

1999

## Interactive Multireslution Curve Editing Using

Stephen P. Alberg  
*University of Rhode Island*

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Terms of Use

All rights reserved under copyright.

---

### Recommended Citation

Alberg, Stephen P., "Interactive Multireslution Curve Editing Using" (1999). *Open Access Master's Theses*. Paper 992.  
<https://digitalcommons.uri.edu/theses/992>

This Thesis is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact [digitalcommons-group@uri.edu](mailto:digitalcommons-group@uri.edu). For permission to reuse copyrighted content, contact the author directly.

INTERACTIVE MULTIREOLUTION  
CURVE EDITING USING  
WAVELETS  
BY  
STEPHEN P. ALBERG

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE  
IN  
COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1999

MASTER OF SCIENCE THESIS

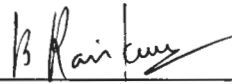
OF

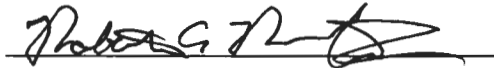
STEPHEN P. ALBERG

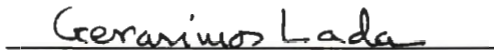
APPROVED:

Thesis Committee

Major Professor









DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

1999

## ABSTRACT

Precise curve fitting is an important feature of many computer applications, from statistical analysis tools, to the editors used by font and graphic designers, to the sophisticated computer-aided design/manufacturing environments developed for engineering systems. B-splines are the most widely used curve forms in such applications; composed of piecewise parametric cubic segments, they are notable for their compact representation, computational efficiency and, in particular, the high degree of continuity they enforce between successive curve segments. Such continuity, however, inhibits the freedom with which local, finer resolution editing may be performed on these curves. Refinement is most directly accomplished by inserting knots into the curve, subdividing the curve into a larger number of segments.

Multiresolution analysis, a form of data analysis based on the use of wavelets, offers a means of determining a *unique* such subdivision of a given curve. The application of this process is also reversible so that curve smoothing or knot removal operations may be performed with the same economy as refinement operations. Furthermore, the special computational properties of wavelets guarantee that such shifts of resolution may be performed in time linear with the size of the curve, suggesting that editing operations on a curve, at a variety of resolutions, may be done at interactive speeds.

## ACKNOWLEDGMENT

I would like to take this opportunity to thank the entire faculty of the Department of Computer Science and Statistics at the University of Rhode Island, all of whom I have had the honor and the privilege of working with or learning from at one time or another over my course of study. This has easily been the most challenging and, at the same time, the most rewarding academic experience I have participated in, and I've participated in a few. In this regard, I would in particular like to honor Dr. Bala Ravikumar for providing students like myself with a model example of patient intellectual inquiry and a love for the grand challenge.

## PREFACE

*Wavelets* are mathematical tools, functions satisfying a specific set of properties, which are used to encode information in a different, more practical form. The encoding takes the form of abstracting from the input data set a coarser, average distribution of the data, which is referred to in signal processing parlance as putting the data through a low-pass filter. At the same time, a high-pass filter, the set of wavelet functions themselves, is applied to the same input so that the high-contrast data lost from the averaging procedure may be preserved as a set of detail values. Such a procedure may be recursively reapplied to each successive coarser data set until a desired resolution of the data is reached. The result is termed a *wavelet transform* of the original data and it contains not only a coarse approximation to the original data set but the accumulated sets of detail coefficients that apply to that data set at different scales of resolution. These accumulated details may be reapplied to the coarsened data plot to retrieve the behavior of that data at finer scales of resolution.

Such a transformed image of an input data set provides views of that data at a variety of scales, providing a form of “mathematical microscope” [Hubb96]. This tool has been of practical benefit in a surprising variety of scientific disciplines. The term “wavelet” actually arose in connection with geological analysis where seismic plots are analyzed to determine the presence of substrata where regions of oil may occur. Signals and two-dimensional imagery may be “denoised” by abstracting out the high-contrast values using wavelets. The special properties satisfied by wavelets also allow them to be used in probability wave analysis

in quantum mechanics where they complement more conventional Fourier analysis tools in addressing the restrictions imposed by the uncertainty principle [Hubb96].

Researchers in computer graphics have recently found several practical uses for wavelets in their work, including both theoretical and practical applications in areas such as image compression, image editing and database querying, surface reconstruction from contour plots, and physical simulation for global illumination and animation. (For an excellent survey of this material, see Stollnitz, et al. [Stol96].)

Among this work is the research presented by Finkelstein and Salesin [Fink94] concerning the application of *multiresolution analysis*, a set of techniques that use wavelets to obtain information about a signal at a multitude of scales, to the problem of curve and, by extension, surface representation. In particular, they observe that the recursively defined structure of multiresolution analysis has an advantageous correspondence with the recursive procedure for subdividing curves (and surfaces) generated by the set of basis functions known as B-splines. The authors claim that the resulting representation, a multiresolution curve, is a unified framework capable of supporting a variety of editing operations including changing the “sweep” of a curve while maintaining its “character” or detail or, alternately, changing its “character” while maintaining its “sweep”, or applying continuous levels of smoothing or being able to edit a curve at a continuous level of detail. Furthermore, since a multiresolution decomposition can be obtained in time that is linear in the size of the input data, in this case the set of control points for a given curve, the manipulations described above may be performed in real time.

The goal of this thesis is to construct a working component capable of modeling multiresolution curves and the above operations in an interactive environment. Such a component, written in an object-oriented language, may be placed in a simple application



for editing such curves directly; or it may be readily integrated within a more complex editing environment, such as a CAD/CAM system, where it would offer a subset of a more general set of precise editing tools. We shall place this component within a simple GUI-based application frame in order to verify the claims of the original research that such curve operations may be performed at interactive speeds. Although this thesis does not extend the theoretical basis of this work, it does enhance its practical utility by realizing the application as a set of Java class files, making it available to any platform supporting a Java Virtual Machine and, at the same time, making this application available for use as a component in a larger Java-based graphical editing environment.

The presentation of the research supporting this implementation will involve four major sections: first, a discussion of the issues of curve representation in a computational environment, with some focus on the B-spline representation; second, a more detailed presentation of wavelets, featuring the prototypical example of the Haar wavelet, as well as an introduction to the concepts of multiresolution analysis; third, a presentation of the theory of multiresolution curve representation, featuring some discussion of the practical mathematical issues involved in this representation; and fourth, a detailed discussion of the design and implementation of the Java-based application, focusing in particular on how the demands of the mathematics involved are met by this application.



## TABLE OF CONTENTS

1. CURVE REPRESENTATIONS.....	1
1.1 Functions, Parametric Curves and Piecewise Polynomial Segments.....	2
1.2 Piecewise Parametric Cubic Curves.....	5
1.21 Hermite Forms.....	5
1.22 Bezier Curves.....	8
1.23 B-splines.....	10
1.24 Types of B-splines.....	14
1.3 Curve refinement.....	15
2. WAVELETS AND MULTIREOLUTION ANALYSIS.....	18
2.1 What are Wavelets?.....	18
2.11 Scaling functions and Wavelets.....	18
2.12 Example: the Haar Wavelet.....	20
2.13 Properties of Wavelets.....	22
2.14 Some Applications.....	24
2.2 Multiresolution Analysis.....	26
3. MULTIREOLUTION CURVES.....	30
3.1 B-splines and Spline Wavelets.....	30
3.2 Integral and Fractional levels of Resolution.....	33
3.3 Implications of Multiresolution Curve Theory.....	41

4.	IMPLEMENTATION DESIGN AND RESULTS.....	42
4.1	Overall Structure of Application .....	43
4.2	Description of Major Components .....	44
4.21	<i>Multiresolution Engine</i> .....	44
4.22	<i>The Application Frame</i> .....	55
4.23	<i>The Application Canvas</i> .....	60
4.3	Evaluation and Results .....	69
5.	CONCLUSIONS .....	74
	APPENDIX A: Endpoint-interpolating cubic B-spline matrices.....	78
	APPENDIX B: MATLAB code for B-spline wavelets. ....	80
	APPENDIX C: CurvEditor code listing.....	85
	BIBLIOGRAPHY .....	147

## LIST OF FIGURES

Figure 1: A function and a "non-function" .....	2
Figure 2: The parametric equation $x = \sin(t)$ , $y = \cos(t)$ , $0 < t < 10\pi$ .....	3
Figure 3: A Hermite curve representation .....	6
Figure 4: Two adjacent Hermite curves. ....	7
Figure 5: A Bezier curve. ....	8
Figure 6: Blending polynomials for Bezier curve.....	10
Figure 7: A B-spline segment (a) and the same segment with two neighboring segments on each side (b). ....	11
Figure 8: B-spline blending functions. ....	12
Figure 9: The four basis functions that define segment $i$ . ....	13
Figure 10: Editing a B-spline at a lower and higher resolution.....	15
Figure 11: The Haar basis for $V^0$ , $V^1$ and $V^2$ . ....	22
Figure 12: The Haar wavelets for $W^0$ , $W^1$ and $W^2$ . ....	23
Figure 13: A multiresolution filter bank.....	27
Figure 14: The B-spline scaling functions and the first four wavelets at level 3. [Fink94]...	32
Figure 15: Fractional-level curves: (a) level 8.0; (b) level 5.4; (c) level 3.1. [Fink94] .....	34
Figure 16: Changing the sweep of a curve without affecting its character. [Fink94].....	35
Figure 17: Changing the character of a curve without affecting its sweep. [Fink94].....	39
Figure 18: Orientation of detail in an edited curve. [Fink94] .....	40

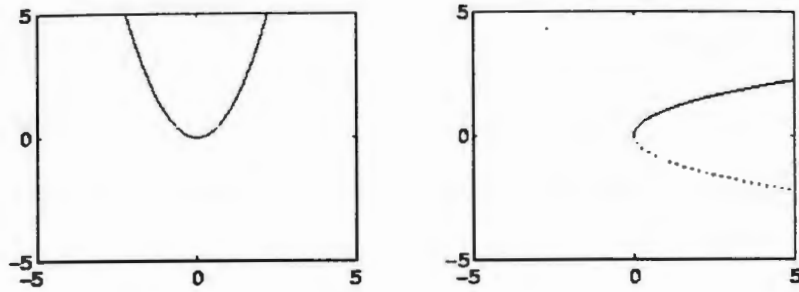
Figure 19: Block matrix diagram of synthesis filters P and Q. ....	46
Figure 20: Revised block matrix after interleaving. ....	47
Figure 21: Optimized banded diagonal form for block matrix PQ. ....	48
Figure 22: The CurvEditor interface.....	56
Figure 23: CurvEditor menu selection options. ....	58
Figure 24: A sample multi-segment curve showing the control points, polygon and knots. ....	63
Figure 25: Interpolation between a high-resolution curve (left) and a low-resolution curve (right). ....	67
Figure 26: Sample comparison of curves produced by (a) CurvEditor and (b) MATLAB. ....	70
Figure 27: Average execution time of (a) refine and (b) coarsen methods per resolution level (in milliseconds). ....	71

## 1. CURVE REPRESENTATIONS

A curve is essentially a set of points. More precisely, in a two-dimensional environment, we may refer to a curve as a set of ordered pairs. Mathematically, this gives us a very precise description of a curve. But the enumeration of such a set, typically drawn from the domain of the real numbers, will be unbounded and of little use in a finite computational environment. Furthermore, since such a set is unbounded, it cannot be exhaustive and a simple enumeration of a sequence of points will have no predictive value for those points omitted from the list. These observations provide the rationale for the procedure of *curve fitting* which attempts to find a relation between an independent variable and a set of points forming the closest approximation to the given sequence of points. This found relation is referred to as a *curve representation* and provides a more economical means for storing and describing the behavior of a given curve. In this section, we will present several forms of curve representations, evaluating their respective merits and defects with regard to the range of curves they are capable of modeling.

Before discussing these more economical representations, however, it should be mentioned that a finite set of points frequently serves as the direct representation of a curve in the form of a polygonal line or, in the three-dimensional case, as a polygon mesh [Fole96]. Such modeling is used when the individual data points are themselves subject to direct editing and often where determining an accurate representation of the curve or surface may be computationally intractable. (We shall see, in fact, that the actual rendering procedure used in our application basically draws just such a polygonal line connecting a sequence of points.) The procedures for determining a multiresolution curve cannot make use of such direct representations, however, so we will not discuss them further.





**Figure 1: A function and a "non-function"**

### 1.1 Functions, Parametric Curves and Piecewise Polynomial Segments

Since, by definition, a *function* is considered to be a set of ordered pairs (again, in a two-dimensional environment), this would seem at first glance to satisfy our basic requirements for an economical curve representation. A function describes a set of points of the form  $(x, f(x))$ , where  $x$  is an independent variable and  $f(x)$  is the output of the relation generating the curve. Functions are capable of generating a wide variety of curves, an example of which is the plot of  $f(x) = x^2$  shown in the left part of Figure 1. However, functions are restricted forms of relations in that for any single input  $x$  there must be a unique value  $f(x)$ . This makes the representation of curves such as the one shown in the right part of Figure 1 impossible, short of breaking the curve into independent segments. Functions are also not rotationally invariant (implied somewhat by Figure 1) and the description of curves with vertical tangents is hampered due to the fact that a slope of infinity is difficult to represent [Fole96].

Functions are sometimes referred to as *explicit representations*, as opposed to equations of the form  $f(x, y) = 0$  which are known as *implicit representations* [Ange97]. An example of

such an equation is the formula  $x^2 + y^2 - r^2 = 0$  which describes the set of points forming a circle around the origin with radius  $r$ . While clearly capable of modeling some curves that functions are unable to model, such equations are frequently underdetermined and may have more solutions than is practical. It is also difficult to use such forms to model a portion of a curve, say, for example, the top half of the circle just described, without applying external constraints to the calculation, in this case constraining  $y \geq 0$  [Ange97, Fole96].

A form of representation that overcomes the limitations of both the explicit and implicit forms is the *parametric representation*, where the elements of each ordered pair  $(x, y)$  are in fact functions on an independent variable  $t$ , meaning that a point should more properly be expressed in the form  $(x(t), y(t))$ . The variable  $t$  can be thought of as being plotted on an axis perpendicular to the  $x, y$  plane and the two-dimensional curve generated as the projection on this plane of the track of this variable as it proceeds along the  $t$  axis (see Figure 2).

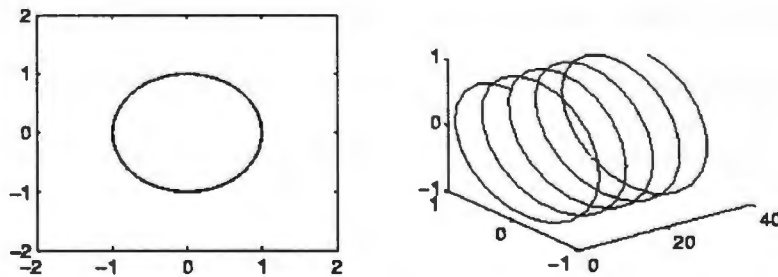


Figure 2: The parametric equation  $x = \sin(t)$ ,  $y = \cos(t)$ ,  $0 < t < 10\pi$ .

One advantage of parametric curves is that they replace the use of geometric slopes with parametric tangent vectors, which can never take on infinite values [Fole96]. This means that it is possible to take a single parametric curve, which may prove to be computationally difficult as a monolithic curve, and decompose it into a succession of parametric curve



segments. The points where adjacent segments join are referred to as *knots*. The plotting of the tangent vectors at the endpoints of each segment verifies the continuity of a curve form from segment to segment.

These segments are then individual parametric functions on  $t$ , forming a piecewise parametric curve. Computationally, the most convenient way for these segments to approximate a given curve is for each segment to model a polynomial function in  $t$ , typically with values of  $t$  on the interval  $[0,1]$ . The simplest such polynomials to compute would be piecewise linear curve segments, but the resulting “curve” would resemble a polygonal line, with typically not very good approximation over the length of a curve segment. Piecewise quadratic segments would be more supple in terms of modeling a given segment but, as we will see shortly, cannot guarantee adequate continuity from segment to segment [Ange97]. In practice, piecewise cubic parametric curves are most often used. Quartic curves are sometimes used in applications where higher-degree derivatives are needed to determine curves and surfaces that are aerodynamically efficient, such as in car and airplane design. However, these curves require more computation to determine the coefficients of each polynomial term and often produce additional “wobble” in the representation as well [Fole96].

