrix} -1368 & 0 \\ 2064 & 240 \\ -1793 & -691 \\ 1053 & 1053 \\ -691 & -1793 \\ 240 & 2064 \\ 0 & -1368 \end{bmatrix}$$

$$Q^3 = \begin{bmatrix} \frac{-394762}{574765} & 0 & 0 & 0 \\ 1 & \frac{-7166160}{28124263} & 0 & 0 \\ \frac{-33030599}{41383080} & \frac{333497715}{478112471} & \frac{6908335}{478112471} & 0 \\ \frac{633094403}{1655323200} & \frac{-881412943}{956224942} & \frac{-74736797}{956224942} & \frac{27877}{1655323200} \\ \frac{-19083341}{137943600} & 1 & \frac{8833647}{28124263} & \frac{-864187}{413830800} \\ \frac{4681957}{165532320} & \frac{-689203555}{956224942} & \frac{-689203555}{956224942} & \frac{4681957}{165532320} \\ \frac{-864187}{413830800} & \frac{8833647}{28124263} & 1 & \frac{-19083341}{137943600} \\ \frac{27877}{1655323200} & \frac{-74736797}{956224942} & \frac{-881412943}{956224942} & \frac{633094403}{1655323200} \\ 0 & \frac{6908335}{478112471} & \frac{333497715}{478112471} & \frac{-33030599}{41383080} \\ 0 & 0 & \frac{-7166160}{28124263} & 1 \\ 0 & 0 & 0 & \frac{-394762}{574765} \end{bmatrix}$$

$$Q^{j\geq 4} = \begin{bmatrix} \frac{-394762}{574765} & 0 & 0 & 0 & \\ 1 & \frac{-1050072320}{4096633377} & 0 & 0 & \\ \frac{-33030599}{41383080} & \frac{2096854390}{2989435167} & \frac{307090}{19335989} & 0 & \\ \frac{633094403}{1655323200} & \frac{-11070246427}{11957740668} & \frac{-6643465}{77343956} & \frac{-1}{24264} & \\ \frac{-19083341}{137943600} & 1 & \frac{6646005}{19335989} & \frac{31}{6066} & \\ \frac{4681957}{165532320} & \frac{-157389496903}{221218202358} & \frac{-29839177}{38671978} & \frac{-559}{8088} & \\ \frac{-864187}{413830800} & \frac{1732435193}{5821531641} & 1 & \frac{988}{3033} & \cdots \\ \frac{27877}{1655323200} & \frac{-27809640281}{442436404716} & \frac{-58651607}{77343956} & \frac{-9241}{12132} & \\ 0 & \frac{171326708}{36869700393} & \frac{6261828}{19335989} & 1 & \\ 0 & \frac{-1381667}{36869700393} & \frac{-1328199}{19335989} & \frac{-9241}{12132} & \\ 0 & 0 & \frac{98208}{19335989} & \frac{988}{3033} & \\ 0 & 0 & \frac{-792}{19335989} & \frac{-559}{8088} & \\ 0 & 0 & 0 & \frac{31}{6066} & \\ 0 & 0 & 0 & \frac{-1}{24264} & \\ & \vdots & & & \ddots \end{bmatrix}$$

Figure 3.2: The synthesis filters P^j and Q^j for cubic B-splines.

Step 2: Select an inner product

We need an inner product for any two functions f and g in V^j. We use the standard form:

$$\langle f \mid g \rangle = \int f(u)\, g(u)\, du$$

Because our basis functions can be expressed as simple cubic polynomials, we can compute their inner products symbolically. Again, this choice determines the orthogonal complement spaces W^j.

Step 3: Select a set of wavelets

Finally, we find a set of wavelets $\Psi^j(u)$ that span W^j. We'll need some new notation. Given two row vectors of functions X and Y, let $[\langle X \mid Y \rangle]$ be the matrix whose kl-th entry is the inner product $\langle X_k \mid Y_l \rangle$. Since, by definition, scaling functions and wavelets at the same level j are orthogonal, we have

$$\left[\langle \Phi^j \mid \Psi^j \rangle\right] \;=\; \left[\langle \Phi^j \mid \Phi^{j+1} \rangle\right] Q^{j+1} \;=\; \mathbf{0},$$

so the columns of Q^{j+1} span the null space of $\left[\langle \Phi^j \mid \Phi^{j+1} \rangle\right]$. Since the scaling functions are piecewise-polynomial functions, it is straightforward to compute the entries of the matrix of inner products using symbolic integration in a symbolic math package. Next, we find a basis that spans the null space of this matrix, also a trivial task for a symbolic math package.

However there is no *unique* basis for the null space. We choose to further constrain the space by finding a matrix Q^{j+1} that has columns with the shortest runs of non-zero coefficients. This Q^{j+1} corresponds to the wavelets with minimal support. We prefer that the wavelets have minimal support because it leads to the narrowest possible bandwidth in the Q^j matrices, that is, faster synthesis. Because these computations can be performed symbolically, the fractions reported in the Q^j matrices of Figure 3.2 are exact (though ugly).

Together, the synthesis filters P^j and Q^j determine the analysis filters A^j and B^j by equation (2.9). However, as the following section will illustrate, these analysis filters lead to an inefficient decomposition and are unnecessary in practice.

$$I^{j\geq 3} = \frac{2^{-j}}{10080}\begin{bmatrix} 1440 & 882 & 186 & 12 & 0 & 0 & \\ 882 & 2232 & 1575 & 348 & 3 & 0 & \\ 186 & 1575 & 3294 & 2264 & 239 & 2 & \\ 12 & 348 & 2264 & 4832 & 2382 & 240 & \cdots \\ 0 & 3 & 239 & 2382 & 4832 & 2382 & \\ 0 & 0 & 2 & 240 & 2382 & 4832 & \\ 0 & 0 & 0 & 2 & 240 & 2382 & \\ 0 & 0 & 0 & 0 & 2 & 240 & \\ 0 & 0 & 0 & 0 & 0 & 2 & \\ & & & \vdots & & & \ddots \end{bmatrix}$$

Figure 3.3: The inner product matrices I^j for cubic B-splines.

3.2.2 Linear-time filter-bank algorithm

Because both the scaling functions and wavelets in our construction have compact support, the synthesis filters P^j and Q^j have a banded structure, allowing reconstruction in $O(m)$ time based on banded matrix multiplication [PFTF92]. However, a potential weakness of our framework is that the analysis filters A^j and B^j turn out to be dense, which would seem to imply an $O(m^2)$-time decomposition algorithm, where m is the number of control points in the original curve C^n. Fortunately, there is a clever trick, due to Quak and Weyrich [QW93], for performing the decomposition of endpoint-interpolating B-splines in $O(m)$-time. The implementation of their algorithm, which based on a transformation into the "dual-space," is described in this section. An alternate method that rearranges the rows and columns of equation (2.7) to form a banded system may be found in Stollnitz *et al*. [SDS96].

Let I^j and J^j be the inner product matrices $\left[\langle \Phi^j \mid \Phi^j \rangle\right]$ and $\left[\langle \Psi^j \mid \Psi^j \rangle\right]$, respectively. Quak and Weyrich [QW93] show that equations (2.1) and (2.2) can then be rewritten:

$$\begin{aligned} I^{j-1}C^{j-1} &= (P^j)^T I^j C^j \\ J^{j-1}D^{j-1} &= (Q^j)^T I^j C^j \end{aligned}$$

Since P^j, Q^j, and I^j are banded matrices, the right-hand side of these equations can be computed in linear time. What remains are two band-diagonal systems of equations, which can

also be solved in linear time using LU decomposition [PFTF92]. Thus, with precomputed matrices I^j, J^j, P^j, and Q^j (for all j) we can decompose C^j into C^{j-1} and D^{j-1} in linear time. Since there are about half as many elements in C^{j-1} as in C^j, half again in C^{j-2}, and so on, the entire filter bank analysis illustrated in Figure 2.1 can be found in linear time.

The matrices I^j for $j \geq 3$ are given in Figure 3.3. Note that I^j is a symmetric matrix with dimensions $(2^j + 3) \times (2^j + 3)$ whose middle columns, for $j \geq 3$, are given by vertical translates of the sixth column. The I^j matrices for $j < 3$ and the J^j matrices may be found by the following equations:

$$\begin{aligned} I^j &= (P^{j+1})^T I^{j+1} P^{j+1} \\ J^j &= (Q^{j+1})^T I^{j+1} Q^{j+1} \end{aligned}$$

Now that we have efficient algorithms for analysis and synthesis of multiresolution curves, we are in a position to explore some of their applications.

3.3 Smoothing

In this section, we address the following problem: *Given a curve with m control points C, construct a best least-squares-error approximating curve with m′ control points C′, where* $m' < m$. Here, we will assume that both curves are endpoint-interpolating uniform B-spline curves.

The multiresolution analysis framework allows this problem to be solved trivially, for certain values of m and m'. Assume for the moment that $m = 2^j + 3$ and $m' = 2^{j'} + 3$ for some nonnegative integers $j' < j$. Then the control points C' of the approximating curve are given by

$$C' = A^{j'+1} A^{j'+2} \cdots A^j C$$

In other words, we simply run the decomposition algorithm, as described by equation (2.1), until a curve with just m' control points is reached. Note that this process can

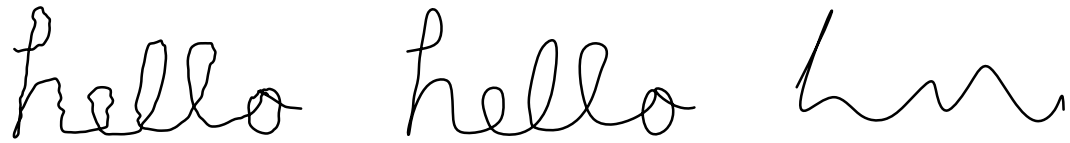


Figure 3.4: Smoothing a curve continuously. From left to right: the original curve at level 8.0, and smoother versions at levels 5.4 and 3.1.

be performed at interactive speeds for hundreds of control points using the linear-time algorithm described in the previous section.

One notable aspect of the multiresolution curve representation is its discrete nature. Thus, in our application it is easy to construct approximating curves with 4, 5, 7, 11, or any $2^j + 3$ control points efficiently, for any *integer level* j. However, there is no obvious way to quickly construct curves that have "levels" of smoothness in between.

The simplest solution we have found is to define a *fractional-level* curve $f^{j+t}(u)$ for some $0 \leq t \leq 1$ in terms of a linear interpolation between its two nearest integer-level curves $f^j(u)$ and $f^{j+1}(u)$, as follows:

$$\begin{aligned} f^{j+t}(u) &= (1-t) f^j(u) \; + \; t f^{j+1}(u) \\ &= (1-t)\, \Phi^j(u)\, C^j \; + \; t\, \Phi^{j+1}(u)\, C^{j+1} \end{aligned} \tag{3.1}$$

These fractional-level curves allow for continuous levels of smoothing. In our application a user can move a control slider and see the curve transform continuously from its smoothest (4 control point) form, up to its finest (m control point) version. Some fractional-level curves are shown in Figure 3.4.

3.4 Editing

Suppose we have a curve C^n and all of its low-resolution and detail parts $C^0, \ldots, C^{n-1}$ and $D^0, \ldots, D^{n-1}$. Multiresolution analysis allows for two very different kinds of curve

editing. If we modify some low-resolution version C^j and then add back in the detail $D^j, D^{j+1}, \ldots, D^{n-1}$, we will have modified the overall sweep of the curve (Figure 3.5). On the other hand, if we modify the set of detail functions D^j, D^{j+1}, ..., D^{n-1} but leave the low-resolution versions $C^0, \ldots, C^j$ intact, we will have modified the character of the curve, without affecting its overall sweep (Figure 3.8). These two types of editing are explored more fully below.

3.4.1 Editing the sweep

Editing the sweep of a curve at an integer level of the wavelet transform is simple. Let C^n be the control points of the original curve $f^n(u)$, let C^j be a low-resolution version of C^n, and let $\widehat{C}^j$ be an edited version of C^j, given by $\widehat{C}^j = C^j + \Delta C^j$. The edited version of the highest-resolution curve $\widehat{C}^n = C^n + \Delta C^n$ can be computed through reconstruction:

$$\begin{aligned} \widehat{C}^n &= C^n + \Delta C^n \\ &= C^n + P^n P^{n-1} \cdots P^{j+1} \Delta C^j \end{aligned}$$

Note that editing the sweep of the curve at lower levels of smoothing j affects larger portions of the high-resolution curve $f^n(u)$. At the lowest level, when $j = 0$, the entire curve is affected; at the highest level, when $j = n$, only the narrow portion influenced by one original control point is affected. The kind of flexibility that this multiresolution editing allows is suggested in Figures 3.5 and 3.6.

In addition to editing at integer levels of resolution, it is natural to ascribe meaning to editing at fractional levels as well. We would like the portion of the curve affected when editing at fractional level $j + t \in [j, j+1)$ to interpolate the portions affected at levels j and $j + 1$. Thus, as t increases from 0 to 1, the portion affected should gradually narrow down from that of level j to that of level $j + 1$, as demonstrated in the lower part of Figure 3.6.

Consider a fractional-level curve $f^{j+t}(u)$ given by equation (3.1). Let C^{j+t} be the set of

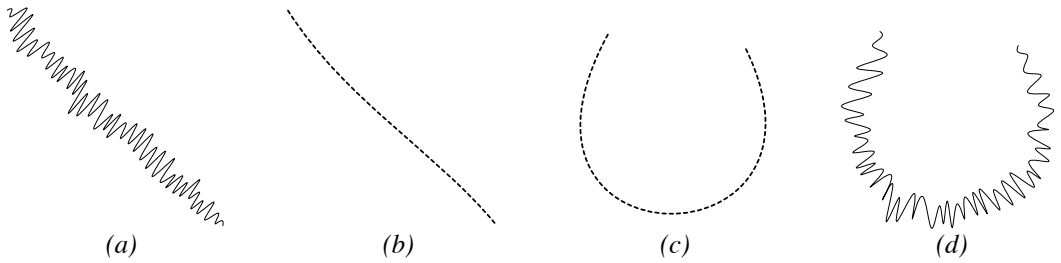


Figure 3.5: Changing the overall sweep of a curve without affecting its character. Given the original curve (a), the system extracts the overall sweep (b). If the user modifies the sweep (c), the system can reapply the detail (d).

control points associated with this curve; that is,

$$f^{j+t}(u) \;=\; \Phi^{j+1}(u)\, C^{j+t} \tag{3.2}$$

We obtain an expression for C^{j+t} by equating the right-hand sides of equations (3.1) and (3.2), and then applying equations (2.5) and (2.3):

$$\begin{aligned} C^{j+t} &= (1-t)\, P^{j+1}\, C^{j} \;+\; t\, C^{j+1} \\ &= P^{j+1}\, C^{j} \;+\; t\, Q^{j+1}\, D^{j} \end{aligned}$$

Suppose now that one of the control points c_i^{j+t} is modified by the user. In order to allow the portion of the curve affected to depend on t in the manner described above, the system will have to automatically move some of the nearby control points when c_i^{j+t} is modified. The distance that each of these control points is moved is inversely proportional to t. For example, when t is near 0, the control points in C^{j+t} are moved in conjunction so that the overall effect approaches that of editing a single control point at level j; when $t = 1$, the nearby control points are not moved at all, since the modified curve should correspond to moving just a single control point at level $j+1$.

Let ΔC^{j+t} be a vector describing how each control point of the fractional-level curve is modified: the i-th entry of ΔC^{j+t} is the user's change to the i-th control point; the other

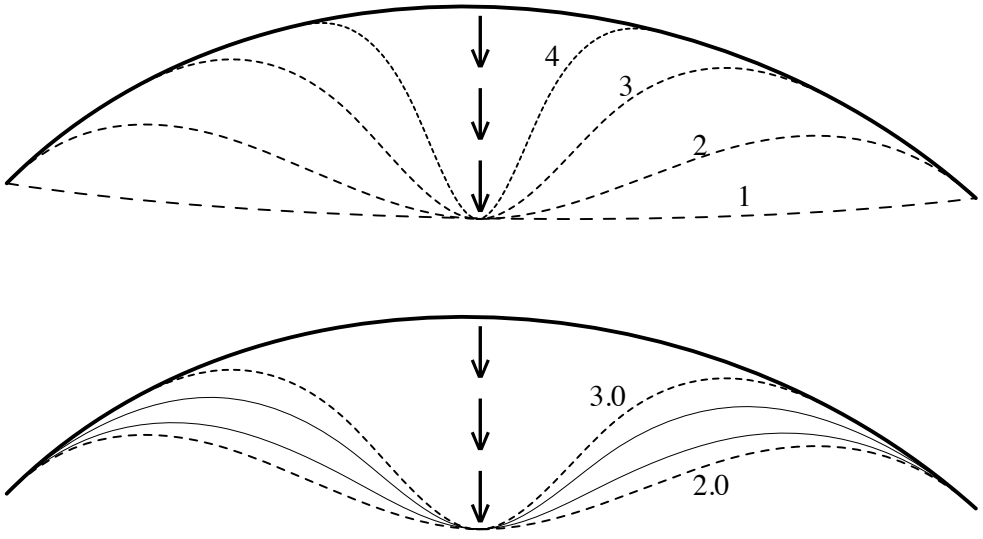


Figure 3.6: The middle of the dark curve is pulled. Upper: Editing at integer levels 1, 2, 3, and 4. Lower: Editing at fractional levels between 2.0 and 3.0.

entries reflect the computed movements of the other control points. Rather than solving for ΔC^{j+t} explicitly, our approach will be to break this vector into two components, a vector ΔC^j of changes to the control points at level j, and a vector ΔD^j of changes to the wavelet coefficients at level j:

$$\Delta C^{j+t} \;=\; P^{j+1}\,\Delta C^j \;+\; t\,Q^{j+1}\,\Delta D^j \tag{3.3}$$

Next, define $\Delta \acute{C}^{j+t}$ to be the user's change to the control points at level $j+t$, that is, a vector whose i-th entry is Δc_i^{j+t}, and whose other entries are 0. Define also a new vector $\Delta \acute{C}^j$ as a change to control points at level j necessary to make the modified control point c_i^{j+t} move to its new position. We choose the vector that is 0 everywhere except for one or two entries, depending on the index i of the modified control point. By examining the i-th row of the refinement matrix P^{j+1}, we can determine whether the modified control point is maximally influenced by either *one* control point c_k^{j+1} or *two* control points c_k^{j+1} and c_{k+1}^{j+1} at

level $j+1$. In the former case, we set $\Delta\acute{c}_k^j$ to be $\Delta c_i^{j+t}/P_{i,k}^{j+1}$. In the latter case, we set $\Delta\acute{c}_k^j$ and $\Delta\acute{c}_{k+1}^j$ to be $\Delta c_i^{j+t}/2P_{i,k}^{j+1}$.

Note that applying either change alone, $\Delta\acute{C}^{j+t}$ or $\Delta\acute{C}^j$, would cause the selected control point to move to its new position; however, the latter change would cause a larger portion of the curve to move. In order to have a "breadth" of change that gradually decreases as t goes from 0 to 1, we can interpolate between these two vectors, using some interpolation function $g(t)$:

$$\Delta C^{j+t} \;=\; (1-g(t))\,P^{j+1}\,\Delta\acute{C}^j \;+\; g(t)\,\Delta\acute{C}^{j+t} \tag{3.4}$$

Thus, ΔC^{j+t} will still move the selected control point to its new position, and it will also now control the "breadth" of change as a function of t.

Finally, equating the right-hand sides of equations (3.3) and (3.4), multiplying with either A^{j+1} or B^{j+1}, and employing the identities (2.10) yields the two expressions we need:

$$\begin{aligned} \Delta C^j &= (1-g(t))\,\Delta\acute{C}^j \;+\; g(t)\,A^{j+1}\,\Delta\acute{C}^{j+t} \\ \Delta D^j &= \frac{g(t)}{t}\,B^{j+1}\,\Delta\acute{C}^{j+t} \end{aligned} \tag{3.5}$$

We now have the choice of any function $g(t)$ that allows ΔD^j to increase monotonically from 0 to 1. We chose the function $g(t):=t^2$ because the change to the wavelet coefficients ΔD^j goes from 0 to 1 like t, and have found that this choice works well in practice.

The changes to the high-resolution control points are then reconstructed using a straightforward application of equation (2.3):

$$\Delta C^n \;=\; P^n P^{n-1}\cdots P^{j+2}\,(P^{j+1}\Delta C^j \;+\; Q^{j+1}\Delta D^j) \tag{3.6}$$

The fractional-level editing defined here works quite well in practice. Varying the editing level continuously gives a smooth and intuitive kind of change in the region of the curve affected, as suggested by Figure 3.6. Because the algorithmic complexity is just $O(m)$, the update is easily performed at interactive rates, even for curves with hundreds of control points.

Editing with direct manipulation

The fractional-level editing described above can be easily extended to accommodate *direct manipulation,* in which the user tugs on the smoothed curve directly rather than on its defining control points [BB89, FB88, Fow92, HHK92]. To use direct manipulation when editing at level $j + t$, we make use of the pseudo-inverse of the scaling functions at levels j and $j + 1$.

More precisely, suppose the user drags a point of the curve $f^{j+t}(u_0)$ to a new position $f^{j+t}(u_0) + \delta$. We can compute the least-squares change to the control points $\Delta\acute{C}^j$ and $\Delta\acute{C}^{j+t}$ at levels j and $j + t$ using the pseudo-inverses $(\Phi^j)^+$ and $(\Phi^{j+1})^+$ as follows:

$$\begin{aligned} \Delta\acute{C}^j &= (\Phi^j(u_0))^+ \, \delta \\ \Delta\acute{C}^{j+t} &= (\Phi^{j+1}(u_0))^+ \, \delta \end{aligned} \tag{3.7}$$

These two equations should be interpreted as applying to each dimension x and y separately. That is, δ should be a scalar (say, the change in x), and the left-hand side and the pseudo-inverses should both be column-matrices of scalars. The modified control points of the highest-resolution curve can then be computed in the same fashion outlined for control-point manipulation, by applying equations (3.5) and (3.6).

Note that the first step of the construction, equation (3.7), can be computed in constant time, since for cubic B-splines at most four of the entries of each pseudo-inverse are non-zero. The issue of finding the parameter value u_0 at which the curve passes closest to the selection point is a well-studied problem in root-finding, which can be handled in a number of ways [Sch88]. In our implementation, we scan-convert the curve once to find its parameter value at every illuminated pixel. This approach is easy to implement, and appears to provide a good trade-off between speed and accuracy for an interactive system.