Where piecewise parametric cubic curves are used, we have four unknown coefficients to determine the curve. There must therefore be four knowns coming into the computation of the curve to solve for these unknown coefficients. The next section examines several piecewise cubic curve representations differentiated by the manner in which these four “knowns” are introduced into the curve solution.

## 1.2 Piecewise Parametric Cubic Curves

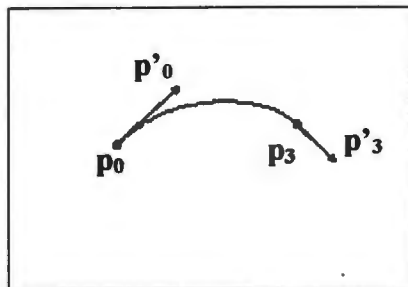
In the following, we shall be presenting alternate representations of the cubic curve segment  $q(t)$  denoted by

$$q(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} c_{0_x} + c_{1_x}t + c_{2_x}t^2 + c_{3_x}t^3 \\ c_{0_y} + c_{1_y}t + c_{2_y}t^2 + c_{3_y}t^3 \\ c_{0_z} + c_{1_z}t + c_{2_z}t^2 + c_{3_z}t^3 \end{bmatrix} = \mathbf{c}_0 + \mathbf{c}_1t + \mathbf{c}_2t^2 + \mathbf{c}_3t^3 = \sum_{k=0}^3 \mathbf{c}_k t^k$$

where  $0 \leq t \leq 1$ . Although we are concerned with two-dimensional curves, the parametric cubic is the lowest-degree curve that is non-planar in three dimensions [Fole96] and it is useful to indicate that the representations we are describing for two-dimensional curves are available for three-dimensional ones as well. In any event, the functions  $x(t)$ ,  $y(t)$  and  $z(t)$  are entirely independent of each other so, rather than having to find twelve equations to solve for twelve unknowns, we need only to find the four constraints that will determine the four coefficients for each of the equations in  $q(t)$ . These constraints are known as the *control points* of the curve segment and the following representations are differentiated by how they make use of this control information.

### 1.21 Hermite Forms

In all of the following representations, we are seeking a way to calculate the values of the control points  $\mathbf{p}_k$  given a parametric cubic segment with coefficients  $\mathbf{c}_k$  so that we can reverse the process, i.e.: determine  $\mathbf{c}_k$  given the control points  $\mathbf{p}_k$ . In the case of the Hermite representation (named for the mathematician), as shown in Figure 3, two of the



**Figure 3: A Hermite curve representation**

control points,  $\mathbf{p}_0$  and  $\mathbf{p}_3$  (they are indexed in this manner for consistency with the Bezier representation that we will see shortly), are marked by the actual endpoints of the curve segment. Since at one end of the segment  $t = 0$  and at the other end  $t = 1$ , we have the following values for  $\mathbf{p}_0$  and  $\mathbf{p}_3$ :

$$\begin{aligned}\mathbf{p}_0 &= \mathbf{q}(0) = \mathbf{c}_0 \\ \mathbf{p}_3 &= \mathbf{q}(1) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3\end{aligned}$$

The other two “points” in this representation are, actually, two tangent vectors passing through  $\mathbf{p}_0$  and  $\mathbf{p}_3$  which correspond to the first-order derivatives of the parametric curve segment at those points (also known as the “velocity” of the curve). These values,  $\mathbf{p}'_0$  and  $\mathbf{p}'_3$  are given by the following:

$$\begin{aligned}\mathbf{p}'_0 &= \mathbf{q}'(0) = \mathbf{c}_1 \\ \mathbf{p}'_3 &= \mathbf{q}'(1) = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3\end{aligned}$$

We may rewrite these equations in matrix form as the following:

$$\begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_3 \\ \mathbf{p}'_0 \\ \mathbf{p}'_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{bmatrix}$$

We observe that the matrix relating the coefficients to the set of control points in the previous expression is invertible. The inverse of this matrix is called the *Hermite geometry*

matrix  $M_H$

$$M_H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix}$$

and thus the formula for the parametric cubic curve is given by

$$q(t) = [1 \quad t \quad t^2 \quad t^3] M_H \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_3 \\ \mathbf{p}'_0 \\ \mathbf{p}'_3 \end{bmatrix}.$$

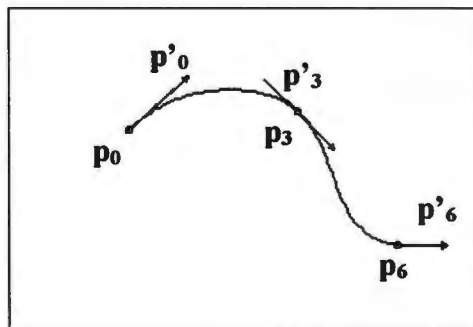


Figure 4: Two adjacent Hermite curves.

When two adjacent Hermite segments meet, as shown in Figure 4, not only do the endpoints of the adjacent segments match but the derivatives of both curves at that point match as well. The former condition is termed  $C^0$  continuity and the latter is termed  $C^1$  continuity. The superscript in both cases corresponds to the order of the derivative both curves share in common at that point. A similar property,  $G^1$  continuity, means that the

value of the tangent vector on one curve is proportional to the tangent vector of the neighboring curve.

### 1.22 Bezier Curves

Like the Hermite representation, the Bezier curve, named after Pierre Bezier, specifies the two endpoints of the segment as control points for the curve. The other two control points,  $\mathbf{p}_1$  and  $\mathbf{p}_2$ , control the ends of the tangent vectors through the endpoints of the curve. All four control points, then, describe a polygon within which the curve segment spans (see Figure 5).

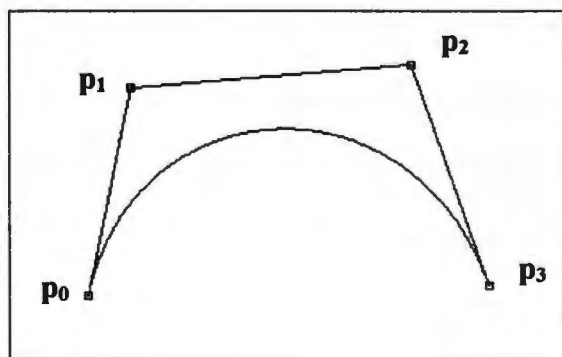


Figure 5: A Bezier curve.

The points  $\mathbf{p}_0$  and  $\mathbf{p}_3$  have the same relation to the coefficients of the curve as in the Hermite representation. The remaining points,  $\mathbf{p}_1$  and  $\mathbf{p}_2$ , are used to approximate the tangents at  $t = 0$  and  $t = 1$ , respectively. Using linear approximations and relating these to the derivatives of the polynomial, we have the following equations:

$$q'(0) = \frac{\mathbf{p}_1 - \mathbf{p}_0}{\frac{1}{3}} = 3(\mathbf{p}_1 - \mathbf{p}_0) = 3\mathbf{p}_1 - 3\mathbf{p}_0 = \mathbf{c}_1$$

$$q'(1) = \frac{\mathbf{p}_3 - \mathbf{p}_2}{\frac{1}{3}} = 3(\mathbf{p}_3 - \mathbf{p}_2) = 3\mathbf{p}_3 - 3\mathbf{p}_2 = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3$$



Taking these equations, we can solve as before to find the *Bezier geometry matrix*  $M_B$

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

and so the formula for the cubic Bezier curve is

$$q(t) = [1 \quad t \quad t^2 \quad t^3] M_B [p_0 \quad p_1 \quad p_2 \quad p_3]^T.$$

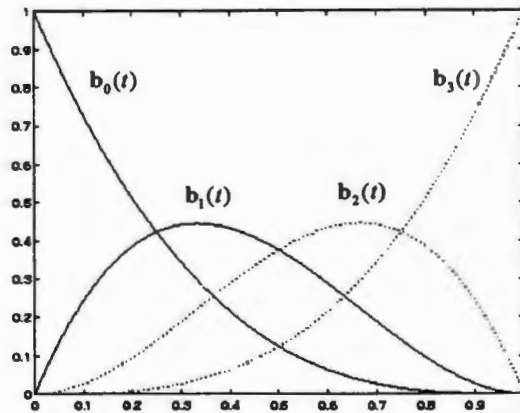
Unlike the Hermite representation, adjacent Bezier curve segments sharing an endpoint will obviously have  $C^0$  continuity but they do not enforce  $C^1$  continuity since different approximations are used to the left and the right of a join point [Ange97]. However, the Bezier curve satisfies an interesting property known as the *convex hull* property. To show this, we will first obtain the values for the blending functions of the Bezier curve, which are given by the transpose of the first two factors in the computation of  $q(t)$  above:

$$\mathbf{b}(t) = M_B^T [1 \quad t \quad t^2 \quad t^3]^T = \begin{bmatrix} (1-t)^3 \\ 3t(1-t)^2 \\ 3t^2(1-t) \\ t^3 \end{bmatrix}$$

The plot of these functions over the interval  $[0,1]$  is shown in Figure 6. Notice that all of the zeros occur when  $t = 0$  or  $t = 1$ . In addition, the value of each function over the interval is  $\leq 1$  and furthermore  $\sum_{i=0}^3 \mathbf{b}_i(t) = 1$ . This means that the representation of the polynomial

$$q(t) = \sum_{i=0}^3 \mathbf{b}_i(t) p_i$$

is a convex sum and that the entire curve lies within the convex hull described by the polygon formed by the control points of the curve, which is clear from Figure 5 [Ange97, Fole96].



**Figure 6: Blending polynomials for Bezier curve.**

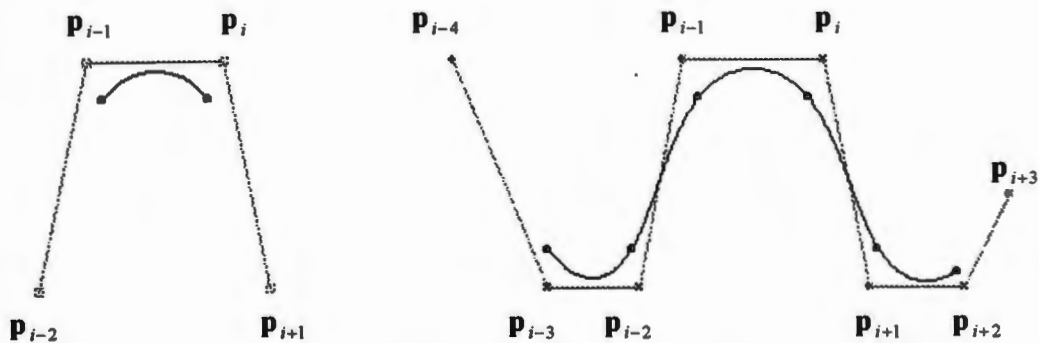
### 1.23 B-splines

The term *spline* originated with shipbuilding and the early days of aircraft design and referred to a long lath of wood or metal that was weighted with metal “ducks”. This was done to bend the spline judiciously for the purpose of producing a curve that could be traced and reproduced and that exhibited second-order or  $C^2$  continuity [Bart87]. Cubic B-splines also enforce  $C^2$  continuity (which is often referred to as the “acceleration” of the curve), as we shall see.

B-splines differ from Hermite and Bezier representations in a number of ways. First of all, the curve that the four control points describe typically does not interpolate (or pass through) any of the control points (see Figure 7a). Secondly, an individual segment must be considered in the context of its adjacent curve segments since three of the four control points defining a particular curve segment also participate in the definition of the neighboring segment (see Figure 7b). Thus, each control point in a B-spline representation can influence up to four adjacent curve segments and so, in a curve with  $m$  such segments, the number of



control points is  $m + 3$ , versus that of a Bezier or Hermite curve where the number of points is  $3m + 1$ .



**Figure 7: A B-spline segment (a) and the same segment with two neighboring segments on each side (b).**

Once again, we are interested in determining the coefficients  $c_k$  of the set of polynomials forming the parametric curve segment  $q_i(t)$  from the set of control points  $p_k$ . That is, we are looking for a matrix  $M$  such that

$$q_{i-1}(t) = [1 \quad t \quad t^2 \quad t^3] M \begin{bmatrix} p_{i-3} \\ p_{i-2} \\ p_{i-1} \\ p_i \end{bmatrix} \quad \text{and} \quad q_i(t) = [1 \quad t \quad t^2 \quad t^3] M \begin{bmatrix} p_{i-2} \\ p_{i-1} \\ p_i \\ p_{i+1} \end{bmatrix}.$$

We expect  $C^0$  and  $C^1$  continuity at the segment join points or *knots*. Therefore we have  $q_{i-1}(1) = q_i(0)$  and  $q'_{i-1}(1) = q'_i(0)$ . We also note that any conditions satisfying these constraints will not use  $p_{i-3}$  since it does not define  $q_i(t)$  nor will they use  $p_{i+1}$  since this point does not define  $q_{i-1}(t)$  [Ange97]. Bartels, et al. [Bart87] show that the following conditions suffice:

$$q_{i-1}(1) = q_i(0) = \mathbf{c}_0 = \frac{1}{6}(\mathbf{p}_{i-2} + 4\mathbf{p}_{i-1} + \mathbf{p}_i)$$

$$q'_{i-1}(1) = q'_i(0) = \mathbf{c}_1 = \frac{1}{2}(\mathbf{p}_i - \mathbf{p}_{i-2})$$

In addition, we obtain the following conditions at point  $q_i(1)$ :

$$q_i(1) = \mathbf{c}_0 + \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3 = \frac{1}{6}(\mathbf{p}_{i-1} + 4\mathbf{p}_i + \mathbf{p}_{i+1})$$

$$q'_i(1) = \mathbf{c}_1 + 2\mathbf{c}_2 + 3\mathbf{c}_3 = \frac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_{i-1})$$

Solving these equations for the coefficients  $\mathbf{c}_k$  gives us the *B-spline geometry matrix*  $M$ :

$$M = \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

B-splines, like Bezier curves, also satisfy the convex-hull property, as can be seen in

Figure 7 and in the plot of the blending functions  $\mathbf{b}_k(t)$  shown in Figure 8:

$$\mathbf{b}(t) = M^T \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix}^T = \frac{1}{6} \begin{bmatrix} (1-t)^3 \\ 4-6t^2+3t^3 \\ 1+3t+3t^2-3t^3 \\ t^3 \end{bmatrix}$$

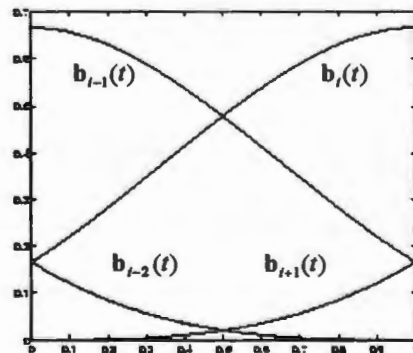


Figure 8: B-spline blending functions.

We note that, again, the only zeros occurring over the interval  $[0,1]$  are at  $t = 0$  and at  $t = 1$  and, in addition, only one blending function attains a zero value. What is more interesting about these functions is that when we juxtapose the blending functions for any adjacent segments with those of our first segment, we get the diagram shown in Figure 9. It is clear from this representation that every control point has associated with it a  $C^2$  piecewise cubic "hat" function, centered over the control point, and that this function is the same when shifted and applied to all of the control points in the curve representation. Thus, each curve segment is the sum of the values of each of these shifted blending functions over its respective interval. In other words, these blending functions form a basis for the polynomial curves thus described and in fact the "B" in B-spline stands for "basis" [Ange97, Bart87]. This will have important implications when we discuss multiresolution curve theory.

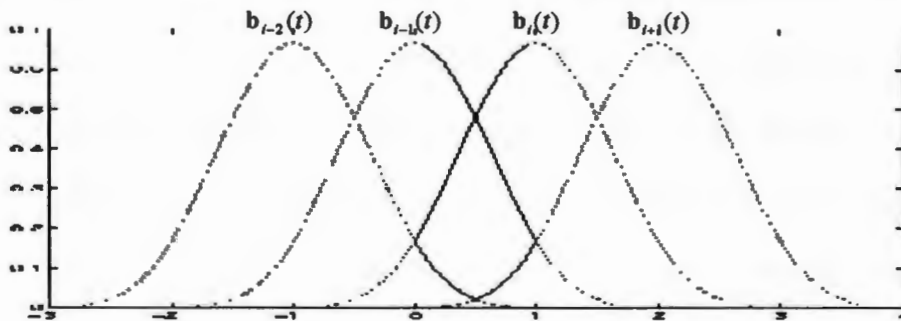


Figure 9: The four basis functions that define segment  $i$ .