For some applications, it may be more intuitive to drag on the high-resolution curve directly, rather than on the smoothed version of the curve. In this case, even when the curve's display resolution is at its highest level, it may still be useful to be able to tug on the curve at a lower editing resolution. In this way, varying levels of detail on the curve can

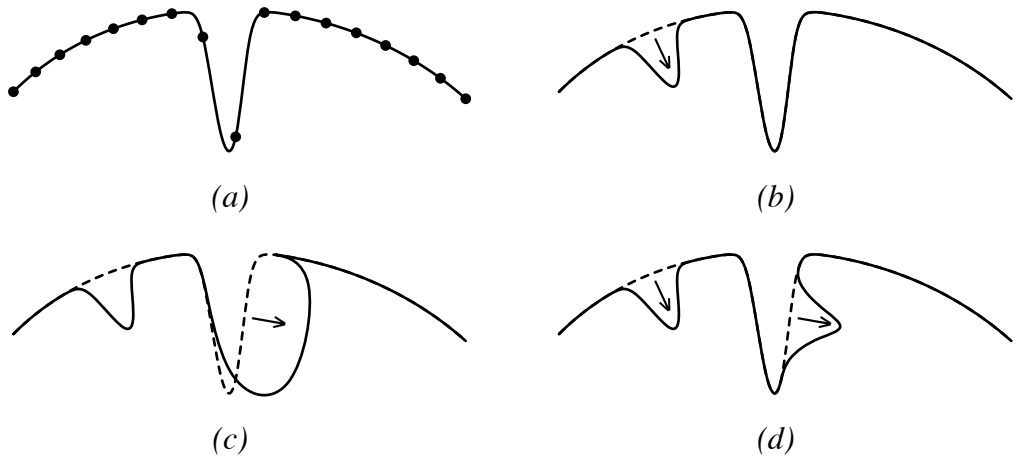
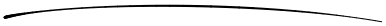


Figure 3.7: Curve (a) has a parameterization that is non-uniform with respect to its length. Direct manipulation on the left part of the curve (b) affects a much smaller fraction of the curve than does direct manipulation at the same level in the middle (c). The last figure (d) shows that a specified *fraction of the curve can be edited, with the system determining the appropriate editing level.*



be manipulated by dragging a single point: as the editing resolution is lowered, more and more of the curve is affected. This type of control can be supported quite easily by setting δ to be the change in the high-resolution curve at the dragged point $f^n(u_0)$, and using the same equations (3.7) above.

Editing a desired portion of the curve

One difficulty with curve manipulation methods is that their effect often depends on the parameterization of the curve, which does not necessarily correspond to the curve's geometric embedding in an intuitive fashion. The manipulation that we have described so far suffers from this same difficulty: dragging at a particular (possibly fractional) level $\ell = j + t$ on different points along the curve will not necessarily affect constant-length portions of the curve. However, we can use the multiresolution editing control to compensate for this defect in direct manipulation, as follows (Figure 3.7).

Let h be a parameter, specified by the user, that describes the desired length of the editable portion of the curve. The parameter h can be specified using any type of physical units, such as screen pixels, inches, or percentage of the overall curve length. The system computes an appropriate editing level ℓ that will affect a portion of the curve of about h units in length, centered at the point $f^n(u_0)$ being dragged.

We estimate ℓ as follows. For each integer-level editing resolution j, let $h^j(u_0)$ denote the length of $f^n(u)$ affected by editing the curve at the point $f^n(u_0)$. The length $h^j(u_0)$ is easily estimated by scan-converting the curve $f^n(u)$ to determine the approximate lengths of its polynomial segments, and then summing over the lengths of the segments affected when editing the curve at level j and parameter position u_0. Note that the length $h^j(u_0)$ tends to decrease with increasing j, although not necessarily monotonically, depending on the kind of detail present in $f^n(u)$. Next, we define j_- to be the largest value of j with $h^{j_-}(u_0) \geq h$, and j_+ to be the smallest value of j with $h^{j_+}(u_0) \leq h$. We use linear interpolation to find a suitable fractional editing level ℓ that lies between j_- and j_+:

$$\ell := \frac{h - h^{j_+}}{h^{j_-} - h^{j_+}} j_- \; + \; \frac{h^{j_-} - h}{h^{j_-} - h^{j_+}} j_+$$

Finally, by representing ℓ in terms of an integer level j and fractional offset t, we can again apply equation (3.7), followed by equations (3.5) and (3.6), as before. Though in general this construction does not *precisely* cover the desired portion h, in practice it yields an intuitive and meaningful control. Figure 3.7 demonstrates this type of editing for a curve with an extremely non-uniform geometric embedding.

3.4.2 Editing the character of the curve

Another form of editing that is naturally supported by multiresolution curves is one of editing the character of a curve, without affecting its overall sweep. Let C^n be the control points of a curve, and let $C^0, \ldots, C^{n-1}, D^0, \ldots, D^{n-1}$ denote the components of its multiresolution decomposition. Editing the character of the curve is simply a matter of replacing the existing set of detail functions $D^j, \ldots, D^{n-1}$ with some new set $\widehat{D}^j, \ldots, \widehat{D}^{n-1}$, and reconstructing.

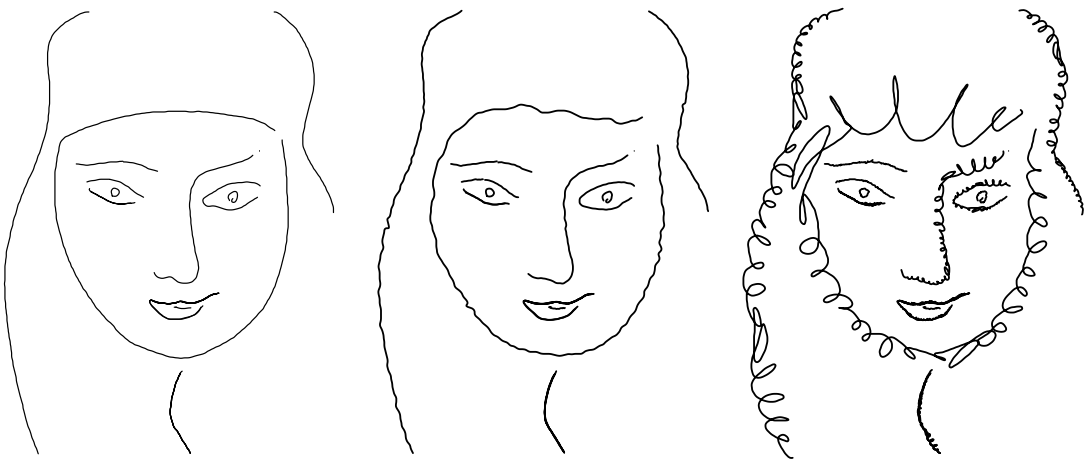


Figure 3.8: Changing the character of a curve without affecting its sweep.

With this approach, we have been able to develop a "curve character library" that contains different detail functions, which can be interchangeably applied to any set of curves. The detail functions in the library have been extracted from hand-drawn strokes; other (for example, procedural) methods of generating detail functions are also possible. Figure 3.8 demonstrates how the character of curves in an illustration can be modified with the same (or different) detail styles. The interactive illustration system used to create this figure is described by Salisbury *et al.*[SABS94].

3.4.3 Orientation of detail

A parametric curve in two dimensions is most naturally represented as two separate functions, one in x and one in y: $f(u) = (f_x(u), f_y(u))$. Thus, it seems reasonable to represent both the control points C^j and detail functions D^j using matrices with separate columns for x and y. However, encoding the detail functions in this manner embeds all of the detail of the curve in a particular xy-orientation. As demonstrated in Figure 3.9, this representation does not always provide the most intuitive control when editing the sweep of the curve.

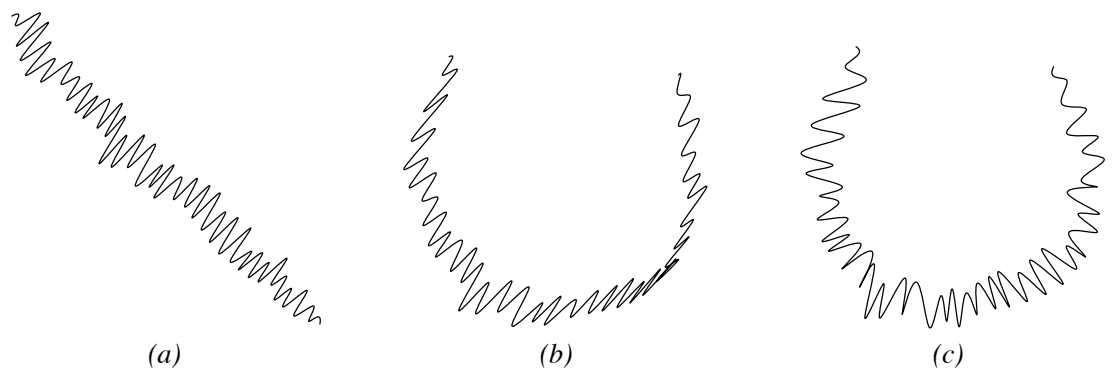


Figure 3.9: Editing the sweep of the original curve (a) using (b) a fixed xy-orientation of detail versus (c) orientation relative to the tangent of the curve.

As an alternative, we employ a method similar to that of Forsey and Bartels [FB88] for representing detail with respect to the tangent and normal to the curve at a coarser level. Specifically, for computing the reference frame for orienting a detail coefficient d_i^j, we use the tangent and normal of the curve $f^{j-1}(u_0)$ at a parameter position u_0 corresponding to the maximum value of the wavelet $\psi_i^j(u)$. Note that the curve $f(u)$ is no longer a simple linear combination of the scaling functions Φ^0 and wavelets Ψ^j; instead, a change of coordinates must be performed at each level of reconstruction for the wavelet coefficients D^j. However, this process is linear in the number of control points, so it does not increase the computational complexity of the algorithm.

We have experimented with both normalized and unnormalized versions of the reference frame; the two alternative versions yield different but equally reasonable behavior. Figure 3.8 uses the unnormalized tangents whereas the rest of the figures in this chapter use normalized tangents.

The choice of whether to use fixed orientation of detail versus orientation relative to the tangent frame often depends on the semantics of the curve. For example, trees on a hillside should probably point in an upward direction, regardless of the the slope of the hill, whereas hair on a body should probably always point in an outward direction.

3.5 Scan conversion and curve compression

Using "curve character libraries" and other multiresolution editing features, it is easy to create very complex curves with hundreds or potentially thousands of control points. In many cases, such complicated curves are printed in a very small form. Conventional scan conversion methods that use all the complexity of these curves are wasteful, both in terms of the network traffic to send such large files to the printer, and in terms of the processing time required by the printer to render curves of many control points within a few square pixels. We therefore explore a form of curve compression that is suitable for the purposes of scan conversion. The algorithm produces an approximate curve that has a guaranteed maximum error, in terms of printer pixels, from the original curve. However, it does not require any particular continuity constraints, as are usually required in data-fitting applications.

As discussed in Section 3.3, the simple removal of wavelet coefficients can be used to achieve a least-squares, or L^2, error metric between an original curve and its approximate versions. However, for scan conversion, an L^2 error metric is not very useful for measuring the degree of approximation: an approximate curve $\widetilde{f}(u)$ can be arbitrarily far from its original curve $f^n(u)$ and still achieve a particular L^2 error bound, as long as it deviates from the original over a small enough segment. In order to scan convert a curve to some guaranteed precision—measured, say, in terms of maximum deviation in printer pixels—we need to use an L^∞ norm on the error. There are many ways to achieve such a bound. The method described here is a simple and fast one, although methods with higher compression ratios are certainly possible.

Let s_i^j (with $0 \leq i \leq 2^j - 1$) be a segment of the cubic B-spline curve $f^j(u)$, defined by the four control points $c_i^j, \ldots, c_{i+3}^j$. Note that each segment s_i^j corresponds to exactly two segments s_{2i}^{j+1} and s_{2i+1}^{j+1} at level $j+1$. Our objective is to build a new approximating curve $\widetilde{f}(u)$ for $f(u)$ by choosing different segments at different levels such that $||\widetilde{f}(u) - f^n(u)||_\infty$ is less than some user-specified ϵ.

Assume, for the moment, that we have some function $ErrBound(s_i^j)$ that returns a bound

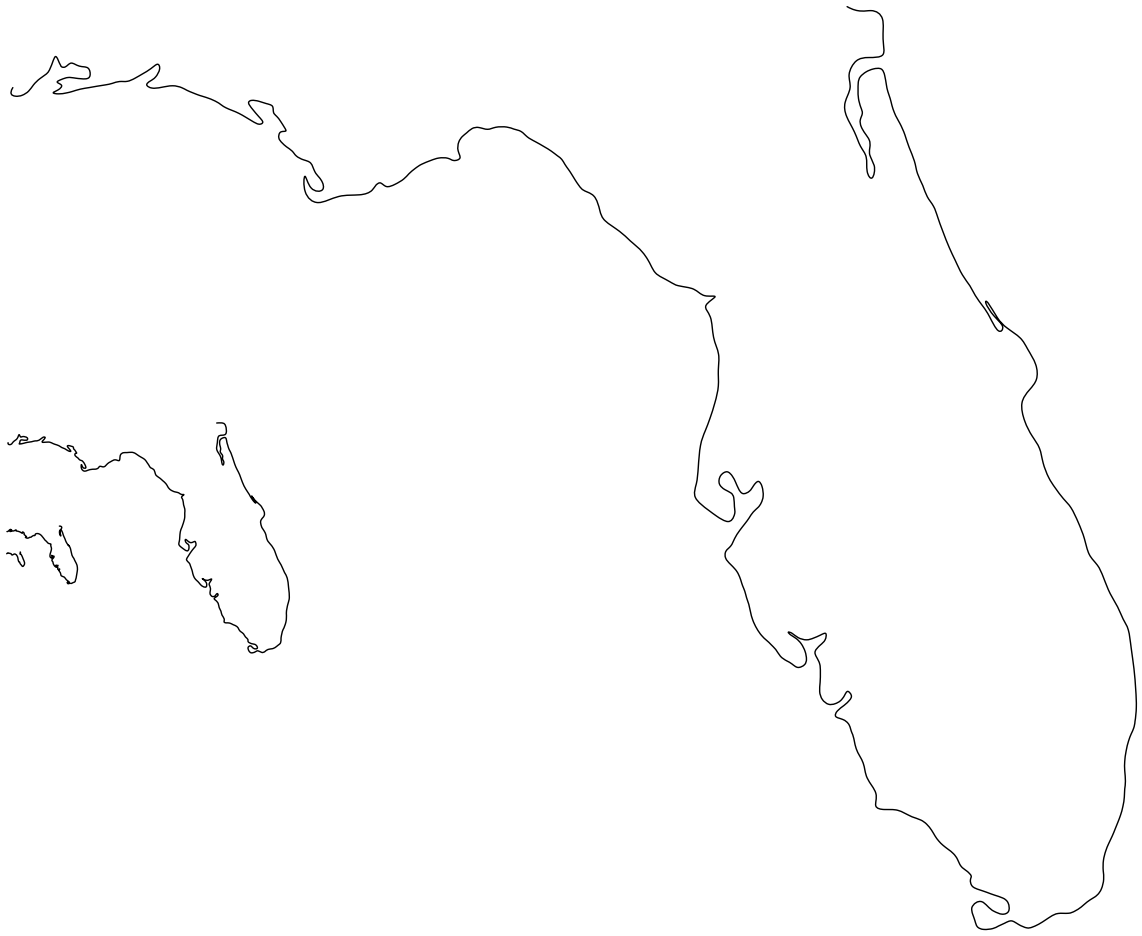


Figure 3.10: Scan-converting a curve to within a guaranteed maximum error tolerance. From left to right, the figures used 5%, 21%, 46%, and 78% of the possible number of Bézier segments. Error is less than 1/200 inch.

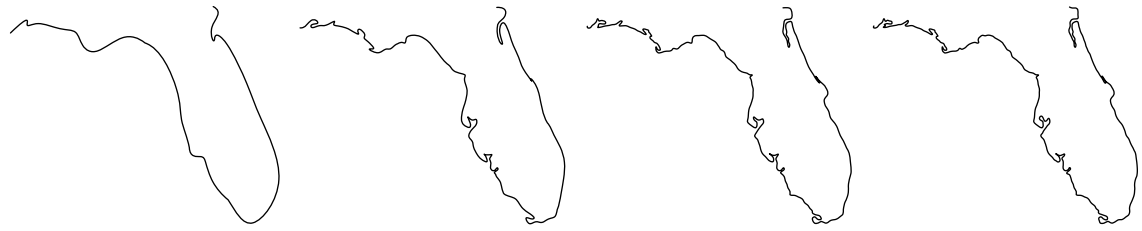


Figure 3.11: Same curves as above, but drawn at constant size.

```
procedure DrawSegment(s_i^j):
    if ErrBound(s_i^j) < ε then
        Output segment s_i^j as a portion of f̃(u)
    else
        DrawSegment(s_{2i}^{j+1})
        DrawSegment(s_{2i+1}^{j+1})
    end if
end procedure

procedure ErrBound(s_i^j):
    if j = n then
        return 0
    else
        return max{ErrBound(s_{2i}^{j+1})+δ_{2i}^{j+1}, ErrBound(s_{2i+1}^{j+1})+δ_{2i+1}^{j+1}}
    end if
end procedure
```

Figure 3.12: Pseudocode to approximate a curve for scan-conversion.

on the L^∞ error incurred from using the segment s_i^j of some approximate curve $f^j(u)$ in place of the original segments of $f^n(u)$ to which it corresponds. We can scan-convert a curve to within any error tolerance ϵ by passing to the recursive routine *DrawSegment*() in Figure 3.12 the single segment s_0^0 corresponding to the lowest-level curve $f^0(u)$. This routine recursively divides the segment to varying levels so that the collection of segments it produces approximates the curve to within ϵ.

To construct the *ErrBound*() routine, let M^j be the B-spline-to-Bézier-basis conversion matrix [BBB87] for curves with $2^j + 3$ control points, and let E^j be a column vector with entries e_j^i defined by

$$E^j \;:=\; M^j\, Q^j\, D^{j-1} \tag{3.8}$$

The vector E^j provides a measure of the distance that the Bézier control points migrate when reconstructing the more detailed curve at level j from the approximate curve at level

$j-1$. Since Bézier curves are contained within the convex hull of their control points, the magnitudes of the entries of E^j provide conservative bounds on approximations to the curve due to truncating wavelet coefficients.

A bound δ_i^j on the L^∞ error incurred by replacing segment s_i^j with its approximation at level $j-1$ is given by

$$\delta_i^j \leq \max_{i \leq k \leq i+3} \left\{ \left\| e_k^j \right\|_2 \right\} \tag{3.9}$$

The *ErrBound*() routine can then be described recursively as it appears in Figure 3.12.

An efficient implementation of the *ErrBound*() routine would use memoization [CLR90] (rather than recursion) in order to avoid recomputing error bounds. In practice, the routine is fast enough in its recursive form that we have not found this optimization to be necessary, at least for scan converting curves with hundreds of control points.

The approximate curve $\widetilde{f}(u)$ is described by a set of Bézier segments, which we use to generate a PostScript file [Ado90]. Note that the scan-conversion algorithm, as described, produces approximate curves $\widetilde{f}(u)$ that are not even C^0 continuous where two segments of different levels abut. Since we are only concerned with the absolute error in the final set of pixels produced, relaxing the continuity of the original curve is reasonable for scan conversion. We can achieve C^0 continuity, however, without increasing the prescribed error tolerance, by simply averaging together the end control points for adjacent Bézier segments as a post-process. We have found that these C^0 curves look slightly better than the discontinuous curves; they also have a more compact representation in PostScript. Figures 3.10 and 3.11 demonstrate compression of the same curve rendered at different sizes.

3.6 Summary and extensions

This chapter describes a wavelet-based multiresolution representation for endpoint-interpolating B-spline curves, and shows how this single representation supports a variety of display and editing operations in a simple and efficient manner.

We have considered (but not implemented) a number of extensions to multiresolution curves, among them:

Handling discontinuities. An important extension is to generalize the multiresolution curve representation and editing operations to respect discontinuities of various orders that have been intentionally placed into a curve by the designer. This extension would allow the techniques to be applied more readily to font design, among other applications. One approach is to try using the multiresolution analysis defined on non-uniform B-splines by Dæhlen and Lyche [DL92]. A challenge associated with employing their multiresolution analysis in a curve editor is that it is difficult to choose levels in the filterbank hierarchy at which discontinuities should vanish.

Sparse representations. Our algorithms have so far used only *complete* wavelet decompositions of the curve's original control points. However, in order to support curve editing at an arbitrarily high resolution, it would be convenient to have a mechanism in place for extending the wavelet representation to a higher level of detail in certain higher-resolution portions of the curve than in others. One such sparse representation might use pruned binary trees to keep track of the various wavelet coefficients at different levels of refinement, in a manner very similar to the one used by Berman *et al*. for representing multiresolution images [BBS94].

Textured strokes. For illustrations, it is useful to associate other properties with curves, such as color, thickness, texture, and transparency, as demonstrated by Hsu and Lee [HL94]. These quantities may be considered extra dimensions in the data associated with each control point. Much of the machinery for multiresolution editing should be applicable to such curves. As a preliminary test of this idea, we have extended our curve editor with a

Figure 3.13: Two curves of varying thickness.

thickness dimension. The thickness along the curve is governed by the thicknesses defined at the control points. It is possible to modify this parameter at any level of resolution, just as one edits the position of the curve. Figure 3.13 shows two curves with varying thickness.

Below many of the figure captions in this dissertation appears a roughly horizontal stroke separating the figure from the rest of the page. Each of these strokes is unique, created by choosing among four basic shapes, four detail functions, and four thickness functions.

Chapter 4

MULTIRESOLUTION IMAGE QUERYING

4.1 Introduction

With the explosion of desktop publishing, the ubiquity of color scanners and digital media, and the advent of the World Wide Web, people now have easy access to tens of thousands of digital images. This trend is likely to continue, providing more and more people with access to increasingly large image databases.

As the size of these databases grows, traditional methods of interaction break down. For example, while it is relatively easy for a person to quickly look over a few hundred "thumbnail" images to find a specific image, it is much harder to locate that image among several thousand. Exhaustive search quickly breaks down as an effective strategy when the database becomes sufficiently large.

One commonly-employed searching strategy is to index the image database with keywords. However, such an approach is also fraught with difficulties. First, it requires a person to manually tag all the images with keys, a time-consuming task. Second, as Niblack *et al*. point out [NBE+93], this keyword approach has the problem that some visual aspects are inherently difficult to describe, while others are equally well described in many different ways. In addition, it may be difficult for the user to guess which visual aspects have been indexed.

This chapter explores an alternative strategy for searching an image database, in which the query is expressed either as a low-resolution image from a scanner or video camera,

or as a rough sketch of the image painted by the user. This basic approach to image querying has been referred to in a variety of ways, including "query by content" [BEN+93, FBF+94, NBE+93], "query by example" [HK92, Kat92, KKOH92], "similarity retrieval" [GS93, KZT93, LC93, PSTT93, TC94] and "sketch retrieval" [KKOH92]. Note that this type of content-based querying can also be applied in conjunction with keyword-based querying or any other existing approach.

Several factors make this problem difficult to solve. The "query" image is typically very different from the "target" image, so the retrieval method must allow for some distortions. If the query is scanned, it may suffer artifacts such as color shift, poor resolution, dithering effects, and misregistration. If the query is painted, it is limited by perceptual error in both shape and color, as well as by the artistic prowess and patience of the user. For these reasons, straightforward approaches such as L^1 or L^2 image metrics are not very effective in discriminating the target image from the rest of the database. In order to match such imperfect queries more effectively, a kind of "image querying metric" must be developed that accommodates these distortions and yet distinguishes the target image from the rest of the database. In addition, the retrieval should ideally be fast enough to handle databases with tens of thousands of images at interactive rates.