Lastly, it should be noted that, since all of the above representations relate a set of control points to a set of coefficients through the use of matrix transformations, this means that we can convert from one representation to another using a composition of these matrices. Thus, all of these representations are equivalent, though of varying economy and precision.

### 1.24 Types of B-splines

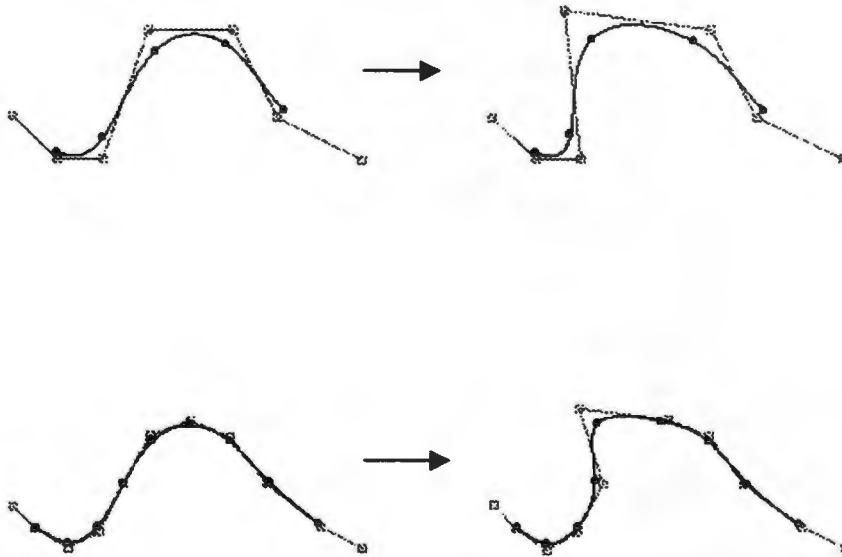
The blending functions described above are applicable to what are known as *uniform* B-splines, that is, those curves in possession of a uniform knot sequence. Such a sequence occurs when each knot in the curve is uniformly spaced and there are no multiplicities or repetitions of a knot at a certain point. This class of B-splines is a special case of the more general form of *non-uniform* B-splines, which admit discontinuities and multiplicities in the knot sequence. We note that, anytime a knot is repeated, this has the effect of pulling the curve closer to the corresponding control point. If a multiplicity of 3 or greater occurs, the knot(s) and control point coincide and the curve interpolates that control point. Under such conditions, the blending functions defined above must be redefined at the points these discontinuities occur. Both Foley, et al. [Fole96] and Bartels, et al. [Bart87] describe the set of recurrences required to redefine the blending functions over the affected intervals.

Both uniform and non-uniform B-splines, as described, are invariant under rotation, scaling and translation transformations. However, such curves are not invariant under perspective transformations. To guarantee this condition; the curve must be defined as a *non-uniform rational* B-spline or *NURBS* [Ange97, Fole96]. Such a curve plots each component of  $q_i(t)$  as a ratio between that component parametric function and another parametric function  $w_i(t)$ . The non-rational B-splines we have discussed would then be modeled as special cases where  $w_i(t) = 1$ . NURBS are a standard tool in most high-end graphic design environments and they account for most of the popularity in the use of B-spline representations in such environments.

With regard to representing multiresolution curves, we shall be using endpoint-interpolating B-splines, which are B-splines that have an overall uniform knot sequence

except at the beginning and ending of the curve where a multiplicity of at least 3 occurs in the sequence. This will necessitate the development of special end-conditions with regard to using the blending (basis) functions but the behavior of the curve away from the ends will be somewhat regular, which will be of great advantage in the final implementation of the multiresolution engine.

### 1.3 Curve refinement



**Figure 10: Editing a B-spline at a lower and higher resolution.**

One of the chief advantages of the B-spline representation is the high degree of continuity it enforces between adjacent curve segments. However, this becomes a drawback if local editing of a curve is desired since a single control point may affect up to four adjacent curve segments. The most direct way to overcome this problem is to refine the curve, introducing more segments into the curve through a procedure for knot insertion, as suggested by Figure 10.



Several algorithms for knot insertion exist in the literature. The simplest may be the deCasteljau algorithm for Bezier curves which consists of first finding the midpoints  $A$ ,  $B$  and  $C$  of the three lines forming the control polygon of the curve, then connecting these with the lines  $AB$  and  $BC$ . Next, find the midpoints of each of these lines. The line connecting these midpoints will also be tangent to the curve and the point at which it contacts the curve is the location of the new knot [Bart87, Fole96].

Farin [Fari88] and Bartels, et al. [Bart87] present some complex algorithms for inserting an arbitrary number of knots into a non-uniform B-spline. These algorithms have some efficiency if a group of knots is inserted at one time but do not perform as well when a sequence of edits is made. Since any knots may be added or removed from any location, there is also no correspondence between a regime of knot insertions and an increase in the objective "resolution" of the curve.

A scheme that attempts to correlate knot insertions with levels of resolution is the hierarchical B-spline refinement technique proposed by Forsey and Bartels [Fors88]. In this idea, an explicit hierarchical framework designed by the user is deployed to permit the editing of the overall form of a curve while preserving any details it acquired at higher levels of resolution. Besides the problem of maintaining the data structures necessary to model this hierarchy, however, the resulting curve will have an infinite number of possible representations.

The research presented by Finkelstein and Salesin [Fink94] improves upon this method by formulating a multiresolution curve representation that uses no additional structures other than the set of control points. Such a representation will implicitly model a hierarchy because of the manner in which finer detail edits are preserved within the representation using the techniques of multiresolution analysis. Furthermore, because the multiresolution

techniques recursively subdivide the curve at each increasing level of resolution, doubling the number of segments at each level, this results in a *unique* multiresolution representation. In order to understand how such a representation is possible we will, in the next section, survey the properties of wavelets and the role they play in multiresolution analysis.

In summary, then, among all of the curve representations we have surveyed, the B-spline is at the same time the most economical, in terms of the size of its data set, and the most precise, for the level of continuity it enforces across the segments of a curve. The high continuity enforced inhibits local level of control and so a systematic and efficient method for knot insertion is desirable for increasing the resolution of a curve. Multiresolution curve representations will meet these requirements, using techniques obtained from the theory of multiresolution analysis.



## 2. WAVELETS AND MULTIREOLUTION ANALYSIS

In this section, we will survey the properties of wavelets and the role that the wavelet transform plays in modeling a general theory of multiresolution analysis.

### 2.1 What are Wavelets?

The mathematical tools known as wavelets gained their initial reputation for success in the context of signal processing. It is difficult to describe what a wavelet is or what its significance is in isolation from such a context; wavelets basically serve as catalysts in such settings, bringing to light certain interesting properties. Compounding this difficulty is the fact that even the most introductory of materials on wavelets relies on an exposition that draws on mathematics of considerable depth. (See, for example, [Chui92] and the “optimistically titled” *A Friendly Guide to Wavelets* [Kais94].)

In this section, then, we will not attempt to present any of the deeper results from the theory of wavelets or provide any validation for some of its stronger claims. Rather, following the style of presentation in [Fink94] and [Stol96], we will accept certain claims as facts and will make use of these in presenting the more salient aspects of the wavelet transform with regard to modeling a multiresolution analysis.

#### 2.11 *Scaling functions and Wavelets*

Most introductions to wavelets [Hubb96, Chui92] preface their discussion of this material by presenting, for contrast and context, another tool used in signal processing for discovering the underlying behavior of a signal which is Fourier analysis. Briefly explained, a Fourier transform takes a given signal or data set and rewrites this signal as a linear combination of sine and cosine basis functions. It is, in fact, a striking fact that nearly all

signals may be decomposed in such a manner. The information conveyed by this transform, the composition of the component frequencies underlying a given signal, unfortunately does not yield any time-dependent details about the signal, just as the original signal itself, with its time-specific events, yields no information about its composite frequencies. A modified version of the Fourier transform, known as the Windowed Fourier Transform, is sometimes used to discover time-related signal behaviors at a more refined level. This version of the transform applies a window to a selected interval of the original signal and essentially performs a Fourier transform on that windowed portion of the signal, obtaining frequency data on that selected interval.

To describe the operation of a wavelet transform let us first assume, without loss of generality, that  $n = 2^m$ , for integer  $m$ , is the size of the signal or data set. A wavelet transform, rather than rewriting the original signal as a linear combination of basis functions, uses two sets of basis functions, one set to coarsen the signal to a lower resolution, the other set to disclose the events of significance at that same scale. The functions in the former group are referred to as *scaling functions*. Although they form a basis, these functions are designed for compact support, i.e.: the output of an individual function is nonzero over only a restricted or bounded interval. The formation of a basis consists then of “sliding” a specific instance of this function over distinct sub-intervals of the domain. They are called scaling functions because to coarsen a signal representation, for example, the individual instance of the sliding function is dilated (expanded) to admit twice the interval over the domain as that of the next finer level of resolution. Notationally, the  $i$ -th scaling function of the set of functions at resolution level  $j$  is expressed as  $\phi_i^j(x)$  where  $i = 0, \dots, 2^j - 1$  and  $j = 0, \dots, m$  and the vector space spanned by these basis functions is denoted  $V^j$ .

Since the coarsening of a signal representation removes data from that representation, it is desirable to retain this data in some form in the event we want to reconstitute the signal at its original resolution. This data is preserved by the other set of basis functions discussed above in the form of a set of detail or difference values, one for each data point removed to produce the coarser signal representation. These basis functions are defined to span the vector space  $W^j$  which is the orthogonal complement to the space  $V^j$  under the inner product  $\langle f | g \rangle$ ; that is, for each  $f$  in  $V^j$  and each  $g$  in  $W^j$ , the condition  $\langle f | g \rangle = 0$  must be true, where typically

$$\langle f | g \rangle = \int_0^1 f(x)g(x)dx.$$

It is these basis functions, those spanning  $W^j$ , that are the functions formally defined as *wavelets*. Notationally, the  $i$ -th wavelet of the set of functions at resolution level  $j$  is expressed as  $\psi_i^j(x)$  where  $i = 0, \dots, 2^j - 1$  and  $j = 0, \dots, m$ .

### 2.12 Example: the Haar Wavelet

An example may help to make the preceding discussion somewhat clearer. Suppose we are given the following "signal":

$$[9 \ 7 \ 3 \ 5]$$

This may be a one-dimensional image of pixel values with an original resolution value of 4. To coarsen this image by one level, we lower the resolution by one half, averaging the pixel values pairwise over the dilated intervals. This leads to the following lower-resolution image:

$$[8 \ 4]$$

where 8 is the average of the first pixel pair and 4 the average of the last pair. At the same time we produce this coarsening of the image, we wish to preserve the original information

contained in the four-value image. We save this information in the form of detail coefficients, which capture the amount of the difference of the old values from the new average. For this first pass, we have the value 1 for the first detail coefficient since the original value 9 is 1 more than the average 8 and the original value 7 is 1 less. Similarly, we obtain -1 as the other detail coefficient since the original value 3 is -1 more than the average 4 and 5 is -1 less. We may repeat this procedure recursively on the new low-resolution image to obtain the final one-dimensional image

$$[6]$$

and saving the detail coefficient 2 since 8 is 2 greater than the average 6 and 4 is 2 less.

Note that at each level change, the size of the data set of the coarser image is the same as the number of detail coefficients preserved and the sum of both these quantities equals the size of the higher-resolution image. That means that at each stage, we can convert from the high to the low-resolution image and save the detail information all within a set of data of constant dimension. If we preserve the image in this way, we have the following sequence of transformed signals:

$$[9 \ 7 \ 3 \ 5]$$

$$[8 \ 4 \ | \ 1 \ -1]$$

$$[6 \ | \ 2 \ 1 \ -1]$$

The last transformed image, with the single pixel value standing for the overall signal followed by the list of detail coefficients, is known as the *wavelet transform* of the image, using for this example the one-dimensional Haar basis.

The Haar basis, incorporating the set of scaling functions and the set of wavelets, is the simplest known wavelet basis. The Haar scaling functions may be more formally expressed as:

$$\phi_i^j(x) = \phi(2^j x - i) \quad i = 0, \dots, 2^j - 1 \quad \text{where} \quad \phi(x) = \begin{cases} 1 & 0 \leq x \leq 1 \\ 0 & \text{else} \end{cases}$$

Figure 11 shows the Haar or box basis for the spaces  $V^2, V^1$  and  $V^0$ .

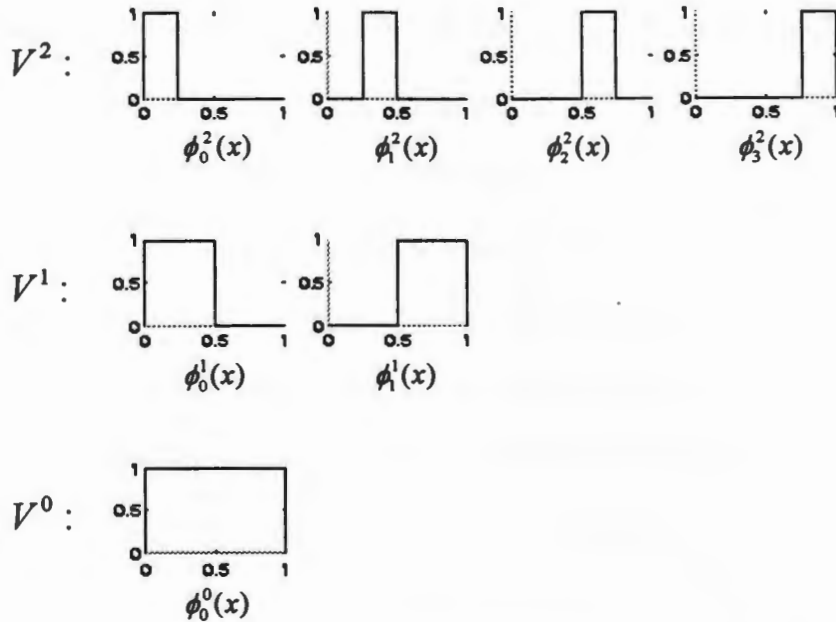


Figure 11: The Haar basis for  $V^0, V^1$  and  $V^2$ .

Likewise, the Haar wavelets, which are orthogonal to the set of basis functions, are defined as follows:

$$\psi_i^j(x) = \psi(2^j x - i) \quad i = 0, \dots, 2^j - 1 \quad \text{where} \quad \psi(x) = \begin{cases} 1 & 0 \leq x < 1/2 \\ -1 & 1/2 \leq x < 1 \\ 0 & \text{else} \end{cases}$$

Figure 12 shows the Haar wavelets for  $W^0, W^1$  and  $W^2$ .

### 2.13 Properties of Wavelets

In the foregoing analysis of the Haar basis we have already observed that the wavelet transform is an information-preserving decomposition which maintains a data set of constant size at all levels of the decomposition. It should also be noted that this is clearly a reversible



decomposition as well. The following are two mathematically equivalent expressions for the value of the same image:

$$\mathfrak{I}(x) = c_0^2 \phi_0^2(x) + c_1^2 \phi_1^2(x) + c_2^2 \phi_2^2(x) + c_3^2 \phi_3^2(x) \quad \text{where } c_0^2, \dots, c_3^2 = [9 \ 7 \ 3 \ 5] \text{ and}$$

$$\mathfrak{I}(x) = c_0^0 \phi_0^0(x) + d_0^0 \psi_0^0(x) + d_1^0 \psi_0^1(x) + d_1^1 \psi_1^1(x) \quad \text{where } c_0^0 = [6], d_0^0 = 2, d_0^1, d_1^1 = 1, -1$$

The first expression simply multiplies the original pixel values by the Haar scaling function relating one pixel to one interval. The last expression first applies the average value using the Haar scaling function relating four pixels to one interval, then applies the first set of differences to that average, and lastly it applies the remaining differences to the previously changed image, resulting in the original set of pixel values again. Implicit in these expressions as well is the idea that the higher-resolution function space is the same as the lower-resolution function and wavelet spaces combined, that is

$$V^j + W^j = V^{j+1}.$$

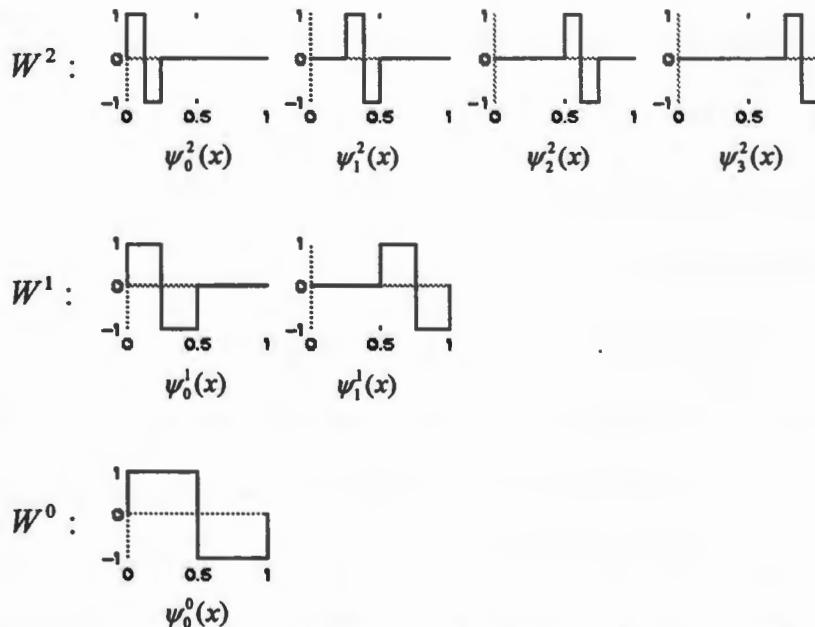


Figure 12: The Haar wavelets for  $W^0$ ,  $W^1$  and  $W^2$ .

The last point of interest concerns the efficiency of the procedure for producing the wavelet transform. The transformation on the original set of pixels required operations of both the scaling functions and the wavelet functions on all  $n$  pixels. After this first transformation, a coarser set of pixels of size  $n/2$  and a set of detail coefficients of size  $n/2$  remain. The transformation may now be recursively applied on the coarser image representation, which is now half the size of the original input. The set of detail coefficients already obtained plays no part in this lower scale transformation, however, so the size of the sub-problem is strictly  $n/2$ . Thus, if we continue to recursively reapply the transform algorithm to the coarsened image at each level until we reach the base case of the single average value, the number of steps we will take to completely transform the image is

$$n + \frac{n}{2} + \frac{n}{4} + \dots + 4 + 2 + 1 \leq 2n$$

And so, the time it takes to produce a wavelet transform is linear in the size of the original data set.

### *2.14 Some Applications*

The “image” used in our illustration of the Haar basis was somewhat small and it also generated detail coefficients of relatively high value. A more real-world instance of this problem would involve an image with a much larger number of pixels (say, a large power of 2). Suppose that this image is converted into a wavelet transform. If we examine the accumulated detail values in this transform, we are likely to discover that a number of these detail values are either 0 or are fairly close to 0, either on the positive or negative side. What this means is that, in the process of averaging that generated the coarsened image at that level, no change was registered between the image at that level and the next finer level. It is obvious, then, that such values may be left out of the representation with impunity since



their contribution to restoring the overall image is minimal to nil. This is the basis of the idea of wavelet compression, which is clearly a lossy compression technique. An obvious application of this technique would involve real-time networked video: since wavelet transforms take time around twice the size of the data set to perform, one could simply drop every other video frame in a feed, compress it by performing the transform and dropping a set number of detail coefficients off of the result, send the reduced frame, pad the message with zeros and then reconstitute the frame by reversing the transform.

Another interesting use of wavelets is in the area of fingerprint categorization. While individual sets of fingerprints are unique, they do evince well-known sets of patterns (e.g.: whorls, loops, etc) which are used as the basis for cataloguing and identifying fingerprints and fingerprint owners. A wavelet transform of a fingerprint image taken to a certain level of coarseness would provide, in numerical form, a similar kind of general pattern signature for a fingerprint, making it a candidate for matching other such signatures with the same classification. This has proven to be such a successful notion in practice that the FBI actually employs wavelet-based fingerprint analysis tools as part of its database environment [Hubb96].

Another interesting use of wavelet signatures, discussed in [Stol96], involves the use of a wavelet transform of an image as a query to find matches in an image database. For example, if one is searching for a picture of a sunset over a blue ocean, one might sketch a round red sphere for the sun above a larger blue volume for the ocean in a picture or bitmap editor. Then, the wavelet transform of this image is obtained and it is cross-checked against elements in the database, reporting scores on possible matches and, in the application presented, delivering thumbnail representations of these matches to the user for possible selection.

## 2.2 Multiresolution Analysis

The preceding material on the generation and the properties of the wavelet transform finds its codification in the development of the general framework of multiresolution analysis. The motivation for the development of this analysis, as narrated in the canonical paper on multiresolution analysis written by Stephane Mallat [Mall89], was a computer vision problem: how to extract meaningful data at an arbitrary scale from two-dimensional imagery. The goal of multiresolution analysis, as stated, is to provide a decomposition that enables a “scale-invariant interpretation” of an image. Of course, we may substitute the word “image” with any meaningful set of data under consideration.

The prerequisite for a multiresolution analysis to occur is the existence within the domain of the representation of a nested set of linear spaces. Such a hierarchy of spaces is possible only when the set of scaling basis functions spanning a given space  $V^j$  is refinable, i.e.: for all  $j$  in  $[1, m]$  there must exist a matrix  $P^j$  such that

$$\Phi^{j-1} = \Phi^j P^j$$

where  $\Phi^j$  is the set of scaling basis functions at level  $j$ . It can be shown that all functions which can be subdivided are refinable, that is the sets of functions at neighboring levels of resolution can be related in this way. Another way to express the idea of this nested hierarchy of spaces is in the form

$$V^0 \subset V^1 \subset V^2 \subset \dots$$

Note that, since  $V^j$  and  $V^{j-1}$  have dimensions  $v(j)$  and  $v(j-1)$ , respectively,  $P^j$  is a  $v(j) \times v(j-1)$  matrix. Similarly, since  $V^{j-1} + W^{j-1} = V^j$ , we can express the set of wavelets at level  $j-1$ ,  $\Psi^{j-1}$ , as linear combinations of the scaling functions  $\Phi^j$ . That is, there must exist a  $v(j) \times w(j-1)$  matrix of constants  $Q^j$  satisfying

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

For example, we can denote the matrices  $P^2$  and  $Q^2$  which perform the refinement from levels 1 to 2 of the Haar basis as the following

$$P^2 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad Q^2 = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}$$

Note that, if we conjoin these matrices in block matrix form, they form a square matrix that is also invertible, which will have some important implications, as we will see shortly. In fact, we may express the general equations stated above in block matrix form as well:

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

This is referred to as a *two-scale relation* for scaling functions and wavelets and the matrices  $P^j$  and  $Q^j$  are known as *synthesis matrices*.

If the preceding precondition regarding the existence of such nested spaces is met, the master strategy of a multiresolution analysis is to deploy a filter bank for transforming an input image into a wavelet transform. A filter bank for converting an input image  $\mathbf{c}^m$  into a wavelet transform would be diagrammed as follows:

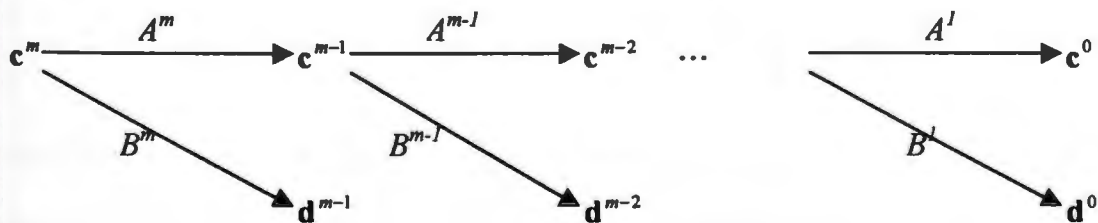


Figure 13: A multiresolution filter bank.

In order to build this filter bank, we must find *analysis matrices*  $A^j$ , the low-pass, downsampling filter that coarsens the signal, and  $B^j$ , the high-pass filter that records the details of significance at this scale, such that

$$\mathbf{c}^{j-1} = A^j \mathbf{c}^j \quad \text{and} \quad \mathbf{d}^{j-1} = B^j \mathbf{c}^j$$

where  $A^j$  is a  $v(j-1) \times v(j)$  matrix and  $B^j$  is a  $w(j-1) \times v(j)$  matrix. The contents of the transformed image, then, begin with the base set of coefficients  $\mathbf{c}^0$  and proceed with all of the  $\mathbf{d}^j$  from right to left along the bottom of the diagram.

If these matrices are chosen appropriately, then the original set of coefficients  $\mathbf{c}^j$  can be recovered from  $\mathbf{c}^{j-1}$  and  $\mathbf{d}^{j-1}$  by using the synthesis matrices  $P^j$  and  $Q^j$  as previously defined:

$$\mathbf{c}^j = P^j \mathbf{c}^{j-1} + Q^j \mathbf{d}^{j-1}$$

This works because the coefficients being modified are the linear multipliers for the basis functions already related by  $P^j$  and  $Q^j$  and so they can be directly related in this fashion.

This observation gives us a way to express the inverse relation of the scaling and wavelet functions at adjacent levels. Using the analysis filters  $A^j$  and  $B^j$  and the fact that  $v(j-1) + w(j-1) = v(j)$ , we can build the block matrix  $\begin{bmatrix} A^j & B^j \end{bmatrix}^T$  to obtain the following:

$$\begin{bmatrix} \Phi^{j-1} & \Psi^{j-1} \end{bmatrix} \begin{bmatrix} A^j \\ B^j \end{bmatrix} = \Phi^j$$

Like the block matrix previously described for combining the synthesis matrices, this square matrix is also invertible. In fact, it is clear from comparing both of these equations that

$$\begin{bmatrix} A^j \\ B^j \end{bmatrix} = \begin{bmatrix} P^j & Q^j \end{bmatrix}^{-1}.$$

For example, the analysis matrices for the Haar basis at level 2 are as follows:



$$A^2 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \text{ and } B^2 = \frac{1}{2} \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

If these are combined in block matrix form and inverted, the result will be the block matrix  $[P^2 \mid Q^2]$ . The following are some useful identities obtainable from this result:

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

It should be pointed out that the Haar basis is one of the rare wavelet bases where the set of wavelets at a given level is orthogonal not only to the set of scaling functions at that same level but also orthogonal to the scaling functions at every coarser level of resolution as well. This is known as an *orthogonal multiresolution basis* and has the nice result that the block matrix forming the set of analysis filters for any level  $j$  is actually the transpose of the block matrix forming the set of synthesis filters at that same level (within a scaling factor), that is

$$[P^j \mid Q^j]^T = [P^j \mid Q^j]^{-1}.$$

This happy result is not the norm for most wavelet bases. The only restriction for this majority of cases is that the spaces  $V^j$  and  $W^j$  be orthogonal to each other at that level which means that, once the synthesis filters  $P^j$  and  $Q^j$  have been determined from the refinement relations between the wavelets and the scaling functions at each level  $j$ , we can invert the block matrix holding the synthesis filters to obtain the block matrix containing the analysis filters.

To sum up, multiresolution analysis leverages the properties of the wavelet transform to produce a linear-time decomposition of an image, obtaining a scale-invariant representation of that image. Perhaps most important is that once the filter bank effecting this analysis has been defined, data representations of different scales may be directly related by these filters without recourse to any of the scaling functions or wavelets that built them in the first place.



### 3. MULTIREOLUTION CURVES

In this section, we will combine the analysis techniques described in the previous section with the curve representation for endpoint-interpolating B-splines obtained in section 1 to produce a unified representation for a multiresolution curve. The material in this section is almost entirely drawn from [Fink94] and [Stol96].

#### 3.1 B-splines and Spline Wavelets

In order for a curve representation to be modeled at a number of different scales, by the requirements of multiresolution analysis the set of basis functions spanning the vector space containing the finest scale representation of the curve must be refinable. For a cubic B-spline, the basis functions in this case correspond to the set of blending functions from which each curve segment is formed. From Figure 9, in fact, we can observe that the piecewise cubic "hat" function, whose pieces form the blending functions over a specific segment, behaves in exactly the manner we would expect from a scaling function. That is, the same function, which is nonzero only over a bounded interval, is shifted to cover successive and, in this case non-distinct, intervals, where the maximum value of the function is centered over the point along the curve where the corresponding control point exerts the greatest influence. Lastly, we know there are several algorithms for knot insertion or for subdividing a given curve. Since the curve representation is capable of subdivision, this means its basis functions are refinable and so the cubic B-spline representation is a candidate for multiresolution analysis.

The task now is to generate the synthesis filters  $P$  and  $Q$  for each level of resolution. This step is somewhat complicated due to the fact that we are using endpoint-interpolating B-splines, meaning that we have multiplicities at both ends of the knot sequence for our

curves. The usual method for finding the blending functions for a curve with a non-uniform knot sequence is to compute them using the Cox-deBoor recurrence [Fari88]:

$$\mathbf{b}_i^0(t) = \begin{cases} 1 & t_i \leq t \leq t_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{b}_i^d(t) = \frac{t - t_i}{t_{i+d} - t_i} \mathbf{b}_i^{d-1}(t) + \frac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} \mathbf{b}_{i+1}^{d-1}(t)$$

where the values  $t_i$  are the indexed knots in the knot sequence and the subscript  $d$  refers to the degree of the polynomial; thus, this recurrence must be called up to three levels to find the appropriate blending values for a piecewise cubic polynomial. Note that the definition of the basis function for the base case of the recurrence, i.e.: with  $d = 0$ , looks similar to the definition of the box function in the Haar basis. Unsurprisingly, it is the same function since the Haar basis is a piecewise constant B-spline. Once the values of the blending functions are determined, the  $P$  synthesis matrix may be built, encoding each B-spline as a linear combination of B-splines that are half as wide. We note that, for each level  $j$ , the number of segments in the curve representation is  $2^j$  and so the number of control points is  $2^j + 3$ . Thus, the matrix  $P^j$  has  $2^j + 3$  rows by  $2^{j+1} + 3$  columns. After finding  $P^j$ , a matrix  $Q^j$  is found which satisfies the equation

$$(P^j)^T \left[ \langle \Phi^j | \Phi^j \rangle \right] Q^j = \mathbf{0}$$

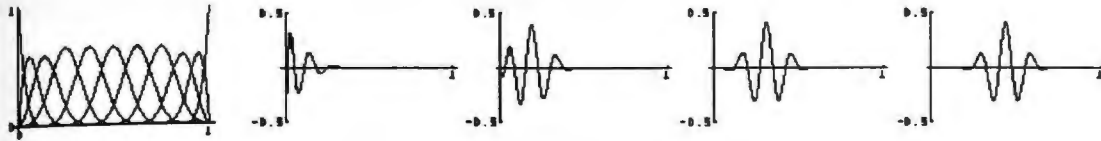
where  $\left[ \langle \Phi^j | \Phi^j \rangle \right]$  is the matrix of inner products of all the basis functions in  $\Phi^j$ . In

essence, the problem is to find  $2^{j+1}$  column basis vectors that can span the nullspace

$(P^j)^T \left[ \langle \Phi^j | \Phi^j \rangle \right]$ . There are many ways to select such vectors but, in practice, the best

way is to constrain the number of nonzero entries in each column and require these entries to be consecutive. Putting as many zeros as possible at the top and bottom of each column

will guarantee that the wavelets will have compact supports. Appendix B contains MATLAB code for the routines that compute the  $P$  and  $Q$  matrices for any input arguments  $d$ , for the degree of the polynomial, and  $j$ , for the level of resolution. Figure 14 shows the plots of the scaling functions and the first four wavelets at resolution level 3.