In this chapter we show how a wavelet decomposition of the query and database images can be used to match a content-based query both quickly and effectively, in spite of the aforementioned difficulties. The input to our retrieval method is a sketched or scanned image, intended to be an approximation to the image being retrieved. Since the input is only approximate, the approach we have taken is to present the user with a small set of the most promising target images as output, rather than with a single "correct" match. We have found that 20 images (the number of slides on a slide sheet) are about the most that can be visually scanned quickly and reliably by a user in search of the target.

In order to perform this ranking, we define an *image querying metric* that makes use of truncated, quantized versions of the wavelet decompositions, which we call *signatures*. The signatures contain only the most significant information about each image. The image

querying metric essentially compares how many significant wavelet coefficients the query has in common with potential targets. We show how the metric can be tuned, using statistical techniques, to discriminate most effectively for different types of content-based image querying, such as scanned or hand-painted images. We also present a novel database organization for computing this metric extremely fast. (Our system processes a 128×128 image query on a database of 20,000 images in under 1/2 second on an SGI Indy R4400; by contrast, searching the same database using an L^1 metric takes over 14 minutes.) Finally, we evaluate the results of applying our tuned image querying metric on hundreds of queries in databases of 1000 and 20,000 images.

The content-based querying method we describe has applications in many different domains, including graphic design, architecture [SSG92], TV production [SI90], multimedia [SZ94], ubiquitous computing [Wei93], art history [Kat92], geology [SDOW93], satellite image databases [KZT93], and medical imaging [KC94]. For example, a graphic designer may want to find an image that is stored on her own system using a painted query. She may also want to find out if a supplier of ultra-high-resolution digital images has a particular image in its database, using a low-resolution scanned query. In the realm of ubiquitous computing, a computer may need to find a given document in its database, given a video image of a page of that document, scanned in from the real-world environment. In all of these applications, improving the technology for content-based querying is an important and acknowledged challenge [GMG+92].

4.1.1 Related work

Our method makes use of a wavelet decomposition of the query and database images in order to form the image querying metric. Wavelet transforms of images have been employed in a number of areas. For example, Berman *et al*. [BBS94] used a wavelet representation of images to perform efficient painting and compositing. Much of the work on wavelet transforms for images has been applied toward image compression [Wic94]. The method

we use to distill signatures from images shares many features with wavelet-based lossy image compression. One might even think of the signatures as highly compressed images, although they are so compressed that it would be virtually impossible to reconstruct the original image from its signature.

Previous approaches to content-based image querying have applied such properties as color histograms [Swa93], texture analysis [KZL94], and shape features like circularity and major-axis orientation of regions in the image [GZCS94], as well as combinations of these techniques.

One of the most notable systems for querying by image content, called "QBIC," was developed at IBM [NBE+93] and is now available commercially. The emphasis in QBIC is in allowing a user to compose a query based on a variety of different visual attributes; for example, the user might specify a particular color composition (x% of color 1, y% of color 2, etc.), a particular texture, some shape features, and a rough sketch of dominant edges in the target image, along with relative weights for all of these attributes. The QBIC system also allows users to annotate database images by outlining key features to search on. By contrast, the emphasis in our work is in searching directly from a query image, without any further specifications from the user—either about the database images or about the particulars of the search itself.

The work of Hirata and Kato [HK92] is perhaps the most like our own in its style of user interaction. In their system, called "query by visual example" (QVE), edge extraction is performed on user queries. These edges are matched against those of the database images in a fairly complex process that allows for corresponding edges to be shifted or deformed with respect to each other.

It is difficult to directly compare our results with these previous methods, since running times are rarely reported, and since the number of tests reported and the size of the databases being searched have generally been quite small. From the little information that has been provided, it appears that the success rate of our method is at least as good as that of other systems that work from a simple user sketch.

To our knowledge, we are the first to use a multiresolution approach for solving this problem. Among other advantages, our approach allows queries to be specified at any resolution (potentially different from that of the target); moreover, the running time and storage of our method are independent of the resolutions of the database images. In addition, the signature information required by our algorithm can be extracted from a wavelet-compressed version of the image directly, allowing the signature database to be created conveniently from a set of compressed images. Finally, our algorithm is much simpler to implement and to use than most previous approaches.

4.1.2 Overview

In the next section, we discuss our approach to image querying in more detail and define an "image querying metric" that can be used for searching with imprecise queries. Section 4.3 describes the image querying algorithm in detail; the algorithm is simple enough that almost all of the code is included here. Section 4.4 describes the application we have built on top of this algorithm, and gives some examples of its use. Section 4.5 describes the results of our tests, and Finally, Section 4.6 summarizes the results and discusses some extensions to this work.

4.2 Developing a metric for image querying

Consider the problem of computing the distance between a query image Q and a potential target image T. The most obvious metrics to consider are the L^1 or L^2 norms:

$$||Q,T||_1 = \sum_{i,j} |Q[i,j] - T[i,j]| \tag{4.1}$$

$$||Q,T||_2 = \left(\sum_{i,j} (Q[i,j] - T[i,j])^2 \right)^{1/2} \tag{4.2}$$

However, these metrics are not only expensive to compute, but they are also fairly

ineffective when it comes to matching an inexact query image in a large database of potential targets. For example, in our experience with scanned queries (described in Section 4.5), the L^1 and L^2 error metrics rank their intended target image in the highest 1% of the database only 3% of the time. (This rank is computed by sorting the database according to its L^1 or L^2 distance from the query, and evaluating the intended target's position in the sorted list.)

On the other hand, the target of the query image is almost always readily discernible to the human eye, despite such potential artifacts as color shifts, misregistration, dithering effects, and distortion (which, taken together, account for the relatively poor performance of the L^1 and L^2 metrics). The solution, it would seem, is to try to find an image metric that is "tuned" for the kind of errors present in image querying; that is, we would like a metric that counts primarily those types of differences that a human would use for discriminating images, but that gives much less weight to the types of errors that a human would ignore for this task. This problem is related to that of finding a good perceptual error metric for images, although, to our knowledge, most previous work in this area has been devoted primarily to minimizing image artifacts, for example, in image compression [JJS93, RAFH92, TH94].

Since there is no obvious "correct" metric to use for image querying, we are faced with the problem of constructing one from scratch, using (informed) trial and error. The rest of this section describes the issues we addressed in developing our image querying metric.

4.2.1 A multiresolution approach

Our goal was to construct an image metric that is fast to compute, that requires little storage for each database image, and that improves significantly upon the L^1 or L^2 metrics in discriminating the targets of inexact queries. For several reasons, we hypothesized that a two-dimensional wavelet decomposition of the images [SDS96] would provide a good foundation on which to build such a metric:

- Wavelet decompositions allow for very good image approximation with just a few coefficients. This property has been exploited for lossy image compression [DJL92].

- Wavelet decompositions naturally encode edge information [MZ92]. Edges are likely to be among the key features of a user-painted query.
- The coefficients of a wavelet decomposition provide information that is independent of the original image resolution. Thus, a wavelet-based scheme allows the resolutions of the query and the target to be effectively decoupled.
- As shown in Chapter 2, wavelet decompositions are fast and easy to compute, requiring linear time in the size of the image and very little code.

4.2.2 Components of the metric

Given that we wish to use a wavelet approach, there are a number of issues that still need to be addressed:

1. **Color space.** We need to choose a color space in which to represent the images and perform the decomposition. (The same issue arises for L^1 and L^2 image metrics.) We decided to try a number of different color spaces: RGB, HSV, and YIQ. Ultimately, YIQ turned out to be the most effective of the three for our data, as reported in Figure 4.5 of Section 4.5.

2. **Wavelet type.** We chose the Haar wavelets described in Section 2.3, both because they are fastest to compute and simplest to implement. In addition, user-painted queries (at least with our simple interface) tend to have large constant-colored regions, which are well represented by this basis. One drawback of the Haar basis for lossy compression is that it tends to produce blocky image artifacts. In our application, however, the results of the decomposition are never viewed, so these artifacts are of no concern. We have not experimented with other wavelet bases; others may work as well as or better than Haar (although will undoubtedly be slower).

3. **Decomposition type.** We need to choose either a "standard" or "non-standard" type of two-dimensional wavelet decomposition [BCR91, SDS96]. In the Haar basis the non-standard basis functions are square, whereas the standard basis functions are rectangular. We would therefore expect the non-standard basis to be better at identifying features that are about as wide as they are high, and the standard basis to work best for images containing lines and other rectangular features. As reported in Figure 4.5 of Section 4.5, we tried both types of decomposition with all three color spaces, and found that the standard basis works best on our data.

4. **Truncation.** For a 128×128 image, there are $128^2 = 16,384$ different wavelet coefficients for each color channel. Rather than using all of these coefficients in the metric, it is preferable to "truncate" the sequence, keeping only the coefficients with largest magnitude. This truncation both accelerates the search for a query and reduces storage for the database. Surprisingly, truncating the coefficients also appears to improve the discriminatory power of the metric, probably because it allows the metric to consider only the most significant features—which are the ones most likely to match a user's painted query—and to ignore any mismatches in the fine detail, which the user, most likely, would have been unable to accurately re-create. We experimented with different levels of truncation and found that storing the 60 largest-magnitude coefficients in each channel worked best for our painted queries, while 40 coefficients worked best for our scanned queries.

5. **Quantization.** Like truncation, the quantization of each wavelet coefficient can serve several purposes: speeding the search, reducing the storage, and actually improving the discriminatory power of the metric. The quantized coefficients retain little or no data about the precise magnitudes of major features in the images; however, the mere presence or absence of such features appears to have more discriminatory power for image querying than the features' precise magnitudes. We found that quantizing each significant coefficient to just two levels—+1, representing large positive coefficients;

or -1, representing large negative coefficients—works remarkably well. This simple classification scheme also allows for a very fast comparison algorithm, as discussed in Section 4.3.

6. **Normalization.** The normalization of the wavelet basis is related to the magnitude of the computed wavelet coefficients: as the amplitude of each basis function increases, the size of that basis function's corresponding coefficient decreases accordingly. We chose a normalization factor that makes all wavelets orthonormal to each other.[1] This normalization factor has the effect of emphasizing differences mostly at coarser scales. Because changing the normalization factor requires rebuilding the entire database of signatures, we have not experimented further with this degree of freedom.

4.2.3 The "image querying metric"

In order to write down the resulting metric, we must introduce some notation. First, let us now think of Q and T as representing just a single color channel of the wavelet decomposition of the query and target images. Let $Q[0,0]$ and $T[0,0]$ be the *scaling function coefficients* corresponding to the overall average intensity of that color channel. Further, let $\tilde{Q}[i,j]$ and $\tilde{T}[i,j]$ represent the $[i,j]$-th *truncated, quantized wavelet coefficients* of Q and T; these values are either -1, 0, or $+1$. For convenience, we will define $\tilde{Q}[0,0]$ and $\tilde{T}[0,0]$, which do not correspond to any wavelet coefficient, to be 0.

A suitable metric for image querying can then be written as

$$||Q,T|| \;=\; w_{0,0}|Q[0,0]-T[0,0]| \;+\; \sum_{i,j} w_{i,j}\left|\tilde{Q}[i,j]-\tilde{T}[i,j]\right|$$

where $w_{i,j}$ are weights. We can simplify this metric in a number of ways.

First, we have found the metric to be just as effective if the difference between the wavelet coefficients $\left|\tilde{Q}[i,j]-\tilde{T}[i,j]\right|$ is replaced by $(\tilde{Q}[i,j] \neq \tilde{T}[i,j])$, where the expression

[1] In contrast, the Haar basis that appear in Figure 2.2 is normalized so that all basis functions have the same height, for schematic reasons.

($a \neq b$) is interpreted as 1 if $a \neq b$, and 0 otherwise. This expression will be faster to compute in our algorithm.

Second, we would like to group terms together into "buckets" so that only a small number of weights $w_{i,j}$ need to be determined experimentally. We group the terms according to the scale of the wavelet functions to which they correspond, using a simple bucketing function $bin(i,j)$, described in detail in Section 4.3.

Finally, in order to make the metric even faster to evaluate over many different target images, we only consider terms in which the *query* has a non-zero wavelet coefficient $\tilde{Q}[i,j]$. A potential benefit of this approach is that it allows for a query without much detail to match a very detailed target image quite closely; however, it does not allow a detailed query to match a target that does not contain that same detail. We felt that this asymmetry might better capture the form of most painted image queries. (Note that this last modification technically disqualifies our "metric" from being a metric at all, since metrics, by definition, are symmetric. Nevertheless, for lack of a better term, we will continue to use the word "metric" in the rest of this chapter.)

Thus, the final "L^q" *image querying metric* $||Q,T||_q$ is given by

$$w_0|Q[0,0]-T[0,0]| + \sum_{i,j\,:\,\tilde{Q}[i,j]\neq 0} w_{bin(i,j)} \left(\tilde{Q}[i,j] \neq \tilde{T}[i,j]\right) \tag{4.3}$$

The weights w_b in equation (4.3) provide a convenient mechanism for tuning the metric to different databases and styles of image querying. The actual weights we use in our application are given in Section 4.3.2, and the method we used to compute these weights is described in Section 4.3.3.

4.2.4 Fast computation of the image querying metric

To actually compute the L^q metric over a database of images, it is generally quicker to count the number of *matching* $\tilde{Q}$ and $\tilde{T}$ coefficients, rather than *mismatching* coefficients, since we expect the vast majority of database images not to match the query image well at all. It is

therefore convenient to rewrite the summation in (4.3) in terms of an "equality" operator ($a = b$), which evaluates to 1 when $a = b$, and 0 otherwise. Using this operator, the summation

$$\sum_{i,j:\tilde{Q}[i,j]\neq 0} w_k \left(\tilde{Q}[i,j] \neq \tilde{T}[i,j]\right)$$

in equation (4.3) can be rewritten as

$$\sum_{i,j:\tilde{Q}[i,j]\neq 0} w_k - \sum_{i,j:\tilde{Q}[i,j]\neq 0} w_k \left(\tilde{Q}[i,j] = \tilde{T}[i,j]\right)$$

Since the first part of this expression $\sum w_k$ is independent of $\tilde{T}$, we can ignore it for the purposes of ranking the different target images in L^q. It therefore suffices to compute the expression

$$w_0 |Q[0,0] - T[0,0]| - \sum_{i,j:\tilde{Q}[i,j]\neq 0} w_{bin(i,j)} \left(\tilde{Q}[i,j] = \tilde{T}[i,j]\right) \tag{4.4}$$

This expression is a weighted sum of the difference in the average color between Q and T, and the number of stored wavelet coefficients of T whose indices and signs match those of Q.

4.3 The algorithm

The final algorithm is a straightforward embodiment of the L^q metric as given in equation (4.4), applied to the problem of finding a given query in a large database of images. The complexity of the algorithm is linear in the number of database images. The constant factor in front of this linear term is small, as discussed in Section 4.5.5.

At a high level, the algorithm can be described as follows: In a preprocessing step, we perform a standard two-dimensional Haar wavelet decomposition [BCR91, SDS96] of every image in the database, and store just the overall average color and the indices and signs of the m largest-magnitude wavelet coefficients. The indices for all of the database images are then organized into a single data structure in the program that optimizes searching. Then, for each query image, we perform the same wavelet decomposition, and again throw away

all but the average color and the largest m coefficients. The score for each target image T is then computed by evaluating expression (4.4).

The rest of the section describes this algorithm in more detail.

4.3.1 Preprocessing step

In Section 2.3 we described analysis for one-dimensional Haar wavelets as a series of matrix multiplications. Here we provide a simple algorithm that computes the standard *two*-dimensional Haar transform. It involves a one-dimensional decomposition on each row of the image, followed by a one-dimensional decomposition on each column of the result.

The procedure *DecomposeArray*() in Figure 4.1 performs this one-dimensional decomposition on an array A of h elements, with h a power of two. In the pseudocode, the entries of A are assumed to be 3-dimensional color components, each in the range $[0,1]$. The various arithmetic operations are performed on the separate color components individually. This procedure performs algorithmically the series of matrix multiplications that appear schematically in the filter bank of Figure 2.1, for the special case of Haar wavelets.

An entire $r \times r$ image T can thus be decomposed using the *DecomposeImage*() procedure in Figure 4.1. After the decomposition process, the entry $T[0,0]$ is proportional to the average color of the overall image, while the other entries of T contain the wavelet coefficients. (These coefficients are sufficient for reconstructing the original image T, although we will have no need to do so in this application.)

Finally, we store only $T[0,0]$ and the indices and signs of the largest m wavelet coefficients of T. To optimize the search process, the remaining m wavelet coefficients for *all* of the database images are organized into a set of six arrays, called the *search arrays*, with one array for every combination of sign (+ or −) and color channel (such as R, G, and B).

For example, let $\mathcal{D}_+^c$ denote the "positive" search array for the color channel c. Each element $\mathcal{D}_+^c[i,j]$ of this array contains a list of all images T having a large positive wavelet coefficient $T[i,j]$ in color channel c. Similarly, each element $\mathcal{D}_-^c[i,j]$ of the "negative"

```
procedure DecomposeArray(A : array[0..h − 1] of color):
    A ← A/√h
    while h > 1 do:
        h ← h/2
        for i ← 0 to h − 1 do:
            A′[i] ← (A[2i] + A[2i + 1])/√2
            A′[h + i] ← (A[2i] − A[2i + 1])/√2
        end for
        A ← A′
    end while
end procedure

procedure DecomposeImage(T : array[0..r − 1, 0..r − 1] of color):
    for row ← 0 to r − 1 do:
        DecomposeArray(T[row, 0..r − 1])
    end for
    for col ← 0 to r − 1 do:
        DecomposeArray(T[0..r − 1, col])
    end for
end procedure
```

Figure 4.1: Pseudocode to compute the wavelet decomposition of an image.

search array points to a list of images with large negative coefficients in c.

These six arrays are used to speed the search for a particular query, as described in the next section. In our implementation, the search arrays are created as a preprocess for a given database and stored on disk. We use a small stand-alone program to add new images to the database incrementally. This program performs the wavelet decomposition for each new image, finds the largest m coefficients, and augments the database search arrays accordingly.

4.3.2 Querying

The querying step is straightforward. For a given query image Q, we perform the same wavelet decomposition described in the previous section. Again, we keep just a signature—

```
function ScoreQuery(Q : array[0..r − 1, 0..r − 1] of color; m : int):
    DecomposeImage(Q)
    Initialize scores[i] ← 0 for all i
    for each color channel c do:
        for each database image T do:
            scores[index(T)] += w^c[0] ∗ |Q^c[0,0] − T^c[0,0]|
        end for
        Q̃ ← TruncateCoefficients(Q, m)
        for each non-zero coefficient Q̃^c[i,j] do
            if Q̃^c[i,j] > 0 then
                list ← D^c_+[i,j]
            else
                list ← D^c_−[i,j]
            end if
            for each element ℓ of list do
                scores[index(ℓ)] −= w^c[bin(i,j)]
            end for
        end for
    end for
    return scores
end function
```

Figure 4.2: Pseudocode to compute the score of each database image.

the overall average color and the indices and signs of the largest m coefficients in each color channel. Now we just need to find which images in the database most closely match the query based on this signature.

Figure 4.2 contains pseudocode for the function *ScoreQuery*(), which computes scores for all the images in the database as follows. We loop through each color channel c. First, we compute the differences between the query's average intensity in that channel $Q^c[0,0]$ and those of the database images. Next, for each of the m non-zero, truncated wavelet coefficients $\tilde{Q}^c[i,j]$, we search through the list corresponding to those database images containing the same large-magnitude coefficient and sign, and update each of those image's scores.

Table 4.1: Weights $w^c[b]$ used in our image query application.

	Painted			Scanned		
b	$w^Y[b]$	$w^I[b]$	$w^Q[b]$	$w^Y[b]$	$w^I[b]$	$w^Q[b]$
0	*4.04*	*15.14*	*22.62*	*5.00*	*19.21*	*34.37*
1	*0.78*	*0.92*	*0.40*	*0.83*	*1.26*	*0.36*
2	*0.46*	*0.53*	*0.63*	*1.01*	*0.44*	*0.45*
3	*0.42*	*0.26*	*0.25*	*0.52*	*0.53*	*0.14*
4	*0.41*	*0.14*	*0.15*	*0.47*	*0.28*	*0.18*
5	*0.32*	*0.07*	*0.38*	*0.30*	*0.14*	*0.27*

The function $bin(i,j)$ provides a way of grouping different coefficients into a small number of bins, with each bin weighted by some constant $w[b]$. For a given set of bins, the best weights $w[b]$ can be found experimentally. The larger the training set, the more weights that can be used. The size of our training set was sufficient for 18 weights: 6 per color channel.

In our implementation, we use the function

$$bin(i,j) \quad := \quad \min\left\{\max\left\{\lfloor \log_2 i \rfloor, \lfloor \log_2 j \rfloor\right\}, 5\right\}$$

which corresponds to the level of the filterbank at which the wavelet coefficient was computed, but groups all of the very high frequency wavelets into bin 5.

For our database of images, a good set of weights (using the YIQ color space and standard decomposition) is shown in Table 4.1. These weights were found as part of the training process described in Section 4.5.1, using a statistical model outlined in the following section. All scaling function coefficients in our implementation are reals in the range $[0,1]$, so their differences tend to be smaller than the differences of the truncated, quantized wavelet coefficients. Thus, the weights on the scaling functions $w[0]$ have relatively large magnitudes because they generally multiply smaller quantities.

As a final step, our algorithm examines the list of scores, which may be positive or negative. The smallest (typically, the most negative) scores are considered to be the closest

matches. We use a "Heap-Select" algorithm [PFTF92] to find the 20 closest matches in linear time.

4.3.3 Tuning the weights of the metric

Our L^q metric (4.3) involves a linear combination of terms. In this section, we discuss how a good set of weights w_k for these terms can be found experimentally. The weights shown in Table 4.1 were found using the method we will describe.

Suppose we have collected a database of images and a group of representative queries for known targets in the database. Now we want to find a set of weights that causes our L^q metric to discriminate the correct target images from the database, given the example queries.

The most straightforward approach for finding these weights would be to use some form of multidimensional continuous optimization over these variables, such as Powell's method [PFTF92], using an evaluation function like "the number of queries that ranked their intended target in the upper 1% of the database." The difficulty is that this kind of evaluation function is fairly slow to compute (on the order of many seconds), since we would like to perform the evaluation over a large number of queries.

An alternative approach is to assume a regression model and to perform a kind of least-squares fit to the data [PFTF92]. For each pair ℓ of query and target images, we record an equation of the form:

$$r_\ell \;=\; v \;+\; \sum_k w_k t_{k,\ell} \;+\; u_\ell$$

where r_ℓ is either 1 or 0, depending on whether or not the query and target are intended to match; $t_{k,\ell}$ is the sum of the terms of equation (4.3) in bucket k; variables v and w_k are the unknowns to be found by the least-square fit; and u_ℓ is an error term to make the equality hold.