**Figure 14: The B-spline scaling functions and the first four wavelets at level 3.**  
[Fink94]

The filter banks for cubic endpoint-interpolating B-splines produced by these routines are shown in Appendix A for the first few levels of resolution. The impact of the non-uniform behavior of the basis functions at the endpoints is revealed in the irregular pattern of values in the first few and last few column vectors in both the  $P$  and  $Q$  matrices. However, above level 3 for the  $P$  matrix and level 4 for the  $Q$  matrix, the behavior of the inner column vectors is extremely regular; in fact, the same column vector is repeated, offset vertically by two rows from its neighbor, with these repeated vectors framed by the irregular end conditions. This simple structure makes the creation of filters for higher levels of resolution relatively easy. Furthermore, the fact that both sets of matrices are banded diagonal matrices means that the number of multiplications, although still linear in the number of rows (i.e.: the number of refined control points), is bounded by the largest number of nonzero entries in the repeated column vectors, making refinement a linear time operation [Pres92].

Although we may combine the  $P$  and  $Q$  filters in block matrix form and invert this to create the analysis filters  $A$  and  $B$ , the resulting matrix will most likely not be a banded

diagonal matrix. Thus, coarsening operations would take quadratic time to compute while refinement operations took linear time. A way to get around this problem is to recast this problem as an instance of a solution of  $Ax = b$ , where we are solving for  $x$ . This is done by taking the  $LU$  decomposition of the combined synthesis matrix  $PQ$ . We will solve for the set of control points  $\mathbf{c}^{j-1}$  knowing the matrix  $PQ$  and the input  $\mathbf{c}^j$ :

$$PQ\mathbf{c}^{j-1} = LU\mathbf{c}^{j-1} = L(U\mathbf{c}^{j-1}) = \mathbf{c}^j$$

This is done in two passes: first, solve for  $Ly = \mathbf{c}^j$  with backsubstitution; next, solve for  $U\mathbf{c}^{j-1} = \mathbf{y}$  using the same method. The  $LU$  decomposition of the matrix  $PQ$  will maintain the efficiencies of the original's banded diagonal form so this operation also can be performed in time linear to the size of the data set.

### 3.2 Integral and Fractional levels of Resolution

Once the filter bank has been created, a multiresolution curve representation is capable of supporting a number of graphical editing operations, including the ability to apply continuous levels of smoothing to a curve; the ability to edit a curve at any continuous level of detail; the ability to change the “sweep” or direction of a curve while maintaining its texture or “character” or, conversely, the ability to modify the “character” of a curve without affecting its overall “sweep”.

The coarsening or refinement of a curve to integral levels of resolution becomes trivial once the filter banks for those levels have been created. Suppose we are given a curve  $q(t)$  which has  $m$  control points  $\mathbf{c} = [c_0 \ \cdots \ c_{m-1}]$  and we wish to construct the approximating curve using  $m'$  control points  $\mathbf{c}' = [c'_0 \ \cdots \ c'_{m'-1}]$ , where  $m' < m$ , assuming both curves are endpoint-interpolating B-splines. If we assume for the moment that



$m = 2^j + 3$  and  $m' = 2^{j'} + 3$  for nonnegative integers  $j' < j$ , then the control points  $\mathbf{c}'$  of the approximating curve are given by

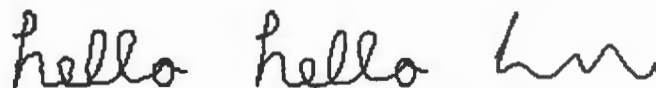
$$\mathbf{c}' = A^{j'+1} A^{j'+2} \dots A^j \mathbf{c}^j$$

That is, we run the multiresolution decomposition algorithm, recursively passing each coarsened set of control points through the filter bank until the desired resolution is reached. Again, since the computations performed at each level are done using the linear time  $LU$  decomposition algorithm previously described, this modification of the curve can be performed at interactive speeds. Note that, in practice, we would also be presenting each coarsened set of control points to the corresponding  $B$  filters as well, preserving the detail information lost through coarsening at each pass in a set of detail coefficients  $\mathbf{d}^{j-1}, \dots, \mathbf{d}^{j'}$ .

This process is straightforward when the desired levels of resolution are discrete in nature. If a fractional level of resolution is desired, there is no obvious way in which a quick approximation can be constructed. Instead, a more practical solution is to define a fractional-level curve  $q^{j+\mu}(t)$  for some value  $0 \leq \mu \leq 1$  which is a linear interpolation between its two nearest integer-level curves  $q^j(t)$  and  $q^{j+1}(t)$ , expressed by the following:

$$\begin{aligned} q^{j+\mu}(t) &= (1 - \mu)q^j(t) + \mu q^{j+1}(t) \\ &= (1 - \mu)\Phi^j(t)\mathbf{c}^j + \mu\Phi^{j+1}(t)\mathbf{c}^{j+1} \end{aligned}$$

This interpolation allows smoothing to take place at any continuous level. An example of such a fractional-level curve is shown in Figure 15.



**Figure 15:** Fractional-level curves: (a) level 8.0; (b) level 5.4; (c) level 3.1. [Fink94]



At any point in our application of coarsening and refinement operations on a multiresolution curve, the curve representation may contain a sequence of low-resolution control points  $\mathbf{c}^0, \dots, \mathbf{c}^{J-1}$  and a set of high-resolution detail parts  $\mathbf{d}^0, \dots, \mathbf{d}^{J-1}$ . This permits two very different kinds of editing to be performed on such a curve. If we edit some low-resolution version of the curve  $\mathbf{c}^j$  and then add back the detail values  $\mathbf{d}^j, \mathbf{d}^{j+1}, \dots, \mathbf{d}^{J-1}$ , we will have changed the overall sweep of the curve while preserving its details (see Figure 16). Conversely, if we leave the low-resolution control points intact and modify instead the set of detail values, we will have altered the character of the curve while preserving its overall sweep (see Figure 17). We next describe these editing operations in more detail.



Figure 16: Changing the sweep of a curve without affecting its character. [Fink94]

Editing a curve at an integral level of resolution is simple. Let  $\mathbf{c}^j$  be the control points of the original curve  $q^j(t)$ , let  $\mathbf{c}^j$  be a low-resolution version of  $\mathbf{c}^j$ , and let  $\hat{\mathbf{c}}^j$  be an edited version of  $\mathbf{c}^j$ , given by  $\hat{\mathbf{c}}^j = \mathbf{c}^j + \Delta\mathbf{c}^j$ . The edited version of the highest-resolution curve  $\hat{\mathbf{c}}^J = \mathbf{c}^J + \Delta\mathbf{c}^J$  can be computed and reconstructed as follows:

$$\begin{aligned}\hat{\mathbf{c}}^J &= \mathbf{c}^J + \Delta\mathbf{c}^J \\ &= \mathbf{c}^J + P^J P^{J-1} \dots P^{j+1} \Delta\mathbf{c}^j\end{aligned}$$

The lower the value for  $J$ , the greater the change to the overall sweep of the curve.

Editing a fractional-level curve is somewhat more complicated. Since a fractional-level curve is an interpolation between curves at neighboring integral levels of resolution based on some value for  $\mu$ , we would like the effect of any edits we perform on a curve of level  $j + \mu$  to interpolate the proportional changes to the curves at levels  $j$  and  $j + 1$ . That is, as  $\mu$  moves from 0 to 1, the curve at level  $j$  is less affected by an edit and the curve at level  $j + 1$  is more affected. Let  $q^{j+\mu}(t)$  be a fractional-level curve and let  $\mathbf{c}^{j+\mu}$  be the set of control points associated with this curve, that is

$$q^{j+\mu}(t) = \Phi^{j+1}(t)\mathbf{c}^{j+\mu}$$

This formulation suggests that the number of control points in  $\mathbf{c}^{j+\mu}$  matches the size of  $\mathbf{c}^{j+1}$  which is intuitively correct; in practice, these same control points are used to edit the curve.

Suppose the user modifies one of the control points  $c_i^{j+\mu}$ . To propagate the effect of this change, the system will have to move some of the nearby control points when  $c_i^{j+\mu}$  is modified. The distance these nearby points are moved is inversely proportional to  $\mu$ ; for example, when  $\mu$  is near 0, the control points at level  $j + \mu$  are subject to a wider propagation of the edit whereas when  $\mu$  is near 1, the displacement to nearby points is more confined.

Let  $\Delta\mathbf{c}^{j+\mu}$  be a vector recording the change to the control points of the fractional-level curve; this is essentially a zero vector except for the  $i$ -th entry which records the edit to  $c_i^{j+\mu}$ .

We will break this vector into two components: a vector  $\Delta\mathbf{c}^j$  recording the changes to the control points of the nearest integral lower-resolution curve and a vector  $\Delta\mathbf{d}^j$  recording the changes to the wavelet coefficients of the same lower-resolution curve and defined as

$\Delta\mathbf{d}^j = B^{j+1}\Delta\mathbf{c}^{j+1}$ . The changes to the higher-resolution curve at level  $J$  are then

reconstructed as follows:

$$\Delta \mathbf{c}^j = P^j P^{j-1} \dots P^{j+2} (P^{j+1} \Delta \mathbf{c}^j + Q^{j+1} \Delta \mathbf{d}^j)$$

We can obtain an expression for  $\Delta \mathbf{c}^{j+\mu}$  by the following derivation:

$$\begin{aligned} \Phi^{j+1}(t) \Delta \mathbf{c}^{j+\mu} &= q^{j+\mu}(t) = (1 - \mu) \Phi^j(t) \Delta \mathbf{c}^j + \mu \Phi^{j+1}(t) \Delta \mathbf{c}^{j+1} \\ &= (1 - \mu) \Phi^{j+1} P^{j+1} \Delta \mathbf{c}^j + \mu \Phi^{j+1}(t) \Delta \mathbf{c}^{j+1} \end{aligned}$$

And so

$$\begin{aligned} \Delta \mathbf{c}^{j+\mu} &= (1 - \mu) P^{j+1} \Delta \mathbf{c}^j + \mu \Delta \mathbf{c}^{j+1} \\ &= (1 - \mu) P^{j+1} \Delta \mathbf{c}^j + \mu (P^{j+1} \Delta \mathbf{c}^j + Q^{j+1} \Delta \mathbf{d}^j) \\ &= P^{j+1} \Delta \mathbf{c}^j + \mu Q^{j+1} \Delta \mathbf{d}^j \end{aligned}$$

Next, we need to define a new vector  $\Delta \mathbf{c}^j$  which records the changes to the control points at level  $j$  necessary to move the modified point  $c_i^{j+\mu}$  to its new position. We also define the vector  $\Delta \mathbf{c}^{j+\mu}$  to record the user's change to the  $i$ -th control point of the curve at level  $j + \mu$ , that is an otherwise zero vector whose  $i$ -th entry is  $\Delta c_i^{j+\mu}$ . The propagation of the effect of the edit is then determined by interpolating between these two vectors, using some interpolation function  $g(\mu)$ :

$$\Delta \mathbf{c}^{j+\mu} = (1 - g(\mu)) P^{j+1} \Delta \mathbf{c}^j + g(\mu) \Delta \mathbf{c}^{j+\mu}$$

So,  $\Delta \mathbf{c}^{j+\mu}$  will move the selected point to its new position and will also propagate the proportional effect of this change to its neighboring control points as a function of  $\mu$ . If we equate the right-hand sides of both versions of the equation and multiply the results by either  $A^{j+1}$  or  $B^{j+1}$ , we get the following:

$$\begin{aligned} A^{j+1} P^{j+1} \Delta \mathbf{c}^j + \mu A^{j+1} Q^{j+1} \Delta \mathbf{d}^j &= (1 - g(\mu)) A^{j+1} P^{j+1} \Delta \mathbf{c}^j + g(\mu) A^{j+1} \Delta \mathbf{c}^{j+\mu} \\ B^{j+1} P^{j+1} \Delta \mathbf{c}^j + \mu B^{j+1} Q^{j+1} \Delta \mathbf{d}^j &= (1 - g(\mu)) B^{j+1} P^{j+1} \Delta \mathbf{c}^j + g(\mu) B^{j+1} \Delta \mathbf{c}^{j+\mu} \end{aligned}$$

If we apply to these expressions the "invertibility" identities we listed in section 2.2, we get the following simplified expressions:

$$\Delta \mathbf{c}^j = (1 - g(\mu))\Delta \mathbf{c}^{j+1} + g(\mu)A^{j+1}\Delta \mathbf{c}^{j+\mu}$$

$$\Delta \mathbf{d}^j = \frac{g(\mu)}{\mu} B^{j+1} \Delta \mathbf{c}^{j+\mu}$$

In practice, any function on  $\mu$  that increases monotonically on  $[0,1]$ , such as  $\mu^2$ , would be suitable for  $g$ . The last detail to be defined is the definition of the vector  $\Delta \mathbf{c}^{j+1}$ . This also will be a vector which is zero everywhere except for one or two entries, depending on the index  $i$  of the modified control point and the  $i$ -th row of the refinement matrix  $P^{j+1}$ . We wish to determine the column index  $k$  of this matrix that identifies the point of maximal influence in the  $i$ -th row of that matrix. If one such point exerts the maximal influence, this means that the modified control point is most influenced by the control point  $c_k^{j+1}$  and we can define  $\Delta \mathbf{c}_k^{j+1}$  as  $\Delta c_i^{j+\mu} / P_{i,k}^{j+1}$ . If two neighboring points exert equal influence, then define both  $\Delta \mathbf{c}_k^{j+1}$  and  $\Delta \mathbf{c}_{k+1}^{j+1}$  as  $\Delta c_i^{j+\mu} / 2P_{i,k}^{j+1}$ .

The reverse of the previous sequence of derivations then boils down into the following sequence of steps for modifying a fractional-level curve:

1. Define  $\Delta \mathbf{c}^{j+\mu} = [0, \dots, 0, \Delta c_i^{j+\mu}, 0, \dots, 0]^T$
2. Define  $\Delta \mathbf{c}^{j+1}$  as described above.
3. Define  $\Delta \mathbf{c}^j$  and  $\Delta \mathbf{d}^j$  according to the equations at the top of this page.
4. Construct the offsets to the highest-resolution curve using the equation at the top of page 37.

The only portion of the above algorithm involving any repetitive computation is step 4 and, since this sequence of steps takes linear time, the entire manipulation should be able to be performed at interactive speeds [Stol96].



We may now discuss the other form of editing operations we can perform on multiresolution curves, namely those involving the detail coefficients. The goal of this process is to perform edits similar to those shown in Figure 17 where the texture or character of a curve may be modified while its general sweep is left unchanged.

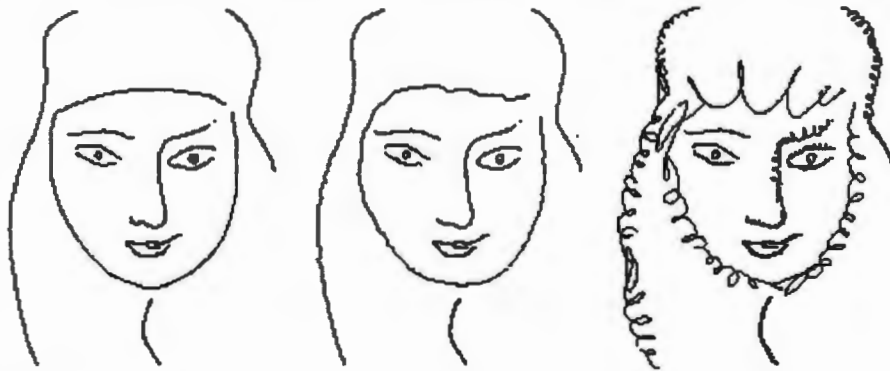


Figure 17: Changing the character of a curve without affecting its sweep. [Fink94]

The “editing” operation in this circumstance is almost trivial. Let  $c^j$  be a curve representation and let  $c^0, \dots, c^{j-1}, d^0, \dots, d^{j-1}$  denote its multiresolution decomposition. To edit the character of a curve at resolution level  $j$ , one simply replaces the detail coefficients  $d^j, \dots, d^{j-1}$  with some new set  $\hat{d}^j, \dots, \hat{d}^{j-1}$  and then reconstructs the curve back to level  $J$ . Finkelstein and Salesin [Fink94] discuss the possibility of preserving a repertoire of such textures in the form of a library of detail coefficients. The actual replacement of some subset of detail coefficients must be performed at some integer level, of course, but the resulting curve will still be subject to all of the other fractional-level manipulations described so far.

There are different ways in which we can process these detail coefficients. The high-level description of these editing operations as we have described them so far ultimately rests on a



representation of a parametric curve that we can treat in terms of its separate functions on  $x$  and  $y$ . This is indeed how we process and modify the control points of the curve and how we would practically render the curve in some graphic context. This may not be the best way to handle the detail coefficients of the curve representation. For example, Figure 18 shows how the detail values, if they are also computed and reconstructed using an  $x,y$  orientation, may result in some non-intuitive and undesirable behavior in the reapplication of that texture. An alternative to this is to specify a change in the curve relative to the tangent and normal directions of the lower-resolution curve  $q^{j-1}(t)$ . These tangent and normal values are computed using the parameter value  $t_0$  corresponding to the peak of the wavelet  $\psi_i^j(t)$ . The implication is that the curve representation is no longer a simple linear combination of the control points and the detail coefficients; instead, a change of coordinates must be applied, both when the details are computed and again when they are reapplied. Since this process is linear in the number of control points, however, this should add no degradation to the overall performance of the algorithm.

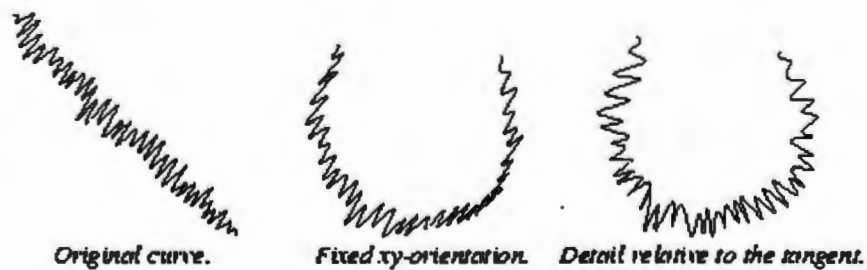


Figure 18: Orientation of detail in an edited curve. [Fink94]

### 3.3 Implications of Multiresolution Curve Theory

We have provided in this section an extremely high-level, mathematical blueprint for a number of very interesting and sophisticated graphical editing operations. Most interesting of all is the claim, which is supported by the consistency of the derivations based upon the core performance of the multiresolution decomposition algorithm, that all of these complex operations may be performed in time linear in the size of the representation, i.e. in real time. These arguments seem persuasive and we may prefer to be convinced by them or at least to use these arguments as the foundation for a more rigorous proof of correctness that would substantiate this claim. However, the most convincing validation (if not proof) of this claim would come from actually building an application that could model these operations in a thorough and convincing manner and to see if, in practice, these theoretical benefits can actually occur.

The remainder of this paper will describe the design and implementation of a GUI-based application designed for the construction and manipulation of multiresolution curve representations and modeling all of the operations described in this section.

#### 4. IMPLEMENTATION DESIGN AND RESULTS

In this section we will describe the design and implementation of an interactive, GUI-based application we shall christen as “CurvEditor”. The purpose of the CurvEditor is to provide an environment for the construction, editing, and refinement and coarsening of multiresolution curves, i.e.: two-dimensional curves whose representations are capable of embedding a range of detail about the curve at a variety of different scales of resolution. Beyond the implementation of this set of operations, as described in section 3 of this paper, the application also models a set of reasonable standard interface operations, such as the ability to store and retrieve curve representations as files; the ability to edit multiple curves; the ability to undo/redo edits and other operations; the ability to modify the display of editorial detail in the graphic environment; and other “good policy” operations.

We shall first of all present a high-level overview of the modular construction of the design, explaining the roles of these separate components. Next, we will discuss the implementation of each of these modules, focusing on the more important functionality in each module and how it serves to model the operations described in section 3. Lastly, we will have some remarks on the performance of the application, both in terms of interactive performance and also correctness, some description of the limitations of the application and unimplemented features, and some remarks on extensions and future work.

This application was written in the high-level programming language Java using the implementation of the language provided in the JDK (Java Developer’s Kit) v. 1.1.7b combined with the Java Foundation Classes (Swing) Libraries. Both of these software development tools are publically available and may be downloaded from the website <http://www.javasoft.com/> which is sponsored by Sun Microsystems, the originators of the Java language specification. This application was also written and tested on a Pentium

266MHz IBM-compatible PC with 32MB of RAM; all statements regarding any subjective evaluation of the interactive performance of this application should be understood in this context.

#### 4.1 Overall Structure of Application

A fundamentally sound strategy for organizing GUI (graphic user interface) based applications of any scale is to first decouple what is known as the “business logic”, or the component(s) where all of the primary computation involved in the application takes place, from the “UI logic”, the detailed and often highly complex code that organizes the visual information displayed to the user and relates the requests for services that the user makes through this interface to the proper routines in the business logic. This strategy was pursued from the outset in the design of this application and, as a result, the organization of this application is built upon two key components: first, the *multiresolution engine* which performs the business logic of the application; and secondly, the *application interface* which is itself organized into subcomponents, as we shall see.

The multiresolution engine is the portion of the code dedicated to creating the filter bank, i.e.: the system of analysis and synthesis filters which enable shifts in resolution, and also to supplying the basic operations *coarsen* and *refine*, through which the resolution shifts occur. These operations take as inputs a set of control points, the basis of the curve representation, and an integer for the resolution level which also functions as an index into the arrays of matrices forming the filter bank. Both operations return the transformed set of control points as a return value. The other major operation in this module is *editCurve* which is invoked when an edit on a fractional-level curve is performed; this method returns the modified set of control points recording the distributed effect of a fractional-level edit, as described in section 3.2.



The design of the application interface borrows an organizational idea from graphics and Java programmer Leen Ammeraal [Amme98]. The interface itself is split into two major components: the application frame, which contains the main method for initiating the program and contains the typical repertoire of GUI components, such as a menu bar, a toolbar and a slider bar, and all of the requisite interfacing logic; and the application canvas, the field within which the curves are actually drawn and manipulated. Structurally, the application frame contains the canvas object and the canvas contains, among its member data, an instantiation of the multiresolution engine. The canvas object also contains an instance of an undo stack, a modified version of the Java Stack object, and a Vector containing possibly multiple instantiations of CurveState objects which is the representation this application uses to store information defining a curve.

The complete code listing for this application can be found in Appendix C.

## **4.2 Description of Major Components**

### **4.2.1 *Multiresolution Engine***

The component serving as the multiresolution engine in this application is defined in the file `Multires.java` as the `Multires` class. The fundamental data comprising a `Multires` object are the arrays for storing the matrices or two-dimensional arrays that represent the analysis and synthesis filters of the filter bank. The primary purpose of this module is to create a filter bank designed explicitly to operate on sequences of control points that represent endpoint-interpolating cubic B-spline curves and to provide coarsen and refine methods which employ this filter bank.

The creation of the filter bank is helped greatly by the repetitive structure of these matrices above a certain level of resolution (see section 3.1 and Appendix A). While the



lower resolution filters are more irregular, the construction of the higher-level matrices may be automated to a great extent. In the `Multires` object, this automated matrix generation is done by the `initMatrices` method, which is called from the constructor.

The `initMatrices` method first of all inserts the matrices for the first three levels of resolution into the arrays `PQ` and `AB`. These matrices have already been hardcoded as statically declared arrays in the class definition. The reason for declaring them as static is an attempt at optimization; a static declaration forces this data to reside as part of the class definition rather than within an instance of a class so this may employ one less level of indirection in terms of data access. The naming of the arrays as `PQ` and `AB` refers to the fact that these matrices are modeling the synthesis and analysis filters as block matrices, with both the `P` and the `Q` matrices, for example, combined in the same matrix. The sources for the `P` and `Q` matrices are the filters shown in Appendix A. The corresponding `AB` matrices are the inverses of the `PQ` matrices. The hardcoded values in these matrices were computed using MATLAB. These matrices are small enough not to warrant any extra optimization regarding matrix multiplications.

Above this level, the matrices are built in a much more optimized manner. To understand what is happening we need to look at the structure of the combined `PQ` matrix at higher levels. First of all, the `P` and `Q` matrices at level  $j$  have, respectively,  $(2^j + 3)$  rows by  $(2^{j-1} + 3)$  columns and  $(2^j + 3)$  rows by  $2^{j-1}$  columns. Figure 19 shows the structure, though not the actual values, of a block matrix containing the `P` and `Q` matrices at level 4. As in the matrices shown in Appendix A, all zero entries are left as blanks. The repeated column vectors are shown in bold. It should be clear that, as the level  $j$  increases, the dimensions of this block matrix double with every increase. If we leave the structure of this

matrix unmodified, even with the number of zeros present in the matrix any computations using this form would take *quadratic* rather than linear time, which is not acceptable.

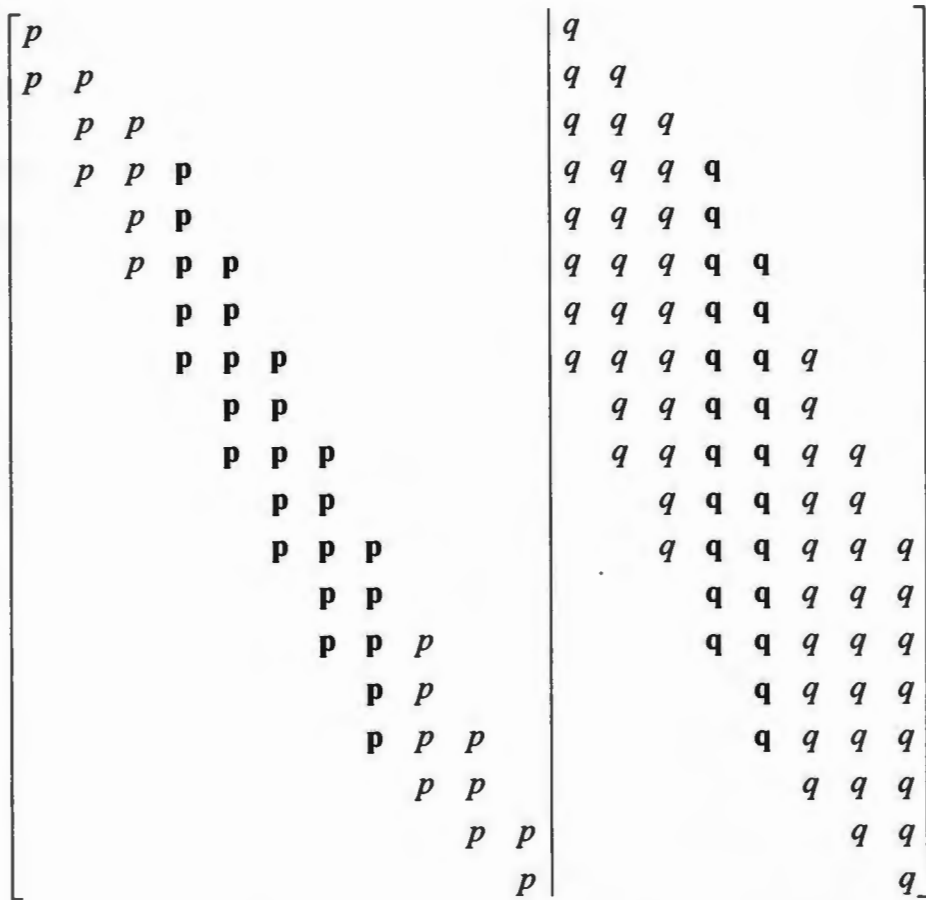


Figure 19: Block matrix diagram of synthesis filters P and Q.

The optimization strategy comes in two parts. The first step is to observe that, since both P and Q in isolation are structured as banded diagonal matrices (recall that we constructed the Q matrices to support explicitly this kind of structure), we may interleave the column vectors of this block matrix to produce a single banded diagonal matrix. The difference in the number of columns is that P has 3 more column vectors than Q. Therefore, such an interleaved matrix will have the first 2 columns and the last 2 columns from P and an alternation of columns from P and Q between these ends. The pattern of the resulting



used is to essentially declare storage for a matrix with the same number of rows but with the number of columns equal to the sum of the number of subdiagonals (diagonal "rows" below the main diagonal) plus the number of superdiagonals (those above the main diagonal) plus 1 (the main diagonal). Next, the column vectors are entered into this matrix in anti-diagonal fashion so that the column that is indexed at a position equal to the subdiagonal width plus one contains all of the entries of the main diagonal. Performing this redistribution on our example gives us the final compacted form in Figure 21.

$$\begin{bmatrix}
 & & & & P & & & & q \\
 & & & & p & P & q & & q \\
 & & & & p & Q & p & q & q \\
 & & & p & q & P & q & p & q & q \\
 & & & q & p & Q & p & q & q \\
 & & q & p & q & P & q & p & q & q \\
 q & & q & p & q & P & q & p & q & q \\
 & q & q & p & Q & p & q & q \\
 q & & q & p & q & P & q & p & q & q \\
 & q & q & p & Q & p & q & q \\
 q & & q & p & q & P & q & p & q & q \\
 & q & q & p & Q & p & q & q \\
 q & & q & p & q & P & q & p & q \\
 & q & q & p & Q & p & q & q \\
 q & & q & p & q & P & q & p \\
 & q & q & p & Q & p \\
 & & q & q & P & p \\
 & & & q & P & p
 \end{bmatrix}$$

Figure 21: Optimized banded diagonal form for block matrix PQ.

This matrix is still somewhat sparse, as evidenced by all the (implicitly) zero entries, but the number of such entries per row is now fewer. More to the point, as the level  $j$  increases and

the number of rows approximately doubles for each increase, the width of the matrices generated will *always be the same*. This is due to the repetitive column vector in the  $Q$  matrices, whose length of nonzero elements decides the width of the transformed block matrix. Thus, the execution time for matrix multiplication is reduced to the number of entries in the column vector to be multiplied times a constant and therefore becomes linear time. The `initMatrices` method literally constructs these compact form matrices by populating the respective anti-diagonals of the matrix with the values contained in the repeated column vectors, which are declared as static one-dimensional arrays. The column vectors belonging to the  $Q$  matrix must first be multiplied by a scalar normalization value which is resolution-level dependent (see Appendix A). The optimized routine `banmul`, that takes a matrix in this form and a column vector as arguments and returns the computed column vector recording their product, is drawn almost directly from the code listing presented in [Pres92].

(A remark about the code contained in [Pres92] is in order. As inestimable as this book is in its practical utility, it has the rather unfortunate habit of demonstrating array processing routines, which are clearly written in C, by indexing into an array of size  $n$  with values in the range  $1..n$ , whereas the syntax normally expected in this language, and in Java as well, handles index values from 0 to  $n - 1$  as legal. No explanation for this strange usage is offered. To use this code, then, one can do one of two things. One can either declare extra storage, i.e.: declare an array of size  $n + 1$  where one of size  $n$  is needed, forgetting about the entry at index 0 and using the remaining range of legal indices. Or, one can rewrite the code so that proper indices are used, which is a highly non-trivial task given the optimizations inherent in the code. In this application, the latter course was taken. So, as an additional public service, the code listing presented in Appendix C contains highly optimized code for



manipulating and performing operations on banded diagonal matrices which indexes the arrays forming these matrices in a legal and proper manner.)

Having manufactured the PQ matrices for several levels of resolution in this manner, `initMatrices` must also create the analysis filters or the AB matrices as well. This time, however, we cannot simply invert the corresponding PQ matrix since the result will not be a banded diagonal matrix and so we will have no opportunity for optimization. Instead, we must decompose the PQ matrix into upper triangular and lower triangular matrices and use these to find a solution for a given set of control points using backsubstitution, as described in section 3.1.