However, there are a number of problems with this method. The first problem is primarily an aesthetic one: once we have computed the weights v and w_k, they will give results

that are in general neither 0 nor 1—and in fact, may not even lie in the interval $[0,1]$. In that case, we are left with the problem of interpreting what these other values should mean. The second problem is even more serious. The difficulty is that when collecting the data for tuning the weights, it is much easier to create data for mismatches than for matches, since in a database of 1000 images, every query corresponds to 999 mismatches and only a single match. If we use all of this data, however, the least-squares fit will tend to give weights that are skewed toward finding mismatches, since the best least-squares fit to the data will be to make every query–target pair score very close to 0. The alternative, using equal-sized sets of matched and mismatched image pairs, means throwing out a lot of perfectly useful and inexpensive data.

For these reasons, we use an approach from statistics called the *logit* model [Mad92]. In the logit model, we assume a regression model of the form:

$$r_\ell^* \;=\; v \;+\; \sum_k w_k t_{k,\ell} \;+\; u_\ell$$

where r_ℓ^* is called a "latent variable," which is not observed directly. Observed instead is a dummy variable r_ℓ, defined by

$$r_\ell \;=\; \begin{cases} 1 & \text{if } r_\ell^* > 0 \\ 0 & \text{otherwise} \end{cases}$$

The idea behind the logit model is that there exists some underlying continuous variable r_ℓ^* (such as the "perceptual closeness" of two images Q and T) that is difficult to measure directly. The continuous variable r_ℓ^* determines a binary outcome r_ℓ (such as "image T is the intended target of the query Q"), which is easily measured. The logit model provides weights w_k, which can be used to compute the *probability* that a given r_ℓ^* produces a positive outcome r_ℓ.

Indeed, under the assumptions of the logit model, the probability P_ℓ that the query–target pair ℓ is indeed a match, is given by

$$P_\ell \;=\; F\left(v + \sum_k w_k t_{k,\ell}\right) \qquad \text{where} \qquad F(x) \;=\; \frac{e^x}{1+e^x}$$

Once the weights are found, since $F(x)$ is monotonic and v is constant for all query–target pairs ℓ, it suffices to compute the expression

$$\sum_k w_k t_{k,\ell}$$

in order to rank the targets in order of decreasing probability of a match.

To compute the weights, we use the logit procedure in SAS [SAS89]. It takes SAS about 30 seconds on an IBM RS/6000 to find appropriate weights for an input of 85 matches and 8500 (randomly chosen) mismatches. (Our database, as well as the 85 queries, will be described in Section 4.5.) While these weights are not necessarily optimal with respect to our preferred evaluation function, they appear to give very good results, and they can be computed much more quickly than performing a multidimensional continuous optimization directly.

4.4 The application

We have built a simple interactive application that incorporates our image querying algorithm. The program is written in C^{++}, using OpenGL and Motif. It runs on SGI workstations.

A screen capture of the running application is shown in Figure 4.3. The user paints an image query in the large rectangular area on the left side of the application window. When the query is complete, the user presses the "Match" button. The system then tests the query against all the images in the database and displays the 20 top-ranked targets in the small windows on the right.[2] (The highest-ranked target is displayed in the upper left, the second-highest target to its right, and so on, in row-major order.)

For convenience, the user may paint on a "canvas" of any aspect ratio. However, our application does not currently use this information in performing the match. Instead, the

[2] To avoid copyright infringements, the database for this example contains only 96 images (all created by artists who have been dead more than 75 years). Because the database is so limited, only the intended target (in the upper-left small window) appears to match the query very closely.

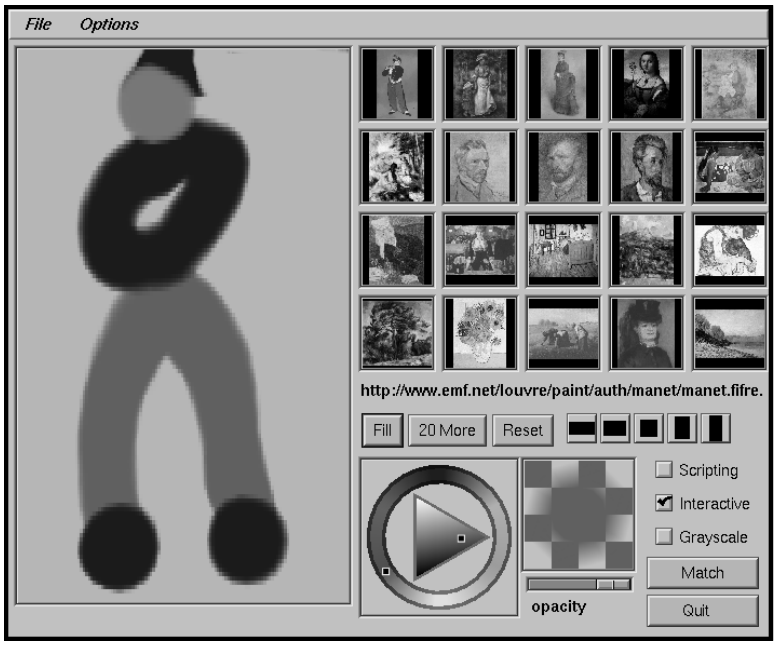


Figure 4.3: The image querying application. The user paints a query in the large rectangular window, and the 20 highest-ranked targets appear in the small windows on the right.

painted query is internally rescaled to a square aspect ratio and searched against a database in which all images have been similarly rescaled as a preprocess. We discuss how a user-specified aspect ratio might also be used to improve the match in Section 4.6.

Figure 4.4(a) shows an example of a painted query, along with the L^q rank of its intended target (c) in databases of 1093 and 20,558 images.

Rather than painting a query, the user may also click on any of the displayed target images to serve as a subsequent query, or use any stored image file as a query. Figure 4.4(b) shows an example of using a low-quality scanned image as a query, again with its L^q rank in the two databases.

Because the retrieval time is so fast (under 1/2 second in a database of 20,000 images), we have also implemented an "interactive" mode, in which the 20 top-ranked target images are updated whenever the user pauses for a half-second or more. Figure 4.8 shows the

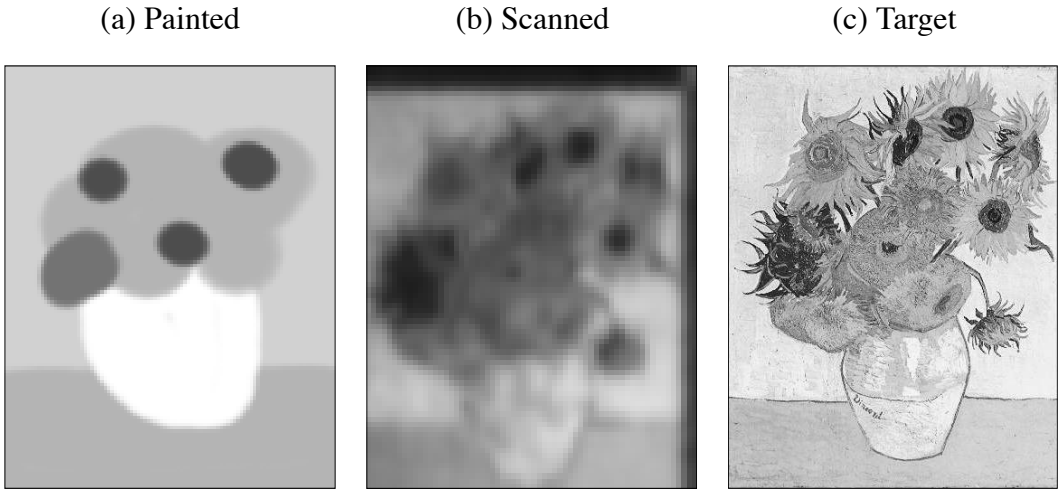


Figure 4.4: Queries and their target: (a) a query painted from memory; (b) a scanned query; and (c) their intended target. Below the queries, the L^q ranks of the intended target are shown for two databases of sizes $1093 \mid 20,558$.

progression of an interactive query, along with the actual time at which each snapshot was taken and the L^q rank of the intended target at that moment in the two different databases. A discussion of the effectiveness of this mode appears in Section 4.5.6.

4.5 Results

To evaluate our image querying algorithm, we collected three types of query data.

The first set, called "scanned queries," were obtained by printing out small $1/2" \times 1/2"$ thumbnails of our database images, using a full-color Tektronix Phaser IISDX printer at 300dpi, and then scanning them back into the system using a Hewlett-Packard ScanJet IIc scanner. As a result of these steps, the scanned images became somewhat altered; in our case, the scanned images generally appeared fuzzier, darker, and slightly misregistered from

the originals. An example of a scanned query is shown in Figure 4.4(b). We gathered 270 such queries, of which 100 were reserved for evaluating our metric, and the other 170 were used as a training set.

The second set, called "painted queries," were obtained by asking 20 subjects, most of whom were first-time users of the system, to paint complete image queries, in the non-interactive mode, while looking at thumbnail-sized versions of the images they were attempting to retrieve. We also gathered 270 of these queries and divided them into evaluation and training sets in the same fashion.

The third set, called "memory queries," were gathered in order to see how well this style of querying might work if users were not looking at small versions of the images they wanted to retrieve, but instead were attempting to retrieve images from memory. To obtain these queries, we asked each subject to initially examine two targets T_1 and T_2, and paint a query for T_1, which we threw away. The subject was then asked to iteratively examine a target T_{i+1} (starting with $i = 2$) and paint query T_i, which had not been viewed since before query T_{i-1} was painted. In this way, we hoped to get a more accurate idea of how well a user might do if attempting to retrieve a familiar image from memory, without being able to see the image directly. An example of a memory query is shown in Figure 4.4(a). We gathered 100 of these queries, which were used for evaluation only.

4.5.1 Training the metric

Each training set was subdivided into 2 equal sets. The first training set of 85 queries was used to determine the weights of the image querying metric, as described in the Section 4.3.3. The second training set of 85 queries was used to find the optimal color space, decomposition type, and number m of coefficients to store for each image. We performed an exhaustive search over all three dimensions, using color spaces RGB, HSV, and YIQ; standard and non-standard wavelet decompositions; and $m = 10, 20, 30, \ldots, 100$. For each combination, we found weights using the first set of images, and then tested these weights on

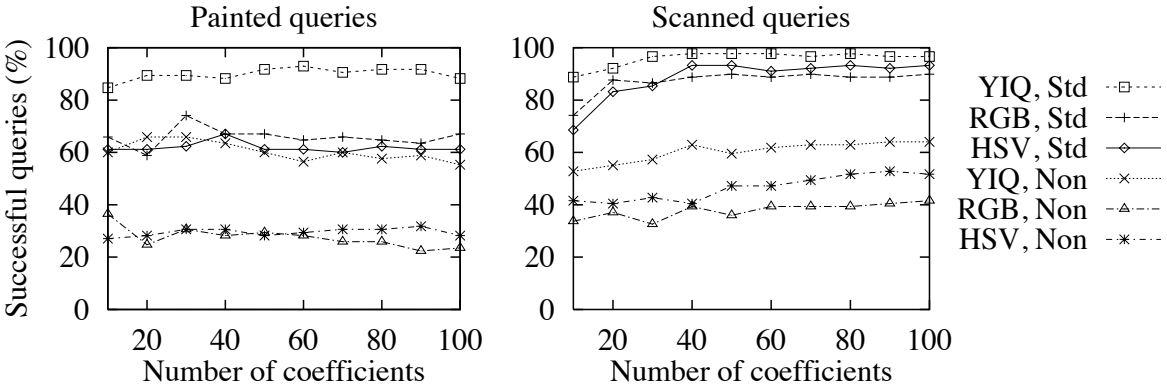


Figure 4.5: Choosing among color spaces (RGB, HSV, or YIQ), wavelet decomposition type (standard or non-standard), and number of coefficients.

the second set of images, using "the percentage of intended targets that were ranked among the top 1% in our database of 1093 images" as the evaluation function.

The results of these tests for scanned and painted queries are reported in Figure 4.5. For scanned queries, 40 coefficients with a standard decomposition and YIQ worked best. The same configuration, except with 60 coefficients, worked best for painted queries. This latter configuration was used for testing the success of memory queries as well.

4.5.2 Performance on actual queries

Using the weights obtained from the training set, we then evaluated the performance using the remaining 100 queries of each type. The graphs in Figure 4.6 compare our L^q metric to the L^1 and L^2 metrics and to a color histogram metric L^c, for a database of 1093 images. The three graphs show, from left to right: scanned queries (using 40 coefficients), painted queries (using 60 coefficients), and memory queries (using 60 coefficients).

The L^1 and L^2 metrics in these graphs were computed on both the full-resolution images (128×128 pixels) and on averaged-down versions (8×8 pixels), which have roughly the same amount of data as the 60-coefficient L^q metric. The color histogram metric L^c was

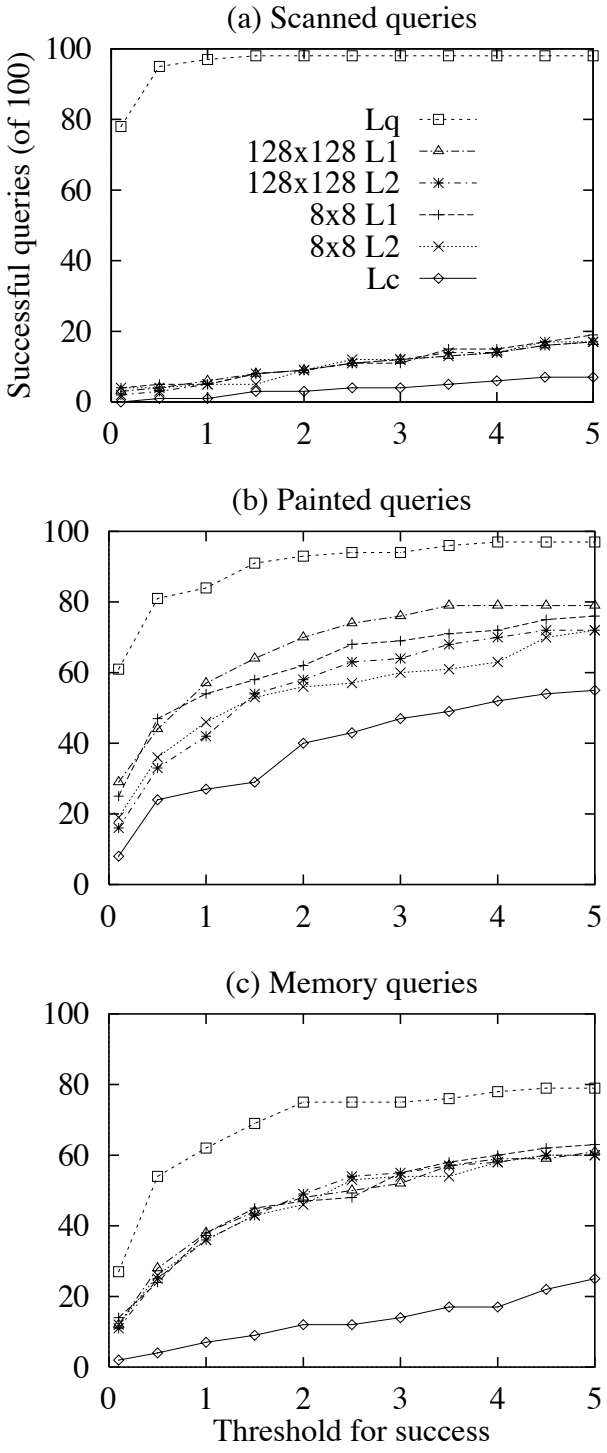


Figure 4.6: Comparison of L^q metric against L^1, L^2, and a color histogram metric L^c. The percentage of queries y ranked in the top x% of the database are plotted on the x and y axes.

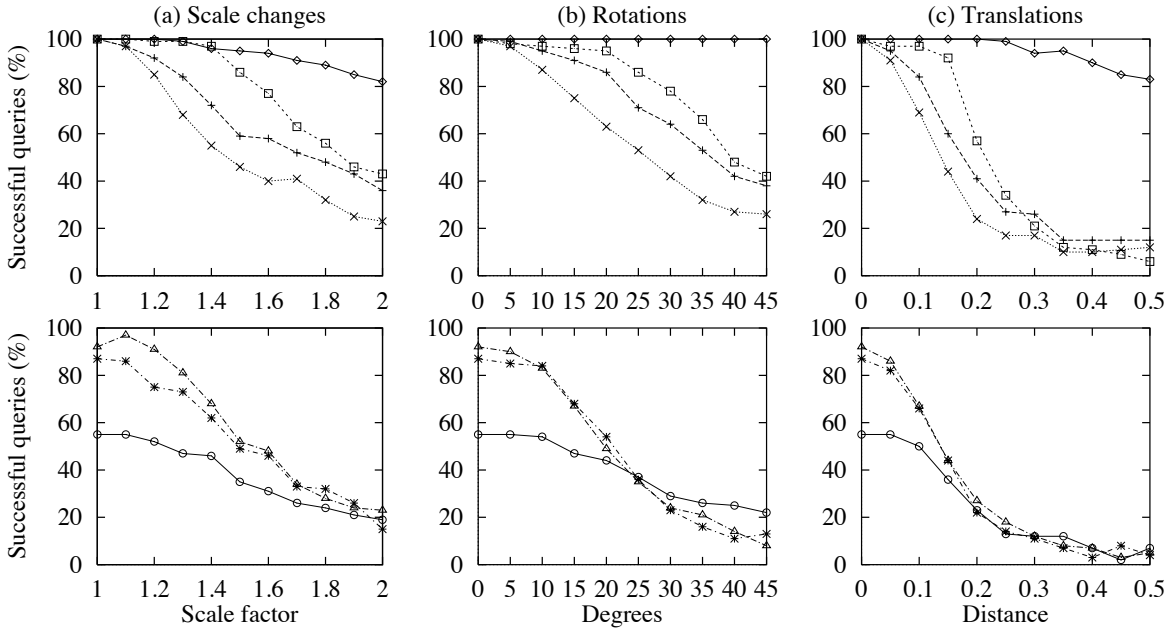


Figure 4.7: Robustness of various querying metrics with respect to different types of image distortions: (a) Scale changes; (b) Rotations; (c) Translations; (d) Color shifts; (e) All four effects combined. The top row (legend at upper right) compares the L^q metric to L^1, L^2 and L^c, using the target itself as the original undistorted query. The bottom row (legend at lower right) shows the same five tests applied to each of our 100 scanned, painted, and memory queries, using the L^q metric. (Continued on next page.)

computed by quantizing the pixel colors into a set of $6 \times 6 \times 6$ bins in RGB space, and then computing an L^1 metric over the number of pixels falling in each bin for the query versus the target.

Results for all six methods are reported by giving the percentage of queries y that were ranked among the top x% of the database images, with x and y plotted on the x- and y-axes. For example, the leftmost data point of each curve, at $x = 1/1093 \approx 0.09\%$, reports the percentage of queries whose intended targets were ranked in first place for each of the six methods; the data points at $x = 1\%$ report the percentage of queries whose intended targets were ranked among the top $\lfloor 0.01 * 1093 \rfloor = 10$ images; and so on.

Note that the scanned queries perform remarkably poorly under the L^1, L^2 and L^c

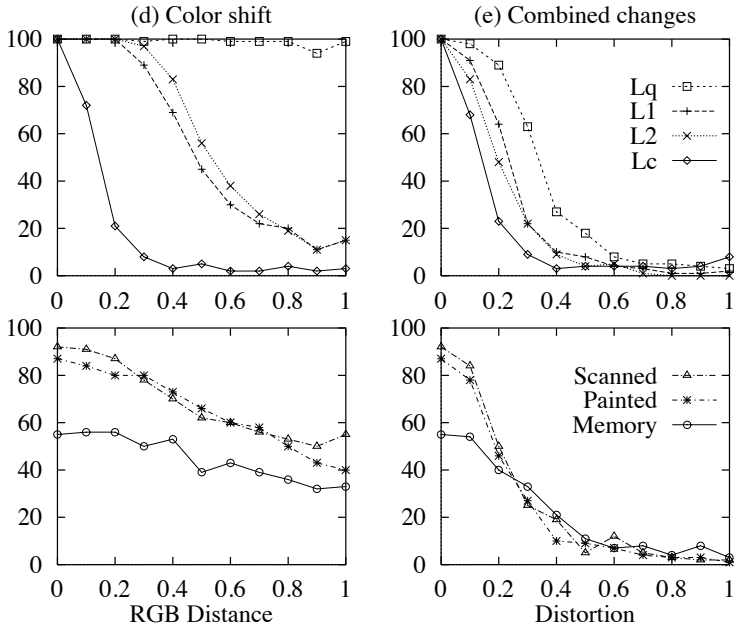


Figure 4.7 *(continued)*

metrics. These poor scores are probably due to the fact that the scanned queries were generally darker than their intended targets, and so matched many incorrect (darker) images in the database more closely.

4.5.3 Robustness with respect to distortions

In order to test more precisely how robust the different metrics are with respect to some of the distortions one might find in image querying, we devised the following suite of tests. In the first test, 100 randomly chosen color images from the database were scaled by a factor s ranging from 1 to 2 and used as a query. In the second test, the same images were rotated by a factor r between 0° and 45°. In the third test, the same images were translated in a random direction by a distance t between 0 and 0.5 times the width of the query. In the fourth test, the colors of these images were uniformly shifted in normalized RGB space in a random

direction by a distance c between 0 and 1. In the final test, all four of these transformations were applied for each test, in the order scale/rotate/translate/color-shift, with s, r, t, and c ranging as in the other tests. For all five tests, in cases where a border of the distorted image was undefined by the transformation (which occurs for rotations and translations), the image was padded with its overall average color. In cases where the color shift would lie outside the RGB cube, the color was clamped to $[0, 1]$ in each channel.

The top row of Figure 4.7 shows the results of these five tests. The curves in these graphs report the percentage of queries whose intended targets were ranked in the top 1% of the 1093-image database. Note that the L^q metric performs as well as or better than all other methods, except for L^c. However, as expected, the L^c metric does very poorly for color shifts, severely reducing this metric's utility in situations where a query's color is not always true. The bottom row shows the same five tests, but applied to each of our 100 scanned, painted, and memory queries—all with the L^q metric.

4.5.4 Effect of database size

We also wanted to test how well our method would perform as the size of the database was increased. We therefore gathered 19,465 images from the World Wide Web, using the WebCrawler [Pin94] to find files on the Web with a ".gif" extension. We computed a signature and thumbnail for each image and stored the resulting database locally, along with a "URL" for each image—a pointer back to the original Web site. The resulting application is a kind of graphical "Web browser," in which a user can paint a query and very quickly see the images on the Web that match it most closely. Clicking on one of these thumbnail images calls up the full-resolution original from the Web.

In order to check how well our metric performed, we created a set of 20 nested databases, with each database containing our original 1093 images plus increasingly large subsets of the Web database. The largest such database had 20,558 images. For each of the three sets of 100 queries, we then evaluated how many of those queries would find their intended target in

the top 1% of the different nested databases. We found that the number of queries matching their correct targets by this criterion remained almost perfectly constant in all three cases, with the number of correctly matching queries varying by at most 2% across the different database sizes.

4.5.5 Speed of evaluation

We measured the speed of our program by running 68 queries 100 times each, with databases ranging in size from $n = 1093$ to $n = 20,558$, and with the number of coefficients ranging from $m = 20$ to $m = 1000$. A regression analysis indicates that the running time is linear in both m and n, with each query requiring approximately $190 + 0.11m + 0.012n$ milliseconds to process on an SGI Indy R4400. This running time includes the time to decompose a 128×128-pixel query, score all n images in the database according to the L^q metric, and find the 20 top-ranked targets.

As two points of comparison, Table 4.2 reports the average running time of our algorithm to that of the other methods surveyed for finding a query using $m = 20$ coefficients per channel in databases of size $n = 1093$ and $n = 20,558$ images. In all cases, the times reported do not include any preprocessing that can be performed on the database images alone.

4.5.6 Interactive queries

To test the speed of interactive queries, we asked users to paint in the interactive mode, and we kept track of how long it took for the intended target to appear among the top 20 images in the database. For these tests, we used just $m = 20$ significant coefficients.

For the first such test, we had 5 users paint a total of 106 interactive queries, allowing them to look at thumbnails of the intended targets. The overall median time to retrieve the target images was 20 seconds.

Table 4.2: Average times (in seconds) to match a single query in databases of 1093 and 20,558 images under different metrics.

Metric	Time	
	$n = 1093$	$n = 20,558$
L^q	*0.19*	*0.44*
L^1 (8×8)	*0.66*	*7.46*
L^2 (8×8)	*0.68*	*6.39*
L^1 (128×128)	*47.46*	*892.60 (est.)*
L^2 (128×128)	*42.04*	*790.80 (est.)*
L^c	*0.47*	*5.03*

Next, in order to see how this median query time might vary with database size, we asked 2 users to paint a total of 21 interactive queries in our database of 20,558 images. For each query, the application kept a log of each paint stroke and the time at which it was drawn. We then used these logs to simulate how quickly the same query would bring up the intended target among the top 20 images in databases of various sizes. The results are shown in Figure 4.9.

To see if painting from memory would affect retrieval time, we selected 20 target images and, for each subject, we randomly divided these targets into two equal sets. Each subject was then asked to paint the 10 images from the first set while looking at a thumbnail of the image, and the 10 images from the second set from memory, in the style described for "memory queries" above. We used 3 subjects in this experiment. We found that the median query time increased from 18 seconds when the subjects were looking at the thumbnails, to 22 seconds when the queries were painted from memory.

In our experience with interactive querying, we have observed that users will typically be able to sketch all the information they know about an image in a minute or less, whether they are looking at a thumbnail or painting from memory. In most cases, the query succeeds within this short time. If the query fails to bring up the intended target within a minute or

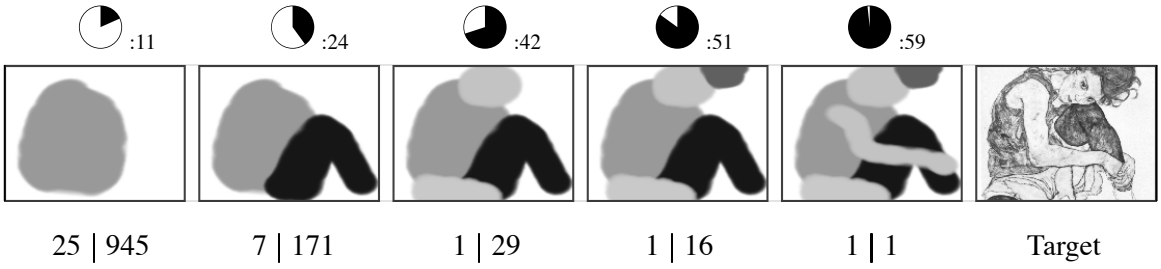


Figure 4.8: Progression of an interactive query. Above each partially-formed query is the actual time (in seconds) at which the snapshot was taken. Below each query are the L^q ranks of the intended target for databases of sizes $1093 \mid 20,558$.

so, users will typically try adding some random details, which sometimes help in bringing up the image. If this tactic fails, users will simply give up and, in a real system, would presumably fall back on some other method of searching for the image. (In this case, we report an "infinite" query time.)

We have observed two benefits of painting queries interactively. First, the time to retrieve an image is generally reduced because the user simply paints until the target image appears, rather than painting until the query image seems finished. Second, the interactive mode subtly helps "train" the user to find images more efficiently, because the application is always providing feedback about the relative effectiveness of an unfinished query while it is being painted.

4.6 Discussion and extensions

The algorithm we have described is extremely fast, requires only a small amount of data to be stored for each target image, and is remarkably effective. It is also fairly easy to understand and implement. Finally, it has parameters that can be tuned for a given database or type of query image.

Although this new image searching method has substantial advantages over previous

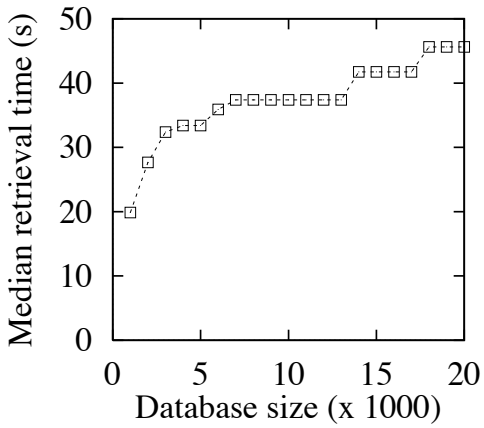


Figure 4.9: The effect of database size on median interactive query time.

approaches, its ultimate utility may depend to a large extent on the size of the image database being searched. Our tests suggest that, for the majority of non-interactive queries, our method will be able to pinpoint the correct target to within a 1%-sized subset of the overall database, regardless of the database's size. Thus, for a database of 100 images, it is easy to pull up the correct image precisely. However, for a database of 20,000 images, the user is still left with a list of 200 potential matches that must be searched visually, or by some other means. On the other hand, with interactive querying, even for a 20,000-image database it is still possible to place the target into the top 20 images the majority of the time. Nonetheless, creating a good query becomes increasingly difficult as the database grows. For a large enough database, even this interactive style of querying would begin to require more precision than most users can provide.

We have tried to perform a number of different tests to measure the success and robustness of our image querying metric. However, it is easy to envision many more tests that would be interesting to perform. One interesting test would be to try to quantify the degree to which different training sets affect our metric's sensitivity to various image distortions. For example, in querying images from memory, colors are less likely to be accurate. Presumably, a training set of "memory queries" would therefore reduce the metric's sensitiv-

ity to color accuracy. How significant is this effect? In addition, it would be interesting to examine whether providing separate training sets for individual users or for particular databases would make a significant difference in the metric's discriminatory power.

Our method also has some limitations, many of which we believe could be addressed by the following extensions:

Aspect ratio. Currently, we allow users to choose an aspect ratio for their query; however, this aspect ratio is not used in the search itself. It would be straightforward to add an extra term to our image querying metric for the similarity of aspect ratio. The weight for this term could be found experimentally at the same time as the other weights are computed.

Perceptually-based spaces. It would be interesting to try using a perceptually uniform color space, such as CIE $L^*a^*b^*$ or $L^*u^*v^*$ [Gla95], to see if it improves the effectiveness of our metric. In the same vein, it may help to compute differences on logarithmically-scaled intensities, which is closer to the way intensity is perceived [HB94].

Image clusters. Images in a large database appear to be "clustered" in terms of their proximity under our image querying metric. For example, using a portrait as a query image in our Web database selects portraits almost exclusively as targets. By contrast, using a "planet" image pulls up other planets. It would be interesting to perform some statistical clustering on the database and then show the user some representative images from the center of each cluster. These could be used either as querying keys, or merely as a way of providing an overview of the contents of the database.

Multiple metrics. In our experience with the system, we have noticed that a good query will bring up the target image, no matter which color space and decomposition method (standard or non-standard) is used; however, the false matches found in these different spaces all tend to be very different. This observation leads us to wonder whether it is possible to develop a more effective method by combining the results of searching in different color spaces and decomposition types, perhaps taking the average of the ranks in the different spaces (or, alternately, the worst of the ranks), as the rank chosen by the overall metric.

Affine transformation and partial queries. A very interesting (and more difficult) direction for future research is to begin exploring methods for handling general affine transformations of the query image or for searching on partial queries. The "shiftable transforms," described by Simoncelli *et al*. [SFAH92], which allow for multiresolution transforms with translational, rotational, and scale invariance, may be helpful in these respects. Another idea for specifying partial queries would be to make use of the alpha channel of the query for specifying the portions of the query and target images over which the L^q metric should be computed.

Chapter 5

MULTIRESOLUTION VIDEO

5.1 Introduction

In Chapters 3 and 4 we made use of multiresolution representations for curves and images—functions in one and two dimensions, respectively. In this chapter we will describe *multiresolution video*, a multiresolution representation for three-dimensional data.

Scientists often run physical simulations of time-varying data in which different parts of the simulation are performed at differing spatial and temporal resolutions. For example, in a simulation of the air flow about an airplane wing, it is useful to run the slowly-varying parts of the simulation—generally, the portion of space further from the wing—at a fairly coarse scale, both spatially and temporally, while running the more complex parts—say, the region of turbulence just aft of the wing—at a much higher resolution. The *multigrid techniques* used frequently for solving large-scale problems in physics [SP94], astronomy [SS95b], meteorology [AGG+92], and applied mathematics [MR94] are a common example of this kind of computation.

Multiresolution video provides a means of capturing time-varying image data produced at multiple scales, both spatially and temporally. Furthermore, we introduce efficient algorithms for viewing multiresolution video at arbitrary scales and speeds. For example, in a sequence depicting the flow of air about a wing, a user can interactively zoom in on an area of relative turbulence, computed at an enhanced *spatial resolution*. Analogously, fast-changing components in a scene can be represented and viewed at a higher *temporal*

resolution, allowing, for example, a propeller blade to be represented and viewed in slow motion.

Moreover, we have found that multiresolution video has applications that are useful even for conventional uniresolution video. First, the representation facilitates a variety of viewing applications, such as multiresolution playback, including motion-blurred "fast-forward" and "reverse"; constant-speed viewing of video over a network with varying throughput; and an enhanced form of video "scrubbing." The representation also provides a controlled degree of lossy compression, particularly in areas of the video that change little from frame to frame. Finally, the representation supports the assembly of complex multiresolution videos from either uniresolution or multiresolution "video clip-art" elements.

5.1.1 Related work

The multiresolution video representation described in this chapter generalizes some of the multiresolution representations that have previously been proposed for images, such as "image pyramids" [TP75] and "MIP maps" [Wil83]. It is also similar to the wavelet-based representations for images described by Berman *et al*. [BBS94] and Perlin and Velho [PV95]. In particular, like these latter works, our representation is sparse, and it supports efficient compositing operations [PD84] for assembling complex frames from simpler elements. However, unlike these works and the applications of the previous chapters, here we depart from a wavelet representation for multiresolution video.

Several commercially available video editing systems support many of the operations described in this chapter for uniresolution video. For example, Adobe After Effects allows the user to view video segments at low resolution and to construct an edit list that is later applied to the high-resolution frames off-line. Discrete Logic's Flame and Flint systems also provide digital video compositing and many other digital editing operations on videos of arbitrary resolution. Swartz and Smith [SS95c] describe a language for manipulation of video segments in a resolution-independent fashion. However, the input and output from

all of these systems is uniresolution video.

Multiresolution video also allows the user to pan and zoom to explore a flat video environment. This style of interaction is similar in spirit to two image-based environments: Apple Computer's QuickTime® VR [Che95] and the "plenoptic modeling" system of McMillan and Bishop [MB95]. These methods provide an image-based representation of an environment that surrounds the viewer. In section Section 5.4.5, we describe how such methods can be combined with multiresolution video to create a kind of "multiresolution video QuickTime VR," in which a viewer can investigate a panoramic environment by panning and zooming, with the environment changing in time and having different amounts of detail in different locations.

While not the emphasis of this work, we also describe a simple form of lossy compression suitable for multiresolution video. Video compression is a heavily-studied area, with too many papers to adequately survey here. MPEG [Gal91] and QuickTime [Rad93] are two industry standards. Other techniques based on multiscale transforms [LK90, NH88] might be adapted to work for multiresolution video.

5.1.2 Overview

The rest of this chapter is organized as follows. Section 5.2 describes our representation for multiresolution video, and Section 5.3 describes how it is created, displayed, and edited. Section 5.4 describes a variety of applications of the multiresolution video representation, and Section 5.5 provides some concrete examples. Finally, Section 5.6 summarizes the chapter and outlines some possible extensions.

5.2 Representation

Our goals in designing a multiresolution video representation were fivefold. In particular, we wanted it to:

- support varying spatial and temporal resolutions;
- require overall storage proportional only to the detail present (with a small constant of proportionality);
- efficiently support a variety of primitive operations for creating, viewing, and editing the video;
- permit lossy compression; and
- require only a small "working storage" overhead, so that video could be streamed in from disk as it is needed.

The rest of this section describes the multiresolution video format we chose and an analysis of the storage required.

5.2.1 The basic multiresolution video format

Given the results described in the previous two chapters, perhaps the most obvious choice for a multiresolution video format would be some kind of three-dimensional wavelet representation (perhaps organized in a sparse octree [Sam90]) whose three dimensions were used to encode the two spatial directions and time. Indeed, such a representation was our first choice, but we found that it did not adequately address a number of the goals enumerated above. Section 5.2.5 will outline some of our reasons for abandoning wavelets.

The structure we ultimately chose is a sparse binary tree of sparse quadtrees. The binary tree encodes the flow of time, and each quadtree encodes the spatial decomposition of a frame (Figure 5.1).

In the binary tree, called the *Time Tree*, each node corresponds to a single image, or *frame*, of the video sequence at some temporal resolution. The leaves of the Time Tree correspond to the frames at the highest temporal resolution for which information is present in the video. Internal nodes of the Time Tree correspond to box-filtered averages of their two

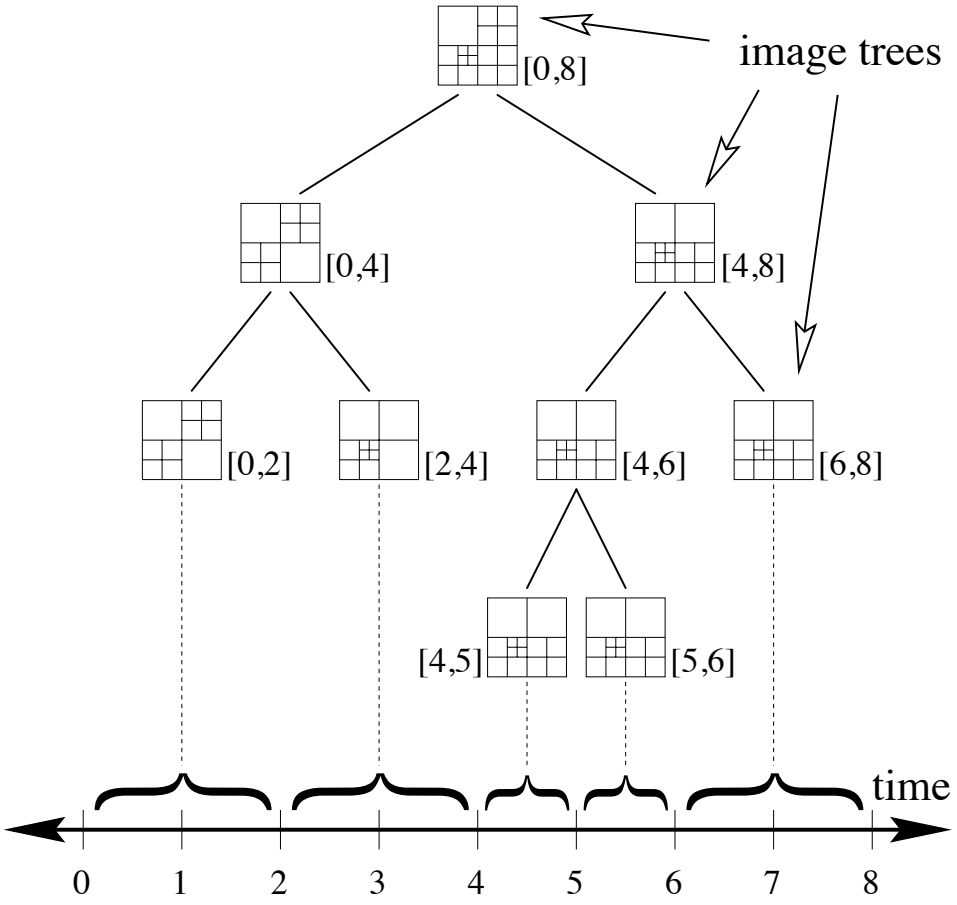


Figure 5.1: Binary tree of quadtrees.

children frames. Visually, these frames appear as motion-blurred versions of their children. Note that this representation supports video sequences with varying degrees of temporal resolution simply by allowing the Time Tree to grow to different depths in different parts of the sequence. For convenience, we will call the child nodes of the Time Tree *child time nodes* and their parents *parent time nodes*. We will use capitalized names for any time node.

Each node of the Time Tree points to a sparse quadtree, called an *image tree*, which represents the multiresolution image content of a single frame of the video sequence. In analogy to the Time Tree, leaves of an image tree correspond to pixels at the highest spatial resolution for which information is present in the particular frame being represented.

```
type TimeNode = record
    frame: pointer to ImageNode
    Half1, Half2: pointer to TimeNode
end record

type ImageNode = record
    type: TREE | COLOR
    uplink: UpLinkInfo
    union
        tree: pointer to ImageSubtree
        color: PixelRGBA
    end union
end record

type ImageSubtree = record
    avgcolor: PixelRGBA
    child[0..1,0..1]: array of ImageNode
end record

type UpLinkInfo = record
    linked: Boolean
    type: FIRST | MIDDLE | LAST
end record
```

Figure 5.2: Pseudocode for multiresolution video data structure.

Internal nodes of an image tree correspond, once again, to box-filtered averages of their children—in this case, to a 2×2 block of higher-resolution pixels. Note that the image tree supports varying spatial resolution simply by allowing the quadtree to reach different depths in different parts of the frame. We will call the child nodes of an image tree *child image nodes* and their parents *parent image nodes*. In our pseudocode we will use lowercase names for any image node. Figure 5.5(b) shows the quadtree structure for a frame of video.

Time Tree nodes are represented by the *TimeNode* record in Figure 5.2, while nodes of the image tree are represented by *ImageNode* and *ImageSubtree* records. Each subtree

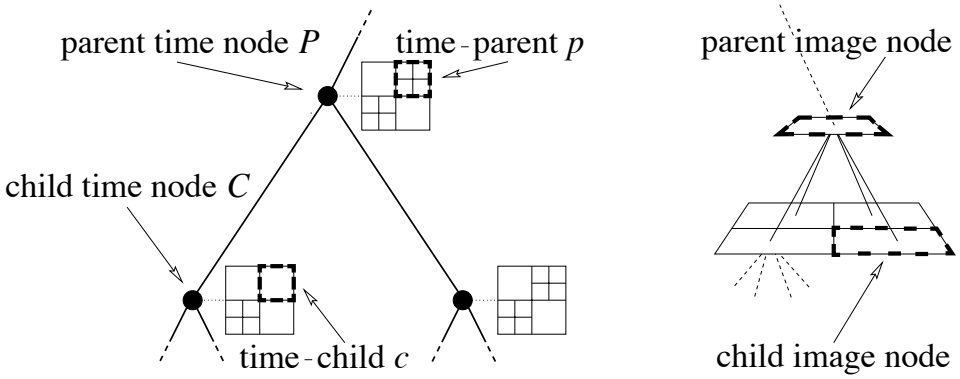


Figure 5.3: Parent-child relationships in the trees.

contains both the average color for a region of the image, stored as an RGBA pixel, and also image nodes for the four quadrants of that region. We compute the average of the pixels as if each color channel were premultiplied by alpha,[1] as prescribed by Porter and Duff [PD84]. Each image node generally contains a pointer to a subtree for each quadrant. However, if a given quadrant only has a single pixel's worth of data, then the color of the pixel is stored in the node directly, in place of the pointer. (This trick works nicely, since an RGBA pixel value is represented in our system with 4 bytes, the same amount of space as a pointer. Packing the pixel information into the pointer space allows us to save a large amount of memory that we might otherwise waste on null pointers at the leaves.) There is also an *uplink* field, whose use we will discuss in the next section.

There is an additional relationship between image nodes that is not represented explicitly in the structure, but which is nevertheless crucial to our algorithms. As described already, there are many different image nodes that correspond to the same region of space, each hanging from a different time node. We will call any two such image nodes *time-relatives*. In particular, for a given image node c hanging from a time node C, we will call the time-

[1] We do not actually premultiply with alpha in our image node representation, in order to preserve color fidelity in highly-transparent regions.

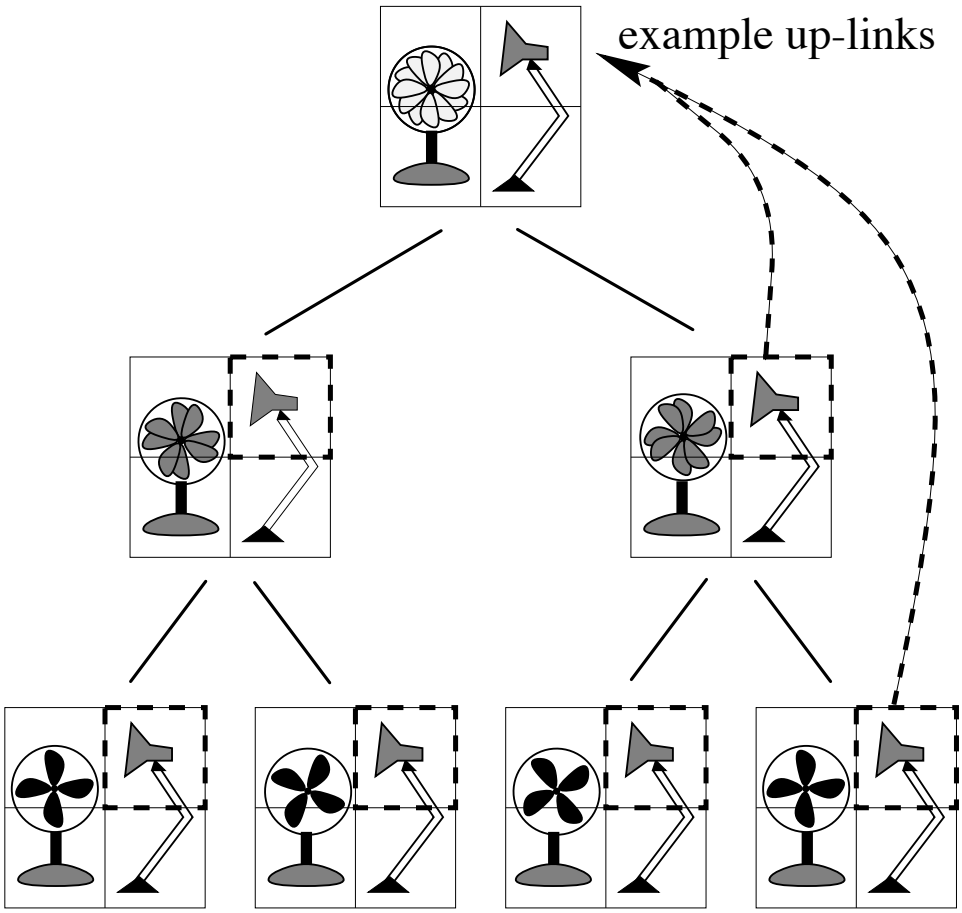


Figure 5.4: Exploiting temporal coherence. The quadrants containing the Luxo lamp are not duplicated in the lower six frames of the Time Tree. Instead, the right two quadrants in all six frames contain "up-links" to the corresponding quadrants in the Time Tree's root.

relative *p* hanging from the parent time node *P* of *C* the *time-parent* of *c*. In this case, the image node *c* is also called the *time-child* of *p*. (See Figure 5.3.) Note that a given node does not necessarily have a time-parent or a time-child, as the quadtree structures hanging from *P* and *C* may differ.