We can describe the LU decomposition algorithm with a simple example. Suppose we are given a square matrix  $A$  and a column vector  $b$  and we wish to find the column vector  $x$  such that  $Ax = b$ . For clarity, let us work with an actual example and let

$$A = \begin{bmatrix} 1 & 3 & 2 \\ -2 & -6 & 1 \\ 2 & 5 & 7 \end{bmatrix}$$

We first of all declare another matrix  $L$  which is initialized as the identity matrix with the same dimensions as  $A$ . We next begin reducing matrix  $A$  to form  $U$ . For example, to reduce the second row, we multiply the first row by 2 and add this to the second row, since the entry in the pivot of the first row is 1. At the same, we place this multiplier, 2, in the first column of the second row, yielding the following two matrices for  $U$  and  $L$  at this step:

$$U = \begin{bmatrix} 1 & 3 & 2 \\ 0 & 0 & 5 \\ 2 & 5 & 7 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We perform the same reduction on the third row at this time as well, this time multiplying the first row by  $-2$  and adding the result to the third row and setting the element in the first column of the third row of  $L$  to  $-2$ :

$$U = \begin{bmatrix} 1 & 3 & 2 \\ 0 & 0 & 5 \\ 0 & -1 & 3 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$$

Note that the next logical step is to interchange the second and third rows in  $U$  in order to have a nonzero pivot value on the second row. We must save this permutation information and so we will maintain an array named `indx` whose  $i$ -th entry is the index of the row that got swapped into this  $i$ -th row. In this example, the values of the array `indx` would read `[0, 2, 2]` since the first row was not swapped but the third row was swapped with the second row. The last row is, of course, left in place so the index of that row, `2`, occupies the last entry of the array. After swapping we have the following upper and lower triangular matrices:

$$U = \begin{bmatrix} 1 & 3 & 2 \\ 0 & -1 & 3 \\ 0 & 0 & 5 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$$

Note that, if we perform the multiplication  $LU$  and reverse the swap of the second and third rows, we get the original matrix  $A$ . Having computed these matrices, we can recast the equation  $Ax = b$  as  $LUx = b$  or  $L(Ux) = b$ . We then find the solution by first of all solving for  $Ly = b$  and then for  $Ux = y$ , using backsubstitution in both cases.

When  $A$  is a banded diagonal matrix and is also of the compact form described above, we can perform an  $LU$  decomposition on this matrix using the routine `bandec` which has also been adapted from [Pres92]. This routine changes the original input matrix  $A$ , just as in our example above, so in practice, when `initMatrices` calls this routine to decompose a PQ

matrix, it first makes a copy of this matrix and passes this copy. This copy will contain the upper triangular component as a result and the lower triangular matrix  $AL$  is contained in another matrix which has the same number of rows but a number of columns limited by the subdiagonal width of  $A$ . The width of the upper triangular part is also, effectively, the superdiagonal width of  $A$ . Combined backsubstitution through both of these matrices, then, in order to compute the effects of the analysis filter, will also take time linear in the size of the data since the number of computations per row of both these matrices is bounded by a constant. The computed decomposed matrices are stored in the two-dimensional array  $AB$ , just as the hard-coded lower-resolution matrices are. In addition, the array  $AB_{indx}$  maintains a record of any row permutations undergone by the upper triangular matrix in the process of computation.

The major utility methods of the `Multires` object are `coarsen`, `refine` and `editCurve`. The `coarsen` method takes as inputs an array of `Point2D` objects and an integer corresponding to the resolution level of the desired analysis filter. First, the `Point2D` class, as defined in the file `Point2D.java`, is a lightweight object for maintaining an  $x,y$  coordinate, with these elements stored as floats. The input points in this case represent the set of control points for a curve with resolution value  $j$ . The `coarsen` method first converts the points to dual one-dimensional arrays of  $x$  and  $y$  coordinates and then applies the analysis filters corresponding to resolution level  $j$  to each of these arrays. In both arrays, the results should be a shortened list of control point data followed by a list of detail coefficients making up the difference of the length. What `coarsen` must do is decide what to do based upon the input level  $j$ . If  $j$  is low enough, one of the low-resolution hard-coded analysis filters is used and so the results are obtained by executing a simple matrix multiplication. If  $j > 3$ , then the multiplication result must be obtained through backsubstitution through the stored

decomposed upper and lower triangular matrices for that level. This computation is performed by calling the `banbks` method, again adapted from [Pres92], which backsubstitutes through both matrices corresponding to that level, using the array of input  $x$  coordinates, then the array of  $y$  coordinates. After this computation is made, we must change the order of the elements in the array; recall that to build the banded diagonal matrix in the first place we had to interleave the order of the column vectors. The output of the backsubstitution evinces this same interleaving and so, as a result, we must un-interleave both of these  $x$  and  $y$  arrays first before we then convert them back into an array of `Point2D` objects and return this array.

The work involved in the `refine` method is very similar. The input set of `Point2D` objects in this case contains a list of control points for a low-resolution curve followed by a list of detail coefficients representing any significant behavior, if any, that when applied to this curve will show up at the next level. Once again, after converting the points to arrays of  $x$  and  $y$  coordinates, a decision is made based upon the value of  $j$  for the multiplication procedure to be used. For low values, again, a simple multiplication using the hard-coded synthesis matrix is performed. For  $j > 3$ , we use the banded diagonal forms which means that the input sets of  $x$  and  $y$  coordinates must be interleaved in the same manner as the column vectors that went into the building of this matrix. Once done, a call is made for each of the  $x$  and  $y$  arrays to the method `banmul`, again adapted from [Pres92], which returns the result of the matrix multiplication. Note that no un-interleaving is necessary here; the column vectors were interchanged but the rows were left intact. These  $x$  and  $y$  arrays are converted back to `Point2D` objects and the resulting array is returned.

Note that in the current implementation, the only valid values for  $j$  are those between 0 and 8. This means that the highest resolution curve that can be computed at present is one



with 256 segments, which was thought sufficient for testing. The `initMatrices` routine fabricates the filter bank to accommodate only this range now. The plan for extending this portion of the implementation involves modifying the `coarsen` and `refine` routines to handle the special case where the resolution level exceeds `LEVEL_MAX`. In such a case, both routines would check the vectors `PQmore` and `ABmore` to see if they had any entries up to the desired level and, if not, fabricate the filter bank matrices for this and all intervening levels that had not yet been created. It is unknown what the execution time would be for this step but it would enjoy the virtue of only needing to be done once since those portions of the filter bank would be available for later operations. A short delay in this instance might be acceptable.

The `editCurve` method is used when a modification is made to a control point of a curve at a fractional-level of resolution. This method represents the implementation of the algorithm described in section 3.2. The inputs to this method are the set of control points for the high-resolution curve bracketing this curve (recall that a fractional-level curve is represented by interpolating between two integer level curves), the floor or the next integer resolution down from the current level, the value for the fractional amount,  $\mu$ , and the *delta-x* and *delta-y* and the array index of the moved point. The input array of points is first converted to arrays of  $x$  and  $y$  coordinates and these arrays are then coarsened to the next lower level of resolution, using similar logic to the `coarsen` method. The control point portion of this new array is then multiplied by  $g(\mu)$ , here  $\mu^2$ , and the detail coefficient portion by  $g(\mu)/\mu$ . Next, we compute the small vector `deltaCprime` which will store the entries of the synthesis matrix of the next level that maximally affect the control point at the current index. The function that computes this vector basically exploits the regular structure of the higher scale synthesis matrices, making decisions on a case by case basis depending on



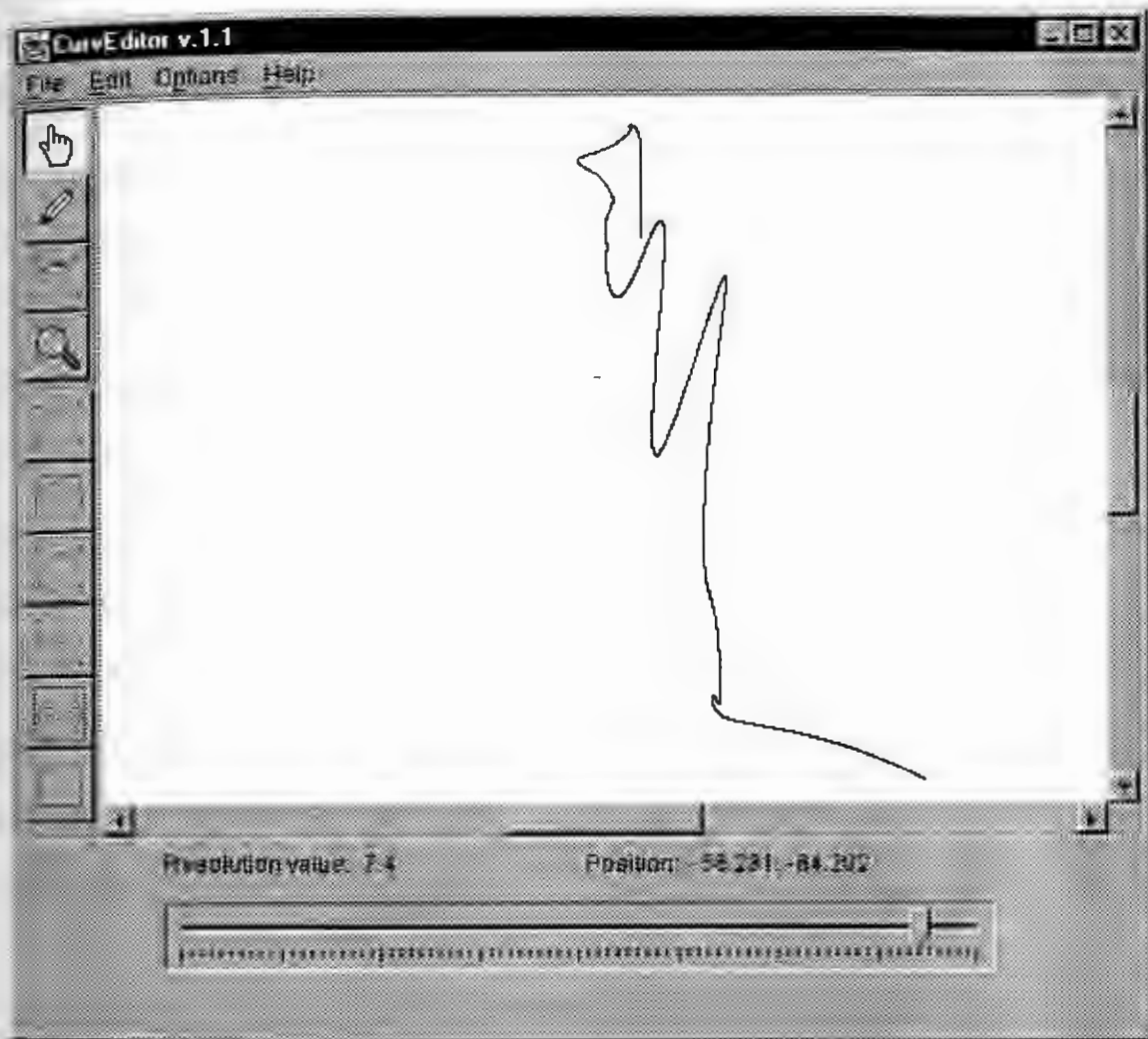
whether one entry has the maximum value in the row corresponding to the index of the control point or whether two entries do. The column corresponding to that entry is returned by this function and this value marks the index in the lower-resolution curve where the computed `deltaCprime` vector is applied. Lastly, after these adjusted delta values are computed they are applied to the input set of control points and this list is then returned.

#### 4.22 *The Application Frame*

The interface for the `CurvEditor` is composed of two major components: the application frame which is represented by the `CurvEditor` object defined in the file `CurvEditor.java`; and the application canvas represented by the `CvBspline` object defined in the file `CvBspline.java`. Both of these components derive a good deal of their functionality from the use of the Java Foundation Classes or Swing libraries, a system of classes that constitutes a wholesale revision of the graphics functionality that had formerly been offered in the Java programming environment. Much of the success of this application is owed to the performance of these objects and we shall make specific citations of particular components where appropriate.

The application frame is modeled by the `CurvEditor` class which extends the Swing class `JFrame`. (Many of the improved graphics classes defined in the Swing libraries employ familiar class identifiers prefaced with the letter 'J'.) This class contains a main method and is in fact the class that initiates the entire application. As a `JFrame`, the `CurvEditor` object is capable of instantiating an independent window (complete with title bar, exit buttons, etc.). The primary purpose of the `CurvEditor` frame is to deploy a system of GUI-based tools through which the user can invoke a number of useful application features. This system includes elements such as a menu bar (`JMenuBar`), a toolbar containing a number of push buttons with icons (`JToolBar`), and a slider bar used to control the resolution of a candidate

curve (JSlider). An image showing the overall appearance of the application is shown in Figure 22.



**Figure 22: The CurvEditor interface.**

The structure of the CurvEditor class is centered entirely on all of the objects comprising the interface. All of the functionality in the class is tied either to the creation and initialization of a major interface component, such as the menu bar or the toolbar, or it is dedicated to managing a specific action tying together the action (and appearance) of an interface component with a member function of the application canvas object. Because of

the tight fit between the interface and the class design, a description of the features offered by the interface may serve as the best description of the class itself.

We shall begin with the toolbar, which is the vertical element displaying the buttons to the left of the interface in Figure 22. There are ten buttons on the toolbar grouped into two sections, with four buttons in the top group and six in the bottom. The topmost group implements a ButtonGroup membership which means that only one button in the group may be selected at any one time, like a radio button. The buttons in this group select the current editing mode and, from top to bottom, the editing options are Edit or Select a curve, Sketch (Scribble) a curve, Build (Construct) a curve, and Zoom in on a certain area. At publication time, neither the Sketch feature nor the Zoom selection were fully implemented behaviors. The buttons in the bottom group are more independent and control various aspects of the environment displayed in the application canvas. From top to bottom these buttons toggle between showing and hiding the control polygon of a curve, showing and hiding the control points of a curve, showing and hiding the knots at the endpoints of the curve segments, showing and hiding a grid, showing and hiding coordinate values in tracking the mouse position, and lastly one for clearing the screen. Each of these buttons has a corresponding swapping method in the CurvEditor class which updates the appearance of the button and dispatches the current selected setting to the application canvas object so that these decisions are echoed there.

Next, we shall discuss the menubar, which contains four menu categories: File, Edit, Options and Help. The pulldown menus associated with each of these entries are shown in Figure 23.

The File menu displays, first of all, options for New, Open and Save and Save As. All of these invoke file dialog boxes or JFileChooser objects for displaying the selected contents of a



particular target directory. An object that implements a FileFilter interface may be invoked to associate icons with specific file types in the pictorial representation of the directory contents offered in the dialog box. A selection of a file for opening will result in the loading of the object contained in that file into the graphic environment, with the dialog box event handler calling the canvas object's loadCurve method. Conversely, a save file option will invoke another event handler that will retrieve the object to be saved from the canvas's saveCurve method. Note that what is actually being loaded and/or saved in this situation are actual objects since the basic curve objects in our system implement the Java Serializable interface and can therefore be exported, as objects with system state, to any stream, including a file.

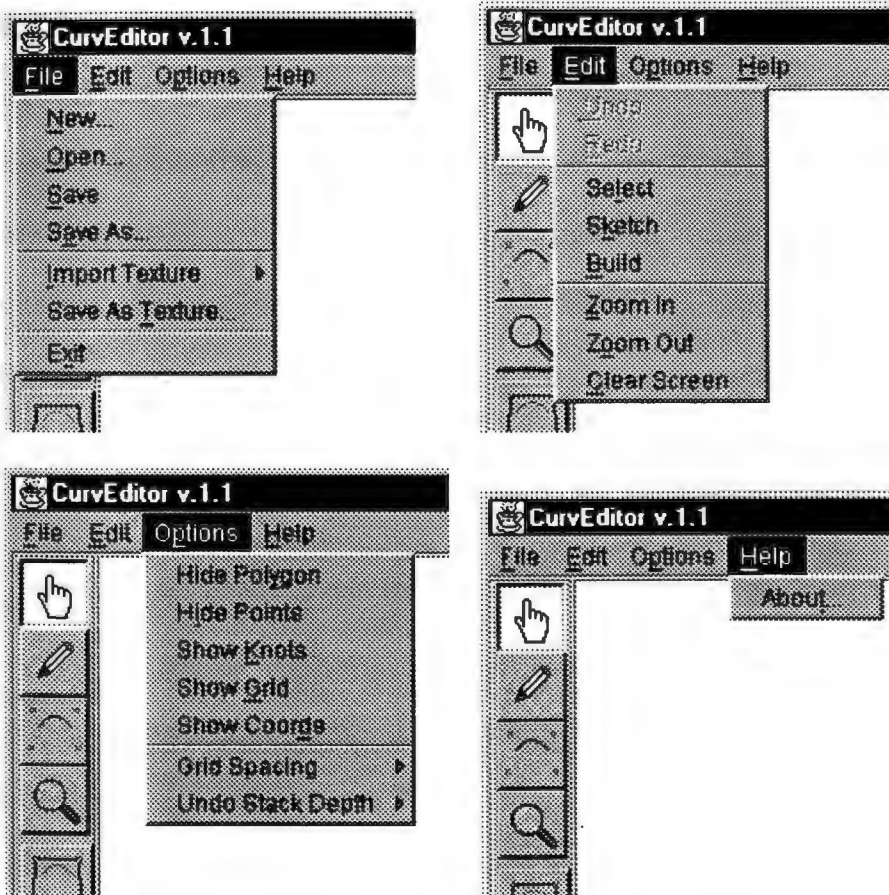


Figure 23: CurvEditor menu selection options.