(a) Frame (b) Quadtree (c) Up-links

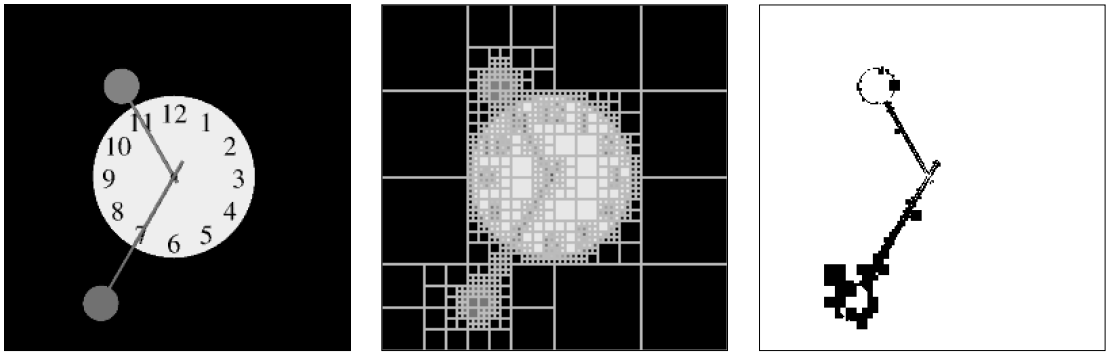


Figure 5.5: Data structures for a frame of video. (a) A frame of video containing a clock. (b) The quadtree structure for the frame. (c) The regions shown in white are covered by up-links, whereas the regions shown in black are local to this frame only.

5.2.2 Temporal coherence

Recall that the representation of each frame exploits spatial coherence by pruning the image tree at nodes for which the image content is nearly constant. We can take advantage of temporal coherence in a similar way, even in regions that are spatially complex.

Consider an image node p and its two time-children c_1 and c_2. Whenever the images in c_1 and c_2 are similar to each other, the image in p will be similar to these images as well. Rather than triplicating the pixels in all three places, we can instead just store the image in the time-parent p and allow c_1 and c_2 to point to this image directly. We call such pointers *up-links*. See Figure 5.4 for a schematic example. Figure 5.5(c) shows in white the regions of a frame contained in up-links.

Note that the pointer for an up-link uses the same field in the data structure as is used by a normal subtree, to save storage space. Up-links are maintained by the *UpLinkInfo* record in Figure 5.2. The *linked* field tells whether or not there is an up-link. There is also a *type* field, which we will describe in Section 5.3.2.

5.2.3 Storage complexity

Now that we have defined the multiresolution video data structure, we can analyze its storage cost. The *type* and *uplink* fields require very few bits, and in practice these two fields for all four children may be bundled together in a single 4-byte field in the *ImageSubtree* structure. Thus, each *ImageSubtree* contains 4 bytes (for the average color), 4×4 bytes (for the children), and 4 bytes (for the flags), yielding a total of 24 bytes. Each leaf node of an image tree comprises 4 pixels, and there are 4/3 as many total nodes in these trees as there are leaves. Assuming P pixels per time node, and using the fact that there are twice as many time nodes as there are leaves (or frames) in the Time Tree yields the overall storage complexity:

$$\frac{24 \text{ bytes}}{\text{ImageNode}} \times \frac{4 \text{ ImageNodes}}{3 \text{ ImageLeaves}} \times \frac{\text{ImageLeaf}}{4 \text{ pixels}} \times \frac{P \text{ pixels}}{\text{TimeNode}} \times \frac{2 \text{ TimeNodes}}{\text{frame}} = \frac{16P \text{ bytes}}{\text{frame}}$$

In addition, each *TimeNode* contains $3 \times 4 = 12$ bytes, and there are twice as many nodes in this tree as there are leaves. Thus, the Time Tree needs an additional 24 bytes/frame. However, since $16P$ is generally much larger than 24, we can ignore the latter term in the analysis. The overall storage is therefore 16 bytes/pixel.

In the worst case—a complete tree with no up-links—we have as many pixels in the tree as in the original image. Thus, the tree takes 4 times as much space as required by just the highest-resolution pixel information alone. It is worthwhile to compare this overhead with the cost of directly storing the same set of time- and space-averaged frames, without allowing any space for pointers or flags. Such a structure would essentially involve storing all powers-of-two time and spatial scales of each image, requiring a storage overhead of $8/3$. Thus, our storage overhead of 4 is only slightly larger than the minimum overhead required. However, as will be described in Section 5.3.1, the very set of pointers that makes our worst-case overhead larger also permits both lossless and lossy compression by taking advantage of coherence in space and time.

5.2.4 Working storage

One of the goals of our representation was to require a small "working storage" overhead, so that video could be streamed in from disk only as it is needed. This feature is crucial for viewing very large sequences, as well as for the editing operations we describe in Section 5.3. As we will see when we discuss these operations in detail, this goal is easily addressed by keeping resident in memory just the image trees that are currently being displayed or edited, along with all of their time-ancestors. For a video clip with 2^k frames, the number of time-ancestors required is at most k.

5.2.5 Comparison with wavelets

In Chapters 3 and 4 we used wavelets to represent multiresolution curves and images. Our first impulse—and, in fact, our first implementation—was to use a 3D wavelet representation for multiresolution video. However, we eventually moved to the data structure described in this section for several reasons. First, the box basis functions we use now are simpler, making it faster to extract and render a frame of video; with wavelets some form of synthesis was necessary. Second, wavelet coefficients require increasing numbers of bits at finer levels of detail, so we had to use floating-point numbers (rather than bytes) to store the color channels—a factor of four expansion in storage. Third, the wavelets we used (non-standard tensor-product Haar wavelets) made it difficult to separate the spatial and temporal dimensions; thus, it was expensive to extract a frame in which the time and space dimensions were scaled differently. While it is possible to construct a wavelet basis that avoids this problem, we were not able to find an efficient compositing algorithm for it. Finally, the current representation takes advantage of areas of a video sequence that have temporal coherence but no spatial coherence; the wavelets we used were unable to compress such sequences.

5.3 Basic algorithms

In this section we describe efficient algorithms for creating, displaying, and editing multiresolution video.

5.3.1 Creating multiresolution video

We begin with the problem of creating multiresolution video from conventional uniresolution video. We break this process into two parts: creating the individual frames, and linking them together into a multiresolution video sequence.

Creating the individual frames

Given a $2^\ell \times 2^\ell$ source frame S we construct an image tree by calling the function *CreateFrame*() in Figure 5.6, passing arguments $(S, 0, 0, \ell)$. Image trees built from images that are not of dimension $2^\ell \times 2^\ell$ are implicitly padded with transparent, black pixels.

The quadtree constructed by *CreateFrame*() is complete. The next step is to take advantage of spatial coherence by culling redundant information from the tree. The function call *PruneTree*(p, a, δ) recursively traverses the image tree p and prunes any subtree whose colors differ from its average color a by less than a threshold δ. Choosing $\delta = 0$ yields lossless compression, whereas using $\delta > 0$ permits an arbitrary degree of lossy compression at the expense of image degradation. The function *ColorDiff*() measures the distance between two colors (r_1, g_1, b_1, a_1) and (r_2, g_2, b_2, a_2). We chose to measure the distance as the sum of the distances between color components, weighted by their luminance values:

$$0.299|r_1a_1 - r_2a_2| + 0.587|g_1a_1 - g_2a_2| + 0.114|b_1a_1 - b_2a_2|$$

In practice, the source material may be multiresolution in nature. For example, the results of some of the scientific simulations described in Section 5.5 were produced via adaptive refinement. It is easy to modify the function *CreateFrame*() to sample source

```
function CreateFrame(S, x, y, ℓ): returns ImageNode
    if ℓ = 0 then return ImageNode(COLOR, S[x, y])
    for each i,j ∈ {0,1} do
        x' ← 2x + i
        y' ← 2y + j
        subtree.child[i,j] ← CreateFrame(S, x', y', ℓ − 1)
    end for
    subtree.avgcolor ← AverageChildren(subtree.child[0..1, 0..1])
    return ImageNode(TREE, subtree)
end function

function PruneTree(p, a, δ): returns Boolean
    if p.type = COLOR then return (ColorDiff(p.color, a) ≤ δ)
    prune ← TRUE
    for each i,j ∈ {0,1} do
        prune ← prune and PruneTree(p.child[i,j], p.avgcolor, δ)
    end for
    if prune = FALSE then return FALSE
    free(p.child[0..1, 0..1])
    p ← ImageNode(COLOR, p.avgcolor)
    return TRUE
end function
```

Figure 5.6: Pseudocode to create and prune a frame of multiresolution video.

material at different levels of detail in different parts of a frame. In this case, the recursive function descends to varying depths, depending on the amount of detail present in the source material.

Linking the frames together

The next step is to link all the frames together into the Time Tree. We first insert all the image trees at the leaves of the Time Tree, and then compute all of the internal nodes by averaging pairs of frames in a depth-first recursion. Now that the complete Time Tree is built, the two procedures in Figure 5.7 discover and create all the up-links.

```
procedure MakeMRVideo(Timetree, δ):
    for each Half ∈ {Half1, Half2} of Timetree do
        if Half ≠ NULL then
            MakeUpLinks(Half.frame, Timetree.frame, δ)
            MakeMRVideo(Half, δ)
        end if
    end for
end procedure

function MakeUpLinks(p, c, δ): returns Boolean
    c.uplink.linked ← FALSE
    if p = NULL or p.type ≠ c.type then
        return FALSE
    else if c.type = COLOR then
        c.uplink.linked ← (ColorDiff(p.color, c.color) ≤ δ)
        return c.uplink.linked
    end if
    link ← TRUE
    for each i, j ∈ {0,1} do
        link ← (link and MakeUpLinks(p.child[i, j], c.child[i, j]), δ)
    end for
    if link = FALSE then return FALSE
    free(c.tree)
    c.tree ← p.tree
    c.uplink.linked ← TRUE
    return TRUE
end function
```

Figure 5.7: Pseudocode to link together frames of multiresolution video.

The *MakeMRVideo*() routine works by finding all of the up-links between the root of the Time Tree and its two child time nodes. The routine then calls itself recursively to find up-links between these children and their descendents in time. Because of the preorder recursion, up-links may actually point to any time-ancestor, not just a time-parent. (See Figure 5.4.)

The *MakeUpLinks*() function attempts to create an up-link from a time-child c to its

```
procedure DrawImage(c, x, y, ℓ):
    if c.uplink.linked and c.uplink.type ≠ FIRST then return
    if c.type = COLOR then
        DrawSquare(x, y, 2^ℓ, c.color)
    else if ℓ = 0 then
        DrawPixel(x, y, c.avgcolor)
    else
        for each i,j ∈ {0,1} do
            DrawImage(c.child[i,j], 2x + i, 2y + j, ℓ − 1)
        end for
    end if
end procedure
```

Figure 5.8: Pseudocode to draw a frame of multiresolution video.

time-parent p. An up-link is created if the two nodes are both subtrees with identical structure, and all of their descendent nodes are sufficiently close in color. The function returns TRUE if such an up-link is created. It also returns TRUE if the two nodes are colors and the two colors are sufficiently close; it furthermore sets the child node's *uplink* flag, which is used to optimize the display operation in the following section.

After executing *MakeMRVideo*(), we traverse the entire Time Tree in a separate pass that sets the *type* field of the *uplink* structure, whose use is explained in the following section.

5.3.2 Display

Drawing a frame at an arbitrary power-of-two spatial or temporal resolution is simple. Displaying at a particular temporal resolution involves drawing frames at the corresponding level in the Time Tree. Displaying at a particular spatial resolution involves drawing the pixels situated at the corresponding level in the image trees.

The up-links that were used in the previous section to optimize storage can also play a

role in optimizing the performance of the display routine when playing successive frames. We would like to avoid refreshing any portion of a frame that is not changing in time; the up-links provide exactly the information we need. In particular, if we have just displayed frame t, then we do not need to render portions of frame $t+1$ (at the same time level) that share the same up-links. We will use the *type* field in the *UpLinkInfo* structure to specify the first and last up-links of a sequence of frames that all share the same parent data. When playing video forward, we do not need to render any region that is pointed to by an up-link, unless it is a FIRST up-link. Conversely, if we are playing backward, we only need to render LAST up-links.

To render the image content c of a single multiresolution video frame at a spatial resolution $2^\ell \times 2^\ell$, we can call the recursive routine *DrawImage*() in Figure 5.8, passing it the root c of an image tree and other parameters $(0,0,\ell)$.

The routine *DrawSquare*() renders a square at the given location and size in our application window, while *DrawPixel*() renders a single pixel. Note that *DrawImage*() assumes that the video is being played in the forward direction from beginning to end. A routine to play the video in reverse would have to use LAST in place of FIRST in the first line. A routine to display a single frame that does not immediately follow the previously displayed frame (for example, the first frame to be played) would have to omit the first line of code entirely.

One further optimization is that we actually keep track of the bounding box of non-transparent pixels in each frame. We intersect this bounding box with the rectangle containing the visible portion of the frame on the screen, and only draw this intersection. Thus, if only a small portion of the frame is visible, we only draw that portion.

The *DrawImage*() routine takes time proportional to the number of squares that are drawn, assuming that the time to draw a square is constant.

Fractional-level zoom

The *DrawImage*() routine as described displays multiresolution video at any power-of-two spatial resolution. Berman *et al*. [BBS94] describe a simple method to allow users to view multiresolution images at any arbitrary scale. We have adapted their method to work for multiresolution video. The basic idea is that if we want to display a frame of video at a fractional level between integer levels $\ell - 1$ and ℓ, we select pixels from the image tree as though we were drawing a $2^\ell \times 2^\ell$ image, and then display those pixels at locations appropriate to the fractional level. So if a pixel would be drawn at location (x, y) in a $2^\ell \times 2^\ell$ image, then it would be drawn at location (x', y') in an $M \times M$ image, where

$$x' = \lfloor xM/2^\ell \rfloor \qquad y' = \lfloor yM/2^\ell \rfloor$$

Furthermore, as with MIP maps [Wil83], we interpolate between the colors appearing at levels ℓ and $\ell - 1$ in the image tree in order to reduce point-sampling artifacts. Drawing at this fractional level is only slightly more expensive than drawing pixels at level ℓ.

Similarly, even though we are selecting frames from the Time Tree corresponding to power-of-two intervals of time, we can achieve "fractional rates" through the video, as will be described in Section 5.4.2.

5.3.3 Combining video clips

This section describes a set of linear-time algorithms for translating, scaling, and compositing multiresolution video sequences. Such operations are useful, for example, in the video clip-art application described in Section 5.4.4. The algorithms are fairly straightforward to implement, given the relatively small amount of pseudocode in this section. However, due to their highly recursive nature, they can be somewhat subtle to understand.

Translating a video clip

When combining video sequences, the various elements may need to be registered with respect to one another, requiring that they be translated and scaled within their own coordinate frames.

The basic operations of translation and scaling are well-understood for quadtrees [Sam90]. However, as with drawing frames, we want these operations to take advantage of the temporal coherence encoded in the up-links of our data structure. For example, suppose we wanted to translate the fan and lamp video of Figure 5.4 a bit to the left. The regions of the video that contain the lamp should only be translated in the root node of the Time Tree, and all the time-children must somehow inherit that translation.

The routine *TranslateTimeTree*() in Figure 5.9 translates a multiresolution video clip, rooted at time node C, by an amount (dx, dy) at level ℓ_{tran} to produce a resulting Time Tree C'. In order to handle up-links, the routine is also passed the parent time node P of C, as well as the result P' of (previously) translating P by the given amount. In the top-level call to the procedure, the parameters P and P' are passed as NULL, and the Time Tree C' initially points to an image node containing just a single clear pixel. As the procedure writes its result into C', the translated image tree is developed (and padded with clear pixels as it is extended).

The pseudocode makes a call to *ComputeSpatialAverages*(), which calculates average colors in the internal nodes of the image tree, using code similar to the *CreateFrame*() routine from Section 5.3.

The *TranslateFrame*() routine translates a single image tree c by an amount (dx, dy) at level ℓ_{tran}. In general the translation can cause large regions of constant color (leaves high in c) to be broken up across many nodes in the resulting tree c'. To handle the up-links, we must pass into the procedure the time-parent p of c, as well as the result p' of (previously) translating p. We also pass into the procedure arguments x, y and ℓ (initially all 0), which keep track of the location and level of node c.

The procedure recursively descends image tree c, pausing to copy any “terminal”

```
procedure TranslateTimeTree(C, C′, P, P′, dx, dy, ℓ_tran):
    TranslateFrame(C.frame, C′.frame, P.frame, P′.frame, dx, dy, ℓ_tran, 0, 0, 0)
    ComputeSpatialAverages(C′.frame)
    for each Half ∈ {Half1, Half2} of Timetree do
        if C.Half ≠ NULL then
            TranslateTimeTree(C.Half, C′.Half, C, C′, dx, dy, ℓ_tran)
        end if
    end for
end procedure

procedure TranslateFrame(c, c′, p, p′, dx, dy, ℓ_tran, x, y, ℓ):
    if c.Type = COLOR or c.uplink.linked or ℓ_tran = ℓ then
        w ← 2^(ℓ_tran − ℓ)
        r ← Rectangle(w · x + dx, w · y + dy, w, w, ℓ_tran)
        PutRectInTree(c, c′, p′, r, 0, 0, 0)
    else
        for each i, j ∈ {0, 1} do
            TranslateFrame(c.child[i, j], c′, p.child[i, j], p′, dx, dy, ℓ_tran,
                                                        2x + i, 2y + j, ℓ + 1)
        end for
    end if
end procedure

procedure PutRectInTree(c, c′, p′, r, x, y, ℓ):
    coverage ← CoverageType(r, x, y, ℓ)
    if coverage = COMPLETE then
        if c.type = COLOR or not c.uplink.linked then
            c′ ← c
        else
            c′ ← p′
            c′.uplink.linked ← TRUE
        end if
    else if coverage = PARTIAL then
        for each i, j ∈ {0, 1} do
            PutRectInTree(c, c′.child[i, j], p′.child[i, j], r, 2x + i, 2y + j, ℓ + 1)
        end for
    end if
end procedure
```

Figure 5.9: Pseudocode to translate a multiresolution video clip.

squares that it encounters as it goes. There are three kinds of terminal squares: large regions of constant color, subtrees that hang from level ℓ_{tran}, and up-links. In the first two cases, we copy the source from the original tree, whereas in the latter case we copy from the time-parent. A square's size and position are combined in a single structure *Rectangle*$(x, y, width, height, \ell_r)$, the coordinates of which are relative to the level ℓ_r. When the procedure finds one of these squares, it copies it into the resulting tree using the procedure *PutRectInTree*(), also in Figure 5.9.

The procedure *PutRectInTree*() recursively descends the result tree c' to find those nodes that are completely covered by the given rectangle, an approach reminiscent of Warnock's algorithm [FvDFH90]. The function *CoverageType*(r, x, y, ℓ) returns a code indicating whether rectangle r completely covers, partially covers, or does not cover pixel (x, y) at level ℓ. For those nodes that are completely covered, *PutRectInTree*() copies either a color or a pointer, depending on the type of node being copied. If the node is a color, then the color is simply copied to its new position. If the node is a pointer but not an up-link, the routine copies the pointer, which essentially moves an entire subtree from the original tree. Finally, if the node is an up-link, the routine copies the corresponding pointer from the (already translated) time-parent p'. Thus, we have to descend the result tree c' and its time-parent p' in lock-step in the recursive call.

As with *DrawImage*(), the complexity of *TranslateFrame*() is related to the number of nodes it copies using *PutRectInTree*(). The latter procedure is dependent on the number of nodes it encounters when copying a rectangle. Since the former call makes a single pass over the source quadtree c, and the collective calls to the latter procedure make a single pass over the resulting image tree c', the overall complexity is proportional to the sum of the complexities of c and c'.

```
procedure ScaleNode(c, c', p, p', s_x, s_y, x, y, ℓ):
    if c.type = COLOR or c.uplink.linked then
        r ← Rectangle(s_x · x, s_y · y, s_x, s_y, ℓ)
        PutRectInTree(c, c', p', r, 0, 0, 0)
    else
        for each i, j ∈ {0, 1} do
            ScaleNode(c.child[i, j], c', p.child[i, j], p', s_x, s_y, 2x + i, 2y + j, ℓ + 1)
        end for
    end if
end procedure
```

Figure 5.10: Pseudocode to scale a frame of multiresolution video.

Scaling a video clip

Here, we consider scaling a Time Tree by some integer factors s_x in the x direction and s_y in y. Note that to shrink a video frame by any power of two in *both* x and y we simply insert more image parent nodes above the existing image root, filling in any new siblings with "clear." Conversely, to magnify a video frame by any power of two, we simply scale all other videos down by that factor, since we are only interested in their *relative* scales. Thus, scaling both x and y by any power of two is essentially free, and we can really think of the scales as being $s_x/2^\ell$ and $s_y/2^\ell$ for any (positive or negative) ℓ. For efficiency, it is best to divide both s_x and s_y by their greatest common power-of-two divisor.

There is a procedure called *ScaleTimeTree*(C, C', P, P', s_x, s_y) whose structure is identical to *TranslateTimeTree*() above and is not presented here. This procedure makes a call to *ScaleNode*() in Figure 5.10, whose arguments and behavior are analagous to those of *TranslateNode*() above. This algorithm for scaling uses the same method as the one for translation—it essentially draws the scaled tree into the result tree, employing the procedure *PutRectInTree*() to do most of the work. Since some trees may need to be both scaled and translated, and since the algorithms use the same approach, in practice we use a third function that performs both operations at once when both a scale and translation are neces-

sary.

The time complexity of scaling is the same as translation: linear in the size of the input and output.

Compositing two video clips

The final operation addressed in this section is compositing two Time Trees A and B using any one of the 12 compositing operations of Porter and Duff [PD84]. The function *CompositeTimeTrees*(A,B,*op*) whose psuedocode appears in Figure 5.11 recursively traverses A and B in a bottom-up fashion, compositing child time nodes first, then their parents. If one of A or B has more temporal resolution than the other, then a temporary node is created by the *NewUplinkNode*() function. Invoking this function with the argument A creates a new *TimeNode* containing a single *ImageNode*, each of whose four children is an up-link pointing to its "time-parent" in A.

The function *CompositeFrames*() works on two image trees a and b, taking a pair of arguments *aUp* and *bUp* that are set to FALSE in the top-level call; these flags are used to keep track of whether a and b are really parts of a time-parent. The function also takes a pair of arguments c_1 and c_2 that are the time-children of the resulting tree. In order to pass c_1 and c_2, the *CompositeTimeTrees*() function must have already computed these time-children, which is why it makes a bottom-up traversal of the Time Tree.

We composite two image trees by traversing them recursively, in lock-step, compositing any leaf nodes. Child colors are propagated up to parents to construct internal averages. The helper function *GC*() (for "*GetChild*" or "*GetColor*") simply returns its argument node if it is a color, or the requested child if it is a subtree.

There are two subtleties to this algorithm. The first is that when the routine finds some region of the result for which both a and b are up-links (or subtrees of up-links), then it can assume that the result will be an up-link as well; in this case it simply returns NULL. Later, after all of the frames in the Time Tree have been composited, we invoke a simple function

```
function CompositeTimeTrees(A, B, op): returns TimeTree
    for each Half ∈ {Half1, Half2} do
        if A.Half = NULL and B.Half = NULL then
            Result.Half ← NULL
        else
            Ahalf ← A.Half
            Bhalf ← B.Half
            if Ahalf = NULL then Ahalf ← NewUplinkNode(A) end if
            if Bhalf = NULL then Bhalf ← NewUplinkNode(B) end if
            Result.Half ← CompositeTimeTrees(Ahalf, Bhalf, op)
        end if
    end for
    Result.frame ← CompositeFrames(A.frame, B.frame, FALSE, FALSE,
                                   Result.Half1.frame, Result.Half2.frame, op)
    return Result
end function

function CompositeFrames(a, b, aUp, bUp, c1, c2, op): returns ImageNode
    if a.uplink.linked then aUp ← TRUE end if
    if b.uplink.linked then bUp ← TRUE end if
    if aUp and bUp then return NULL end if
    if a.Type = COLOR and b.Type = COLOR then
        if c1 = NULL or c2 = NULL then
            return ImageNode(COLOR, CompositePixels(a, b, op))
        else
            return ImageNode(COLOR, Average(c1.avgcolor, c2.avgcolor))
        end if
    end if
    for each i, j ∈ {0, 1} do
        result.child[i, j] ← CompositeFrames(GC(a, i, j), GC(b, i, j),
                                             aUp, bUp, GC(c1, i, j), GC(c2, i, j), op)
    end for
    result.avgcolor ← AverageChildColors(result)
    return result
end function
```

Figure 5.11: Pseudocode to composite two multiresolution video clips.

that traverses the Time Tree once, replacing all NULL pointers with the appropriate up-link. (This assignment cannot occur in *CompositeFrames*() because the nodes to which the up-links will point have not been computed yet.) The second subtlety is that if time-child c_1 or c_2 is NULL it means that the resulting image node has no time-children: either the node is part of an image tree that hangs from a leaf of the Time Tree, or its children are up-links. In either case we perform the compositing operation. If, on the other hand, c_1 and c_2 exist, then we are working on an internal node in the Time Tree and we can simply average c_1 and c_2.

The compositing operation described in this section creates a new Time Tree that uses up-links to take advantage of any temporal coherence in the resulting video. Since this resulting Time Tree is built using two bottom-up traversals, the complexity of creating it is linear in the size of the input trees.

5.4 Applications

We now describe several applications of the primitive operations presented in the last section. These applications include multiresolution playback, with motion-blurred "fast-forward" and "reverse"; constant perceived-speed playback; enhanced video scrubbing; "video clip-art" editing and compositing; and "multiresolution video QuickTime VR."

These applications of multiresolution video serve as "tools" that can be assembled in various combinations into higher-level applications. We describe our prototype multiresolution video editing and viewing application in Section 5.5.

5.4.1 Multiresolution playback

The primary application of multiresolution video is to support playback at different temporal and spatial resolutions. To play a video clip at any temporal resolution 2^k and spatial resolution $2^\ell \times 2^\ell$ we simply make successive calls to the procedure *DrawImage*(), passing it a series of nodes from level k of the Time Tree, as well as the spatial level ℓ. We can zoom

in or out of the video by changing the level ℓ.

Similarly, for "motion-blurred" fast-forward and reverse, we use a smaller time level k. In our implementation the motion-blur effect comes from simple box filtering of adjacent frames. Though box-filtering may not be ideal for creating high-quality animations, it does appear to be adequate for searching through video.

Sometimes it may be desirable to have a limited degree of motion blur, which might, for example, blur the action in just the first half of the frame's time interval. This kind of limited motion blur can be implemented by descending one level deeper in the Time Tree, displaying the first child time node rather than the fully motion-blurred frame.

5.4.2 Constant perceived-speed playback

During video playback, it is useful to be able to maintain a constant perceived speed, despite variations in the network throughput or CPU availability. Multiresolution video provides two ways of adjusting the speed of play, which can be used to compensate for any such variations in load. First, by rendering individual frames at a finer or coarser spatial resolution, the application can adjust the rendering time up or down. Second, by moving to higher or lower levels in the Time Tree, the application can also adjust the perceived rate at which each rendered frame advances through the video.

These two mechanisms can be traded off in order to achieve a constant perceived speed. One possibility is to simply adjust the spatial resolution to maintain a sufficiently high frame rate, say 30 frames/second. If, however, at some point the degradation in spatial resolution becomes too objectionable (for instance, on account of a large reduction in network bandwidth), then the application can drop to a lower frame rate, say, 15 frames/second, and at the same time move to the next higher level of motion-blurred frames in the Time Tree. At this lower frame rate, the application has the liberty to render more spatial detail, albeit at the cost of more blurred temporal detail.

Note that by alternating between the display of frames at two adjacent levels in the Time

Tree, we can play at arbitrary speeds, not just those related by powers of two. This behavior is analogous to "3:2 pulldown," a technique for transferring 24 frame-per-second film to 30 frame-per-second video.

5.4.3 Scrubbing

Conventional broadcast-quality video editing systems allow a user to search through a video interactively by using a slider or a knob, in a process known as "scrubbing." In such systems, frames are simply dropped to achieve faster speeds through the video.

Multiresolution video supports a new kind of scrubbing that shows all of the motion-blurred video as the user searches through it, rather than dropping frames. In our implementation, the user interface provides a slider whose position corresponds to a position in the video sequence. As the user moves the slider, frames from the video are displayed. The temporal resolution of these frames is related to the speed at which the slider is pulled: if the slider moves slowly, frames of high temporal detail are displayed; if the slider moves quickly, blurred frames are displayed.

The benefits of this approach are similar to those of the constant perceived-speed playback mechanism described above. If the slider is pulled quickly, then the application does not have an opportunity to display many frames; instead, it can use the motion-blurred frames, which move faster through the video sequence. In addition, the motion blur may provide a useful visual cue to the speed at which the video is being viewed.

5.4.4 Clip-art

In our multiresolution video editor (Figure 5.12), the user may load video fragments, scale them, arrange them spatially with respect to each other, and preview how they will look together. These input fragments may be thought of as "video clip-art" in the sense that the user constructs the final product as a composite of these elements.

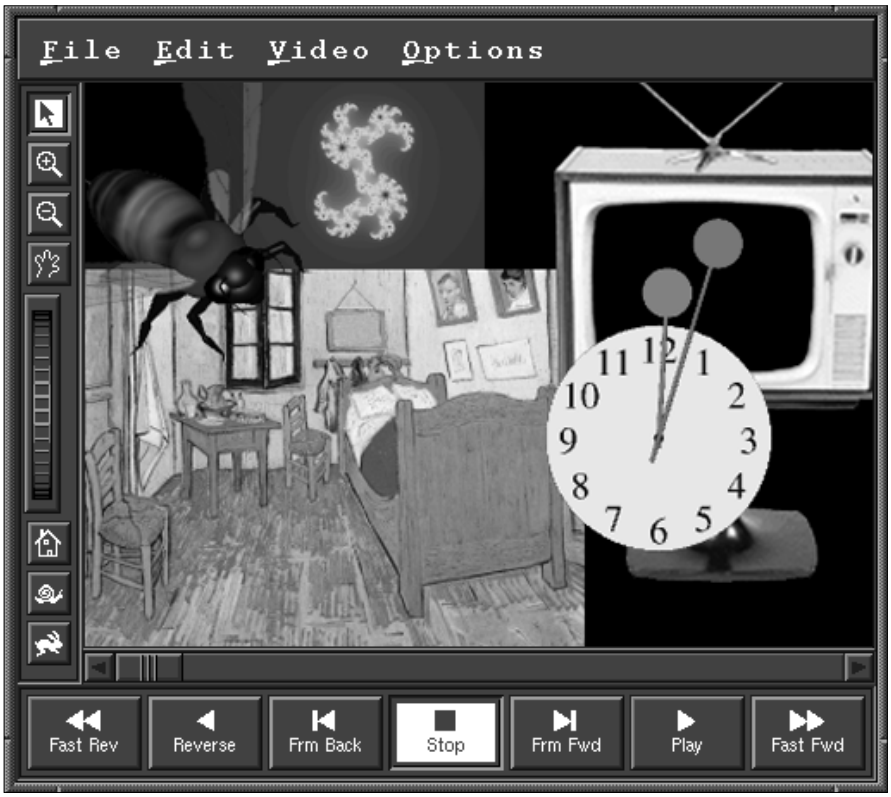


Figure 5.12: The multiresolution video application.

Since the final composition can take a long time to construct, our application provides a preview mode, which shows roughly how the final product will appear. The preview may differ from the final composite in that it performs compositing on the images currently being displayed rather than on the underlying video, which is potentially represented at a much higher resolution. (The degree to which the preview differs from the final composite is exactly the the amount by which the "compositing assumption" [PD84] is violated.) When viewing the motion-blurred result of compositing two video sequences, there is a similar difference between the preview provided in our editor and the actual result of the compositing operation.

Once the desired effect is achieved, the various clip-art elements can be translated, scaled, and composited (typically while the user goes off to get a cup of coffee) into a single

multiresolution video, using the operations described in Section 5.3.3. This video may be saved for subsequent viewing, or it may be combined with other elements as clip-art to form an even more elaborate multiresolution video.

5.4.5 Multiresolution video QuickTime VR

Apple Computer's QuickTime VR (QTVR) allows a user to explore an environment by looking from a fixed camera position out into a virtual world in any direction. Chen [Che95] proposes a potential augmentation of QTVR based on quadtrees that would provide two benefits. First, it would allow users to zoom into areas where there is more detail than in other areas. Second, it would reduce aliasing when the user zooms out. We implemented this idea, and extended it in the time dimension as well. Two simple modifications to multiresolution video were all that were required to achieve this "multiresolution video QuickTime VR" (MRVQTVR?!). First, we treat the video frames as panoramic images, periodic in the x direction. Second, we warp the displayed frames into cylindrical projections based on the view direction.

5.5 Results

We have implemented all of the operations of the previous section as part of a single prototype multiresolution video editing and viewing application, shown in Figure 5.12. Using the application, a user can zoom in and out of a video either spatially or temporally, pan across a scene, grab different video clips and move them around with respect to each other, play forward or backward, and use several sliders and dials to adjust the zoom factor, the speed of play through the video, the desired frame rate, and the current position in time.

Figure 5.13 illustrates how multiresolution video can be used for visualization of multiresolution data, in this case, an animation of the Julia set [FvDFH90]. The data were generated procedurally, with higher spatial resolution in places of higher detail, as described

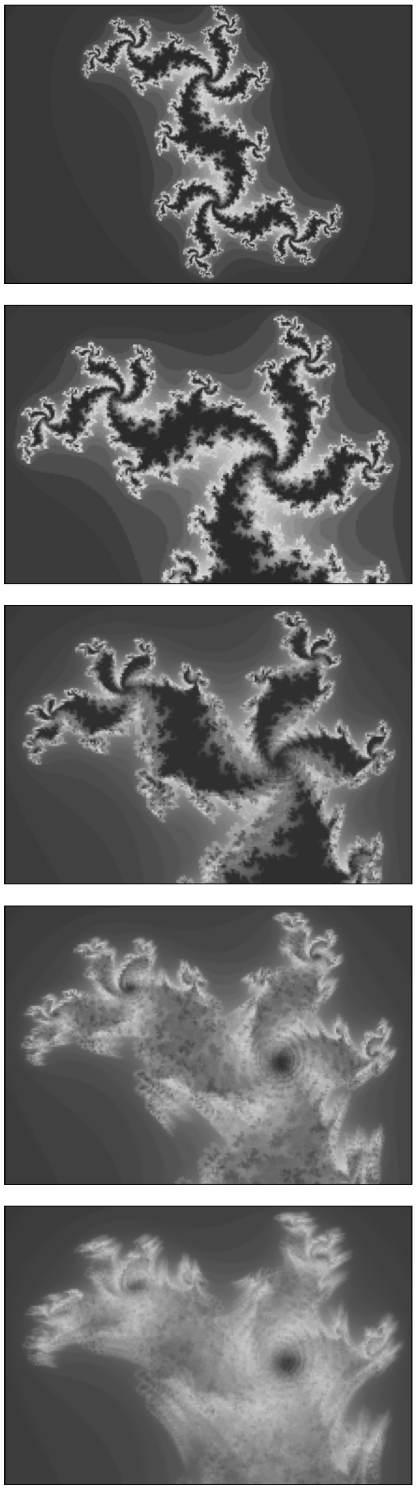


Figure 5.13: Julia set.

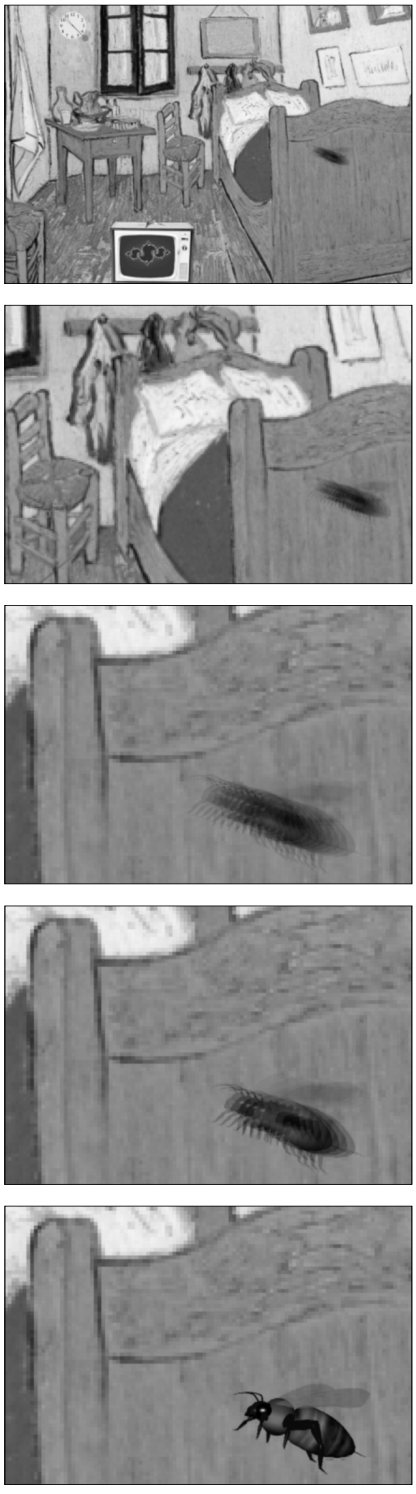


Figure 5.14*: Van Gogh room.*

in Section 5.3.1. The top three cells show increasing spatial detail, and the lower two cells show increasing "motion blur."

Figure 5.14 shows the result of arranging and compositing the many "clip-art" elements from the work area of the application shown in Figure 5.12 into a single multiresolution video, and then viewing this video at different spatial and temporal resolutions. The background is a still painting by Vincent Van Gogh. In the foreground are a moving clock on the wall, an animation of a flying bee, and a television framing the Julia set animation of Figure 5.13. The lower frames show zooming in spatially to see the bee up close, then slowing the video to reduce temporal blur. (Apologies to Van Gogh.)

Figure 5.15 shows *wind stress*, the force exerted by wind over the earth's surface, measured for 2000 days over the Pacific Ocean by the National Oceanographic and Atmospheric Administration (NOAA). Wind stress is a vector quantity, which we encoded in multiresolution video using hue for direction and value for magnitude. The left image shows a leaf time node (reflecting a single day's measurements), while the right image shows the root time node (reflecting the average wind stress over the 2000-day period). Note the emergence of the dark continents in the right image, which reveals the generally smaller magnitude of wind stress over land.

The left side of Figure 5.16 shows a frame from a computational fluid dynamics simulation in which two fluids (one heavy, one light) interact in a closed tank. The simulation method [LB96] adaptively refines its sample grid in regions where the function is spatially complex, so the resolution of the data is higher at the interface between the two fluids than it is in the large, constant regions containing just one fluid. This refinement also occurs in time, providing higher temporal resolution in areas that are changing rapidly. The right image shows a close-up of center region of the left image.

Figure 5.17 shows a simulation in which a galaxy is swept about a cluster of other astronomical bodies and is eventually ripped apart by their gravitational forces. The left image shows a close-up—late in the entire simulation— focused on the galaxy. The right image shows an even closer view of a single frame in which some whimsical high-resolution

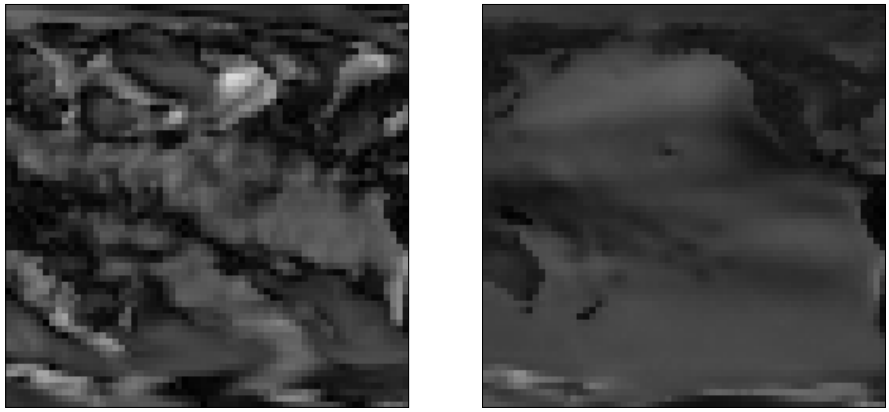


Figure 5.15: Wind stress over the Pacific Ocean.

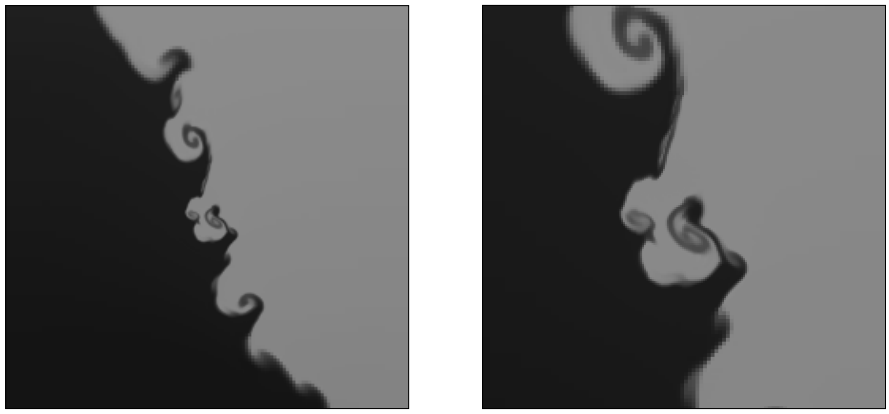


Figure 5.16: Fluid dynamics simulation.

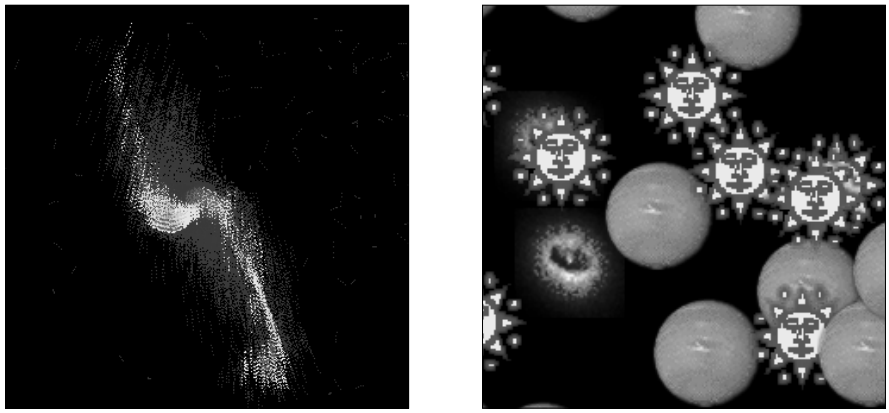


Figure 5.17: Astrophysical simulation of a galaxy.

Figure 5.18: Panoramic image for multiresolution video QuickTime VR, and two views in the scene.

detail has been added. (However, creating the entire video sequence at this level of detail would be prohibitively expensive.)

Finally, Figure 5.18 shows a QTVR panoramic image that we have adapted for use with multiresolution video QuickTime VR. Over the picture frame on the wall we have composited the entire Van Gogh room video from Figure 8.

Table 5.1 reports information about the storage space for the examples in Figures 5.13–5.18. The "Disk Size" column gives the total amount of space required to store the entire structure on disk, with averages and pointers included, after it has been compressed without loss using a Lempel-Ziv compressor [ZL77]. The next column, "Memory Size" gives the total space required in memory, including all averages, pointers, and flags. The "Unires Size" column reports the total space that would be required to store the raw RGBA pixel values, assuming the entire video had been expanded to its highest spatial and temporal res-

Table 5.1: Sizes (in Kb) of some example multiresolution video clips.

Figure	Video	Disk Size	Memory Size	Unires Size
7	*Julia set*	*23,049*	*58,926*	*67,109*
8	*Van Gogh*	*46,738*	*98,798*	*34,359,730*
9	*Wind stress*	*68,566*	*134,201*	*33,554*
10	*Fluids*	*40,091*	*106,745*	*536,870*
11	*Galaxy*	*37,222*	*315,098*	*137,438,953*
12	*Panorama*	*47,723*	*100,804*	*2,199,023,256*

olution present anywhere in the multiresolution video, but not including spatial or temporal averages. With the exception of the wind stress data, all of the video clips were smaller (in several cases much, much smaller) in the multiresolution video format than they would be in a uniresolution format, despite the overhead of the spatial and temporal averages. The wind stress data was difficult to consolidate because it has very little spatial or temporal coherence. The galaxy data compressed very well on disk because all of the colors stored in our structure (most of which were black) were selected from a small palette of very few colors.

5.6 Extensions

This chapter presents a new form of video capable of representing different amounts of detail in different parts of a video clip. Multiresolution video appears to be fruitful in a variety of applications. Yet a number of areas remain open for investigation. Among them are:

User-interface paradigms. As in the multiresolution image work of Berman *et al.* [BBS94], there is an important user-interface issue to be addressed: How does the user know when there is more spatial or temporal detail present in some part of the

video? We have considered changing the cursor in areas where there is more spatial detail present than is currently being displayed. Perhaps a timeline showing a graph of the amount of temporal detail present in different parts of the video would address the corresponding temporal problem.

Better compression. We currently require the up-links in our representation to point to a time-ancestor, primarily because coherence is fairly easy to discover this way. However, by relaxing this restriction—that is, by allowing up-links to point to any other place in the structure—we might be able to achieve much better compression, particularly for areas that have spatially repeating patterns. Unfortunately, finding the optimal set of up-links in this more general setting could be very expensive.

Spatial and temporal antialiasing. So far we only have used box-basis functions to represent the colors in multiresolution video. When the user zooms in to view a region at a higher spatial resolution than is present in the frame, large blocky pixels are displayed. Furthermore, if the user zooms in in time to view frames at higher temporal detail than is present in the video sequence, the motion is choppy. It would be interesting to explore the use of higher-order filters to produce smoother interpolations when the user views regions at higher resolution than is represented.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This dissertation has described multiresolution representations for curves, images and video, and demonstrated their utility in several application areas. As enumerated in Chapter 1, multiresolution representations enjoy a number of useful properties: control, feature detection, compression and refinement. These properties have been leveraged by the applications described in this document:

- **Control.** Our curves employ a multiresolution representation to provide control of curves at different levels of detail. Likewise, the editing algorithms for multiresolution video make use of similar control qualities.

- **Feature detection.** Our distance metric for images is based on the feature detection properties of a wavelet image representation. Furthermore, the ability to capture features from one curve and apply them to another curve also falls out of their multiresolution representations.

- **Compression.** The small "signatures" for images stored in our image database can be thought of as highly compressed images distilled through wavelet compression. Also, our multiresolution representation for video provides both lossless and lossy compression.

- **Refinement.** Multiresolution curves give the user the ability to add fine detail to a simple basic curve. Finally, our multiresolution video representation provides the same capability over video.

The conjunction of these properties empowers us to manage large data sets. With rapid increases in memory, storage and bandwidth, as well as the blossoming of the Internet and the advent of the World Wide Web, we now have access to ever-increasing amounts of information. As examples, the ability to manipulate shapes without worrying about a multitude of control points, the ability to quickly search through a large database of images, and the ability to view and manipulate large time-changing data sets all demonstrate the power of multiresolution representations. Thus, these representations have come to play a significant role among our tools for managing the information available to us.

The bulk of this dissertation has been devoted to three multiresolution applications, each of which has proven useful in its own domain. Multiresolution curves provide several useful operations on curves: smoothing, editing at different levels of detail, and approximation for scan conversion. The image querying application has proven to be a fast and effective way to find images in a large database. Finally, multiresolution video facilitates a variety of viewing and editing operations on video possessing different levels of detail both spatially and temporally.

The use of multiresolution representations is not without cost, however. The work described in this dissertation raises a number of concerns:

- Multiresolution representations are generally more complicated than uniresolution representations. Designing multiresolution data structures and algorithms appropriate to a new application takes thought and experimentation. For example, some number of person-months were spent on several inefficient representations for multiresolution video before we arrived at the one described here.

- While uniresolution representations can often make use of implicit data structures such as arrays, the sparse nature of many multiresolution data structures requires

storage overhead for pointers and flags. In the case of a wavelet representation, this storage overhead may be further exacerbated because wavelet coefficients can require extra precision over their uniresolution counterparts. However, these forms of storage overhead are usually compensated for by the compression aspects of multiresolution representations. The very pointers that are required in many sparse representations allow those representations to take advantage of coherence. Likewise, although wavelet coefficients may require more space to represent, wavelets lead to a natural form of lossy compression.

- There may be time concerns associated with multiresolution representations, especially if some form of synthesis is required in order to extract stored data. However, in some cases, it is the multiresolution nature of the representation that drives an efficient algorithm, for example the image querying application. Furthermore, all of the algorithms presented in this thesis are linear in the size of their inputs and outputs.

6.2 Wavelet caveats

Wavelets have recently become fashionable in the graphics community, and they have proven beneficial in a number of application areas. However, wavelets are not a panacea, and some care should be taken when one considers whether or not to use them. There are a variety of other multiresolution representations available. As discussed in Section 5.2.5, our first implementation of multiresolution video employed wavelets, but we subsequently moved to a different representation. Some of the drawbacks wavelets worth considering when undertaking a new project are enumerated in this section.

First, wavelet representations can be somewhat complex, perhaps increasing development time and probably making the solution a bit difficult to understand.

Second, wavelets may impose a diadic structure on the functions that they represent. For example, the wavelets developed in Chapter 3 were designed for curves with $2^j + 3$

control points, for some integer j. Curves with arbitrary numbers of control points must be converted before they can be represented in that basis.

Third, as the name "wavelet" suggests, wavelet basis functions usually wiggle up and down in some local region. (See Figure 3.1 on page 20.) When capturing a function that isn't represented well by the basis (for example, representing a sharp discontinuity using a smooth basis) these basis functions can yield "ringing."

Finally, because of many multiplications involved in the filterbank, it is prudent to represent wavelet coefficients using floating point numbers. For applications that require fixed point representations, pains must be taken to ensure that the wavelet coefficients do not lose too much precision.

Often these limitations are compensated by the many advantages of using wavelets. For example, the extra precision required by wavelets may be offset by the natural and efficient compression that they afford. Therefore, an analysis early in the project weighing the benefits and liabilities of wavelets may save some time later.

6.3 Future Work

The variety of applications described in this dissertation suggests a number of broad areas for future research, particularly some hybrid techniques.

Cursive handwriting recognition. Perhaps our image querying method could be combined with multiresolution curves to attack the problem of cursive handwriting recognition. Previous approaches to handwriting recognition have attempted to divide written words into small curve segments and then recognize words based on probable combinations of these segments. Here we consider treating each cursive word as a single curve, and distilling from it a wavelet-based "signature," as done for image querying. A brief investigation of this technique showed some promise, but also identified some of the challenge for handwriting recognition: features that give rise to similar shapes on a page for two curves representing the same word may actually occur at very different parametric locations within the curves.

Figure 6.1: Surface manipulation at different levels of detail. From left to right: original, narrow change, medium change, broad change.

Video querying. It would also be interesting to cross the image querying method with multiresolution video to address the problem of searching for a given frame or scene in a video sequence. The simplest approach would be to consider each frame of the video as a separate image in the database and to apply our method directly. A more interesting approach would be to explore using three-dimensional multiresolution decomposition of the video sequence, combined perhaps with some form of motion compensation in order to take better advantage of the extra coherence among the video frames.

Surfaces. The wavelet-based image querying method might be applicable to multiresolution surfaces to identify shapes in a library of 3D models. Furthermore, an important extension of the manipulation techniques for multiresolution curves would be to generalize them to surfaces. As a test of multiresolution surface editing, we built a simple editor that allows a user to modify a bicubic tensor-product B-spline surface [BBB87, Far92, HL92] at different levels of detail. Figure 6.1 shows several manipulations applied to a surface over 1225 control points modeling a human face. Note that tensor-product surfaces are limited in the kinds of shapes they can model seamlessly. Lounsbery *et al.* [LDW93] discuss a multiresolution representation for surfaces of arbitrary topology and Eck *et al.* [EDD+95] extended this work to meshes of arbitrary connectivity. Many of the techniques described in this chapter should extend directly to their surfaces as well. In particular, fractional-level display and editing are applicable in the same way as for curves and tensor-product sur-

faces. The compression technique for scan-converting curves might also be used for rendering simplified versions of polyhedra within guaranteed error tolerances.

BIBLIOGRAPHY

[Ado90] Adobe Systems, Incorporated. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, Inc., second edition, 1990.

[AGG+92] J. Adams, R. Garcia, B. Gross, J. Hack, D. Haidvogel, and V. Pizzo. Applications of multigrid software in the atmospheric sciences. *Monthly Weather Review*, 120(7):1447–1458, July 1992.

[BB89] R.H. Bartels and J.C. Beatty. A technique for the direct manipulation of spline curves. In *Proceedings of the 1989 Graphics Interface Conference*, pages 33–39, London, Ontario, Canada, June 1989.

[BBB87] R. Bartels, J. Beatty, and B. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann, 1987.

[BBS94] Deborah F. Berman, Jason T. Bartell, and David H. Salesin. Multiresolution painting and compositing. In *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, pages 85–90, July 1994.

[BC90] Michael J. Banks and Elaine Cohen. Realtime spline curves from interactively sketched data. *Computer Graphics*, 24(2):99–107, 1990.

[BCR91] G. Beylkin, R. Coifman, and V. Rokhlin. Fast wavelet transforms and numerical algorithms I. *Communications on Pure and Applied Mathematics*, 44:141–183, 1991.

[BEN+93] R. Barber, W. Equitz, W. Niblack, D. Petkovic, and P. Yanker. Efficient query by image content for very large image databases. In *Digest of Papers. COMPCON Spring '93*, pages 17–19, San Francisco, CA, USA, 1993.

[CG91] George Celnicker and Dave Gossard. Deformable curve and surface finite elements for free-form shape design. In *Proceedings of SIGGRAPH '91*, pages 257–265, July 1991.

[Che95] Shenchang Eric Chen. Quicktime VR—an image-based approach to virtual environment navigation. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 29–38, August 1995.

[Chu92] Charles K. Chui. *An Introduction to Wavelets*. Academic Press, Inc., Boston, 1992.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[CPD+96] Andrew Certain, Jovan Popovic, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive multiresolution surface viewing. In *SIGGRAPH 95 Conference Proceedings*, pages 91–98. ACM SIGGRAPH, August 1996. held in New Orleans, August 1996.

[CQ92] Charles K. Chui and Ewald Quak. Wavelets on a bounded interval. *Numerical Methods of Approximation Theory*, 9:53–75, 1992.

[CSS+96] Per H. Christensen, Eric J. Stollnitz, David H. Salesin, , and Tony D. DeRose. Global illumination of glossy environments using wavelets and importance. *ACM Transactions on Graphics*, 15(1):37–71, January 1996.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*. SIAM, Philadelphia, PA, 1992. Notes from the 1990 CBMS-NSF Conference on Wavelets and Applications at Lowell, MA.

[DJL92] R. DeVore, B. Jawerth, and B. Lucier. Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, 38(2):719–746, March 1992.

[DL92] M. Daehlen and T. Lyche. Decomposition of splines. In Tom Lyche and L. L. Schumaker, editors, *Mathematical Methods in Computer Aided Geometric Design II*, pages 135–160. Academic Press, New York, 1992.

[EDD+95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 173–182. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[Far92] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, third edition, 1992.

[FB88] D. Forsey and R. Bartels. Hierarchical B-spline refinement. *Computer Graphics*, 22(4):205–212, 1988.

[FB91] D. Forsey and R. Bartels. Tensor products and hierarchical fitting. In *Curves and Surfaces in Computer Vision and Graphics II, SPIE Proceedings Vol. 1610*, pages 88–96, 1991.

[FBF+94] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intel-*

ligent Information Systems: Integrating Artificial Intelligence and Database Technologies, 3(3-4):231–262, 1994.

[FJS96] Adam Finkelstein, Charles E. Jacobs, and David H. Salesin. Multiresolution video. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series. ACM SIGGRAPH, August 1996. held in New Orleans, Louisiana, 4-9 August 1996.

[Fow92] Barry Fowler. Geometric manipulation of tensor product surfaces. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, March 1992. Available as Computer Graphics, Vol. 26, No. 2.

[FS94] Adam Finkelstein and David H. Salesin. Multiresolution curves. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 261–268. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[Gal91] D. Le Gall. MPEG: A video compression standard for multimedia applications. *CACM*, 34(4):46–58, April 1991.

[GC94] Steven J. Gortler and Michael F. Cohen. Variational modeling with wavelets. Technical Report TR 456-94, Princeton University, April 1994.

[Gla95] Andrew S. Glassner. *Principles of Digital Image Synthesis*, volume 1. Morgan Kaufmann, San Francisco, 1995.

[GMG+92] William I. Grosky, Rajiv Mehrotra, F. Golshani, H. V. Jagadish, Ramesh Jain, and Wayne Niblack. Research directions in image database management. In *Eighth International Conference on Data Engineering*, pages 146–148. IEEE, 1992.

[GS93] T. Gevers and A. W. M. Smuelders. An approach to image retrieval for image databases. In V. Marik, J. Lazansky, and R. R. Wagner, editors, *Database and Expert Systems Applicatons (DEXA '93)*, pages 615–626, Prague, Czechoslovakia, 1993.

[GSCH93] Steven J. Gortler, Peter Schröder, Michael F. Cohen, and Pat Hanrahan. Wavelet radiosity. *Computer Graphics*, 27(3), August 1993. To appear.

[GZCS94] Yihong Gong, Hongjiang Zhang, H. C. Chuan, and M. Sakauchi. An image database system with content capturing and fast image indexing abilities. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 121–130. IEEE, 1994.

[HB94] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Addison-Wesley, Reading, Massachusetts, 1994.

[HB95] David J. Heeger and James R. Bergen. Pyramid-Based texture analysis/synthesis. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 229–238. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[HHK92] William M. Hsu, John F. Hughes, and Henry Kaufman. Direct manipulation of free-form deformations. *Computer Graphics*, 26(2):177–184, 1992.

[HK92] K. Hirata and T. Kato. Query by visual example — content based image retrieval. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology (EDBT '92)*, pages 56–71, Vienna, Austria, 1992.

[HL92] Josef Hoschek and Dieter Lasser. *Fundamentals of Computer Aided Geometric Design*. A K Peters, Ltd., Wellesley, Massachusetts, third edition, 1992.

[HL94] Siu Chi Hsu and Irene H. H. Lee. Drawing and animation using skeletal strokes. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 109–118. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[JFS95] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. Fast multiresolution image querying. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 277–286. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[JJS93] N. Jayant, J. Johnston, and R. Safranek. Perceptual coding of images. In *Proceedings of the SPIE — The International Society for Optical Engineering*, volume 1913, pages 168–178, 1993.

[Kat92] T. Kato. Database architecture for content-based image retrieval. In *Proceedings of the SPIE — The International Society for Optical Engineering*, volume 1662, pages 112–123, San Jose, CA, USA, 1992.

[KC94] Patrick M. Kelly and T. Michael Cannon. CANDID: Comparison Algorithm for Navigating Digital Image Databases. In *Proceedings of the Seventh Inter-*

national Working Conference on Scientific and Statistical Database Management Storage and Retrieval for Image and Video Databases. IEEE, 1994.

[KKOH92] T. Kato, T. Kurita, N. Otsu, and K. Hirata. A sketch retrieval method for full color image database — query by visual example. In *Proceedings of the 11th IAPR International Conference on Pattern Recognition*, pages 530–533, Los Alamitos, CA, USA, 1992.

[KZL94] Atreyi Kankanhalli, Hong Jiang Zhang, and Chien Yong Low. Using texture for image retrieval. In *International Conference on Automation, Robotics and Computer Vision*. IEE, 1994.

[KZT93] A. Kitamoto, C. Zhou, and M. Takagi. Similarity retrieval of NOAA satellite imagery by graph matching. In *Storage and Retrieval for Image and Video Databases*, pages 60–73, San Jose, CA, USA, 1993.

[LB96] Randy LeVeque and Marsha Berger. AMRCLAW: adaptive mesh refinement, 1996. `http://www.amath.washington.edu/~rjl/amrclaw/`

[LC93] J. Liang and C. C. Chang. Similarity retrieval on pictorial databases based upon module operation. In S. Moon and H. Ikeda, editors, *Database Systems for Advanced Applications*, pages 19–26, Taejon, South Korea, 1993.

[LDW93] Michael Lounsbery, Tony DeRose, and Joe Warren. Multiresolution surfaces of arbitrary topological type. Technical Report 93-10-05, University of Washington Department of Computer Science and Engineering, October 1993.

[LGC94] Zicheng Liu, Steven J. Gortler, and Michael F. Cohen. Hierarchical spacetime control. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando,*

Florida, July 24–29, 1994), Computer Graphics Proceedings, Annual Conference Series, pages 35–42. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[LK90] A. S. Lewis and G. Knowles. Video compression using 3D wavelet transforms. *Electronics Letters*, 26(6):396–398, 15 March 1990.

[LM87] T. Lyche and K. Morken. Knot removal for parametric B-spline curves and surfaces. *Computer Aided Geometric Design*, 4(3):217–230, 1987.

[LM92] T. Lyche and K. Morken. Spline-wavelets of minimal support. In D. Braess and L. L. Schumaker, editors, *Numerical Methods in Approximation Theory*, volume 9, pages 177–194. Birkhauser Verlag, Basel, 1992.

[Mad92] G. S. Maddala. *Introduction to Econometrics*. Macmillan Publishing Company, second edition, 1992.

[Mal89] Stephane Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[Man83] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Co., New York, rev 1983.

[MB95] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 39–46, August 1995.

[Mey94] David Meyers. Multiresolution tiling. *Computer Graphics Forum*, 13(5):325–340, December 1994.

[MR94] S. McCormick and U. Rude. A finite volume convergence theory for the fast adaptive composite grid methods. *Applied Numerical Mathematics*, 14(1–3):91–103, May 1994.

[MZ92] Stephane Mallat and Sifen Zhong. Wavelet transform maxima and multiscale edges. In Ruskai, et al, editor, *Wavelets and Their Applications*, pages 67–104. Jones and Bartlett Publishers, Inc., Boston, 1992.

[NBE+93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: Querying images by content using color, texture, and shape. In *Storage and Retrieval for Image and Video Databases*, pages 173–187. SPIE, 1993.

[NH88] Arun N. Netravali and Barry G. Haskell. *Digital Pictures*. Plenum Press, New York, 1988.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[PF93] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 57–64, August 1993.

[PFTF92] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Fetterling. *Numerical Recipes*. Cambridge University Press, second edition, 1992.

[Pin94] Brian Pinkerton. Finding what people want: Experiences with the WebCrawler. In *The Second International WWW Conference '94: Mosaic and the Web*, October 1994.

[PJ79] Mark Powell-Jones. *Impressionist painting*. Phaidon Press, Oxford, 1979.

[PS83] Michael Plass and Maureen Stone. Curve-fitting with piecewise parametric cubics. *Computer Graphics*, 17(3):229–239, July 1983.

[PSTT93] G. Petraglia, M. Sebillo, M. Tucci, and G. Tortora. Rotation invariant iconic indexing for image database retrieval. In S. Impedovo, editor, *Proceedings of the 7th International Conference on Image Analysis and Processing*, pages 271–278, Monopoli, Italy, 1993.

[PV95] Ken Perlin and Luiz Velho. Live paint: Painting with procedural multiscale textures. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 153–160, August 1995.

[QW93] Ewald Quak and Norman Weyrich. Decomposition and reconstruction algorithms for spline wavelets on a bounded interval. CAT Report 294, Center for Approximation Theory, Texas A&M University, April 1993.

[Rad93] Steven Radecki. *Multimedia With Quicktime*. Academic Press, 1993. ISBN 0-12-574750-0.

[RAFH92] T. R. Reed, V. R. Algazi, G. E. Forrd, and I. Hussain. Perceptually-based coding of monochrome and color still images. In *DCC '92 — Data Compression Conference*, pages 142–51, 1992.

[SABS94] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen–and–ink illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[Sam90] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[SAS89] SAS Institute Inc. *SAS/STAT User's Guide, Version 6, Fourth Edition, Volume 2*. SAS Institute Inc., 1989.

[Sch88] Philip J. Schneider. Phoenix: An interactive curve design system based on the automatic fitting of hand-sketched curves. Master's thesis, Department of Computer Science and Engineering, University of Washington, 1988.

[SDOW93] R. Shann, D. Davis, J. Oakley, and F. White. Detection and characterisation of Carboniferous Foraminifera for content-based retrieval from an image database. In *Storage and Retrieval for Image and Video Databases*, volume 1908, pages 188–197. SPIE, 1993.

[SDS96] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, San Francisco, 1996.

[SFAH92] E. P. Simoncelli, W. T. Freeman, E. H. Adelson, and D. J. Heeger. Shiftable multi-scale transforms. *IEEE Transactions on Information Theory*, 38:587–607, 1992.

[SI90] M. Shibata and S. Inoue. Associative retrieval method for image database. *Transactions of the Institute of Electronics, Information and Communication Engineers D-II*, J73D-II:526–34, 1990.

[SP94] P. S. Sathyamurthy and S. V. Patankar. Block-correction-based multigrid method for fluid flow problems. *Numerical Heat Transfer, Part B (Fundamentals)*, 25(4):375–94, June 1994.

[SS95a] Peter Schröder and Wim Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 161–172. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[SS95b] I. Suisalu and E. Saar. An adaptive multigrid solver for high-resolution cosmological simulations. *Monthly Notices of the Royal Astronomical Society*, 274(1):287–299, May 1995.

[SS95c] Jonathan Swartz and Brian C. Smith. A resolution independent video language. In *ACM Multimedia 95*, pages 179–188. ACM, Addison-Wesley, November 1995.

[SSG92] P. L. Stanchev, A. W. M. Smeulders, and F. C. A. Groen. An approach to image indexing of documents. *IFIP Transactions A (Computer Science and Technology)*, A-7:63–77, 1992.

[Swa93] Michael J. Swain. Interactive indexing into image databases. In *Storage and Retrieval for Image and Video Databases*, volume 1908, pages 95–103. SPIE, 1993.

[SZ94] Stephen W. Smoliar and Hong Jiang Zhang. Content-based video indexing and retrieval. *IEEE Multimedia*, 1(2):62–72, 1994.

[TC94] Chen Wu Tzong and Chin Chen Chang. Application of geometric hashing to iconic database retrieval. *Pattern Recognition Letters*, 15(9):871–876, 1994.

[TH94] Patrick C. Teo and David J. Heeger. Perceptual image distortion. In *Human Vision, Visual Processing and Digital Display V, IS&T/SPIE's Symposium on Electronic Imaging: Science & Technology*, 1994. In press.

[TP75] S. L. Tanimoto and Theo Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, 1993.

[Wic94] Mladen Victor Wickerhauser. *Adapted Wavelet Analysis from Theory to Software*. A. K. Peters, Wellesley, MA, 1994.

[Wil83] Lance Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):1–11, July 1983.

[WW92] William Welch and Andrew Witkin. Variational surface modeling. *Computer Graphics*, 26(2):157–166, 1992.

[ZL77] L. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform.Theory*, IT-23(3), May 1977.

VITA

Adam Finkelstein was born at the Stanford University Hospital on July 1, 1965, brought into the world by a doctor with one arm. The bulk of his youth was spent beyond the Orange Curtain in Southern California.

Adam was an undergraduate student at Swarthmore College (class of '87) where he studied physics and computer science, occasionally. In Pennsylvania he learned the value of wearing socks in the winter.

From 1987 to 1990, Adam was a software engineer at TIBCO (formerly Teknekron Software Systems) in Palo Alto. He wrote software for people who trade stock, and learned to windsurf.

In 1990, Adam became a computer science student at the University of Washington, where he acquired a Masters Degree, played a lot of ultimate frisbee, and finally earned his doctorate in 1996.

Early in 1997 Adam will join the Computer Science Department at Princeton University.