The remaining options offered under the File menu have to do with manipulating the texture of an existing curve. Import Texture brings up a pictorial list of three generic textures and an optional menu for loading a user-defined texture into the environment. Once again, an object is read in from a file, this time using the member `loadTexture` method, and this object is passed along to the canvas's `importTexture` method which will then revise whatever curve has been selected with the new set of details. The option Save As Texture also invokes a dialog box, this one prompting the user to save a curve as a texture rather than as a full-blown curve representation, which involves a different file format.

Under the Edit menu we first of all have the options for Undo and Redo. These selections invoke the obvious operations on the canvas object's undo stack which holds the most recent moves up to a limit that the user selects under the Options menu. In principle, anytime a curve is augmented with the Build curve selection or is edited by moving one of its control points, these qualify as moves to be saved to the undo stack. The remaining entries under the Edit menu, Select, Sketch, Build, Zoom In/Out and Clear Screen, all repeat the functionality already indicated for the corresponding buttons on the toolbar.

The Options menu also offers backup selection capability, this time echoing the toggle functionality of all of the Show/Hide buttons in the toolbar. In addition to these selections we have two additional submenus, one to select the grid size, which offers the user the choices of 10, 15 or 25 coordinate points for the grid dimension; the other to set the undo limit, i.e. the depth or the number of moves the undo stack will "remember", with options for 1, 5 and 10 moves.

The Help menu simply has an About option which brings up a dialog box identifying the authorship of the application.



The other major component of the interface is the prominent slider bar located at the bottom of the window. As may be inferred, this slider bar controls the shifts in resolution for a selected curve. The current value of the slider is displayed in a JLabel object with the caption "Resolution value: ". The actual value obtained from a slider is an integer so this object is calibrated with 80 ticks and the return value is simply divided by 10 and displayed this way. (Recall that the current implementation of the multiresolution engine constructs its filter banks with a maximum level value of 8.) This value is also transmitted in this way to the member canvas object; in fact, a reference to the slider itself is passed directly to the canvas object when it is instantiated.

The canvas object once declared is passed as an argument to the constructor of a JScrollPane object set within the application frame. Doing this makes the canvas a scrollable client of the scroll pane, allowing the canvas to be able to display curves that may be larger than the field afforded by the frame.

#### 4.23 *The Application Canvas*

The canvas component of the interface is a member object of the application frame class and it contains among its own member data an instantiation of a Multires object. The canvas class in this application is the CvBspline class which extends the JLabel class and is implemented in the file CvBspline.java. The canvas object maintains two classes of member data. First, the system state, meaning the characteristics of all of the curves being drawn in the canvas at any one moment, is preserved in the Vector object state. This Vector stores CurveState objects, defined in the file CurveState.java, which contain the set of control points (arrays of Point2D objects) needed to define a single curve as well as some additional information such as the curve's current resolution value (an integer) and its fractional resolution value (a float). Also belonging to this first class of member data are the

two stacks for managing the Undo and Redo commands. Both of these are of type `UndoStack`, defined in the file `UndoStack.java`, which extends the basic capabilities of a `Stack` object by defining a stack depth limit  $d$  and overloading the push operation to make sure that only the last  $d$  entries into the stack are preserved. As the user makes moves (either through building a curve or editing one of its control points) the prior `CurveState` is saved to the undo stack by a call within one or another of the mouse event handlers, known as `MouseListener` methods. If the user selects Undo, the current `CurveState` (maintained as a state variable `C`) is pushed onto the Redo stack and the last move popped from the Undo stack and redrawn. If Redo is subsequently selected, the current state is pushed to the Undo stack, the prior state is popped from the Redo stack, and the last move is redrawn again. If, after undoing a move, the user makes a new move in the canvas, the Redo stack is erased. Lastly, there is a `Vector` called `Scribble` used for recording a sketched curve in the canvas environment, sketched in this case by means of mouse input. This `Vector` is used when the application is in `SKETCH` mode and the user wishes to hand sketch a curve representation. Once such a curve has been completed, i.e.: at the point that the mouse is released, the next step is to convert the resulting set of data points, which has been saved in the `Vector` by repeated polls to the current mouse position, to a set of control points, i.e.: a B-spline representation. At publication time, some options on how to execute this conversion step were being researched and so this conversion step is not presently implemented in the application.

This first class of member data concerns maintaining a history within a current editing session. The other class of member data concerns the specific instance of a curve currently being edited. This includes first of all the `CurveState` variable `c` which identifies one of the curves saved in the `state` vector as the one curve selected for editing at that moment. Other

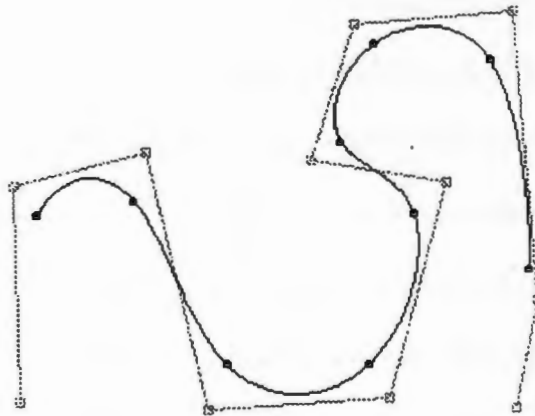
state variables in this class include those recording the center coordinates of the canvas and its height and width, necessary if the application window is resized; the variable `opState` which tracks the current editing mode (EDIT, SKETCH, DRAW or ZOOM); variables `stateIndex` for identifying the index in `State` of the current curve and `Index` for identifying the index of the control point in the current curve being edited; variables for recording the current `undoLimit` and `gridSpacing` values; and an entire range of boolean state variables for determining whether the control polygons, control points, knots or the grid will be drawn or not the next time the system is repainted. The last group of state variables each have their own member function, called by the appropriate event handler in the application frame to toggle their state. The variables `stateIndex` and `Index` actually get updated every time the method `paintComponent` is invoked since the most efficient way to plot whether the location of a mouse event is close enough to some specific visual feature, such as a control point for a specific curve, is to compare the locations of both of these as the feature is being redrawn.

A small third class of member data involves a set of helping objects, including a reference to the slider bar in the application frame, a `JLabel` for displaying the current coordinate values of the mouse cursor, and, of course, an instance of the multiresolution engine.

As mentioned, there are four editing modes in the canvas environment, only two of which, EDIT and DRAW, have been fully implemented. Much of the event handling logic is performed by the various `MouseListener` classes defined locally within the constructor. The primary methods of these classes, `mousePressed`, `mouseDragged`, `mouseReleased`, etc., perform some sophisticated checks and upgrades of the state variables of the canvas on a case basis, depending on the current editing mode.



When the system is in DRAW mode, the user can either build a new curve in the canvas window or add to a previously selected one, a selection having been made while in EDIT mode. To build a curve, the user simply clicks the mouse at a location, after which a new control point is drawn on the canvas. After four such control points are drawn, the first segment of the described curve appears. If the user has toggled the control polygon, the points and the knots to appear, then these will be drawn also. As the user adds additional clicks, additional control points appear and more segments are added to the curve (see Figure 24). To exit DRAW mode, the user must select a new editing option from the toolbar or the menubar.



**Figure 24: A sample multi-segment curve showing the control points, polygon and knots.**

When the system is in EDIT mode, the user can click on an empty region of the canvas to reset the `stateIndex` state variable, thereby allowing a new curve to be initiated. The user may also click on either the control points of a curve or the curve proper to select that curve for editing, also updating the `stateIndex` variable to the value corresponding to the index of this curve in the `state` vector. Once a curve is selected, which is denoted visually as the curve whose control points are filled squares versus outlined ones, the user may click

down on a control point and drag it to a new position, modifying as a result the representation of the curve. This of course can happen at both integral and fractional scales of resolution. At an integer level, the selected control point is itself the only one whose position is updated as a result of the user selection. At a fractional level of resolution, the proportional changes to neighboring control points must be calculated using the `editCurve` method of the member `Multires` object. This is done from both the `mouseDragged` and the `mouseReleased` mouse listener methods. The `CurveState` object maintains within its representation, at all times, two sets of control points: that of the high-resolution curve and that of the low-resolution curve. The reason for this is that it is more convenient to have both integer resolution curves handy when we need to render a fractional-level curve. (When an integer level curve is rendered, only the high-res curve representation is used.) To make use of the `editCurve` method, then, the low-resolution set of control points is passed as an argument, along with its resolution level, the current `mu` value, and the index of the control point and its *delta-x* and *delta-y* values. This method returns the revised set of control points after the effect of the delta has been propagated proportionally to the neighboring control points. Once obtained, this low-resolution curve is then refined to produce the corresponding high-resolution curve at the next level and this high-res curve then replaces the prior high-res curve in the current `CurveState`. Now, the newly edited curve can be rendered.

The other major activity that can occur only in EDIT mode is the shifting of the resolution of the currently selected curve. (In our implementation, it is not possible to coarsen or refine multiple curves since they may begin at different levels.) When the user adjusts the slider control in the application frame, the `ChangeListener` associated with that slider invokes the `Canvas` object method `shiftResolution`, passing to this method call the



float representing the new resolution value of the curve. Note that the only time any actual change to the sets of control points is necessary is either when we shift down from resolution  $j.1$  to  $j.0$  or shift up from  $j.0$  to  $j.1$  since, at these points, the integer-level curves within the current `CurveState` will both have to be upgraded one way or the other. When a shift down occurs, from level  $j.1$  to  $j.0$ , the curve goes from a fractional-level representation to an integral-level one. The resolutions of the low and high-res curves formerly bracketing the fractional-level were  $j$  and  $j + 1$ , respectively. We must now shift these to  $j - 1$  and  $j$ . This means first of all the low-res curve is copied to the high-res curve in the `CurveState`. Then, we coarsen a copy of this high-res curve to produce the corresponding curve at the next lowest level of resolution,  $j - 1$ , and make this our low-res curve in the `CurveState`. When the user increases the resolution, from  $j.0$  to  $j.1$ , we repeat roughly the same procedure in reverse. Since the bracketing curve resolutions now shift from  $j - 1$  and  $j$  to  $j$  and  $j + 1$ , respectively, the high-res curve now gets copied to the low-res curve and the high-res curve then undergoes refinement by calling the `Multires` object's `refine` method. All other shifts of resolution simply change the current value of the `mu` variable in the `CurveState`, which subsequently affects how the curve is rendered.

Every change to a state variable and every mouse event processed within the application canvas triggers a call to the canvas object's inherited `repaint` method which, in turn, calls the `paintComponent` method. This method goes through all of the curves in the `State Vector` and renders them, one by one, using a call to the overloaded `bspline` method. At the same time, as elements are being plotted and redrawn, a comparison with the last recorded mouse position is made since, when the system is in EDIT mode, it is by these comparisons that any changes to state variables such as `stateIndex` or `Index` are obtained.

If any state variables regarding the drawing of control points, polygons, knots or grids are set at this time, these elements are also drawn.

The `bspline` method has two forms: one takes as arguments a single set of control points and the number of points in the set; the other takes two sets of control points, the length of the higher resolution set, and the value  $mu$ . The first form is for drawing integer-level resolution curves and the second form is for fractional-level curves. If the value of  $mu$  in the second form of the method is 0, that method calls the integer version of the method.

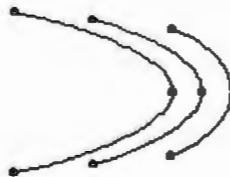
The integer-level resolution version of `bspline` is the more straightforward of the two. The rendering algorithm basically iterates through the set of control points, in successive overlapping groups of four, and computes the coefficient values  $c_k$  from each group of control points. After this, the  $x$  and  $y$  position of the next point in the curve is plotted using Horner's method:

$$((c_3 \times t + c_2) \times t + c_1) \times t + c_0$$

Each point in the curve, except the first, connects with its preceding neighbor by drawing a line between them. Thus, after all of the attempted precision and clever computation, the curves we finally generate are nothing more than polygonal lines after all!

The rendering of the fractional-level curve is somewhat trickier. Recall that in order to render a curve of fractional resolution we interpolate proportionally between the two adjacent integer level resolution curves on the basis of the value of  $mu$ . Since the refinement of a low-resolution curve segment evolves into two adjacent high-resolution curve segments, this means we are actually interpolating between one curve on the low-resolution end and a sequence of two curves on the high-resolution end. This means, first of all, we have to be somewhat careful in the manner in which we sample values for  $t$  on the interval  $[0,1]$ . For the integer-level renderings, sampling 50 such points works well in practice. However, if we

maintain this practice for interpolation purposes, we will have 50 points on the low end versus 100 points on the high end. Obviously, what we will need to do is to sample the first 25 points of the low end curve and every other point from a 50 point sample of the first high-end curve and interpolate the first 25 points, then repeat the process for the remaining half of the curve. In computing the coefficients that determine these curve segments, we must also notice that the low-end curve segment is built from four control points while the high-end pair of curve segments are built from five control points, three of which are shared by the two segments. The coefficients for each of the high-end segments, however, will be distinct. Thus, even though the high-end curve segments share control information, we must generate three overall distinct sets of four coefficients to describe each of the three segments as a basis for computing the interpolation. Once we have done all this, we may plot the respective  $x$  and  $y$  positions of the fractional-level curve by plotting the samples from the low-end curve and each segment of the high-end curve, multiplying the latter by  $mu$  and the former by  $1 - mu$ . Figure 25 shows an example of such a plot. Once a segment of the fractional-level curve has been rendered, we adjust the indices into both sets of control points so that we evaluate the next adjacent low-resolution curve segment and the next *pair* of high-resolution curve segments.



**Figure 25: Interpolation between a high-resolution curve (left) and a low-resolution curve (right).**

It should be mentioned that the Swing libraries confer their greatest benefit to this application right here at the rendering stage. Nearly all of the new graphical objects offered

by the Swing libraries, including the JLabel class, inherit from the JComponent class. One of the most useful methods in this class is one simply called `setDoubleBuffered` which takes a boolean argument. The effect of calling `setDoubleBuffered(true)` is to make any graphical updates operating on that component implement automatically a double-buffering strategy for redisplaying the updated visual image. Thus the programmer is freed completely from having to manage such a buffering policy himself. (It should also be mentioned that the Swing libraries offer no canvas object *per se*. The usual tactic, as performed in this application, is to use another JComponent, such as JLabel. Unfortunately, such an object resists maintaining any fixed size, say for example a height and width each three times those of the application frame, unless forced to by inserting an icon of that size. Thus, at start up, an icon composed of nothing but a huge field of white is inserted into the canvas object to give it its expanded dimensions. This step single-handedly accounts for a noticeable delay in the initialization of the application; fortunately, it only needs to be done once.)

The last functionality of interest in the CvBspline class is that for saving and importing textures. The format of a texture is simply the wavelet transform of the curve describing the texture, i.e.: a Vector containing a set of four control points determining the lowest-resolution form of that curve followed by a set of detail coefficients which record the desired texture. To save a texture is simply to coarsen the selected curve, extract the Vector containing the control points and detail information from the curve's CurveState, and return this Vector to the event handler in the application frame calling this method. To import a texture, a curve in the canvas environment must first be selected. This curve is then coarsened down to four control points and the detail coefficients from the imported Vector of texture points are then copied over into the selected curve. (If the size of the texture is greater, the size of the curve is increased accordingly.) Once done, the curve is then refined



back to its prior level of resolution. Lastly, coarsening this curve creates the companion low-resolution curve to this one in the selected CurveState. Note that, in these operations and in the `loadCurve` and `saveCurve` methods, which simply load and return the `State Vector`, respectively, only the `CvBspline` class has any dealings with `CurveState` objects. The I/O routines of the application frame deal only with the more generic `Vector` objects (which happen to contain `CurveState` objects or `Point2D` objects).

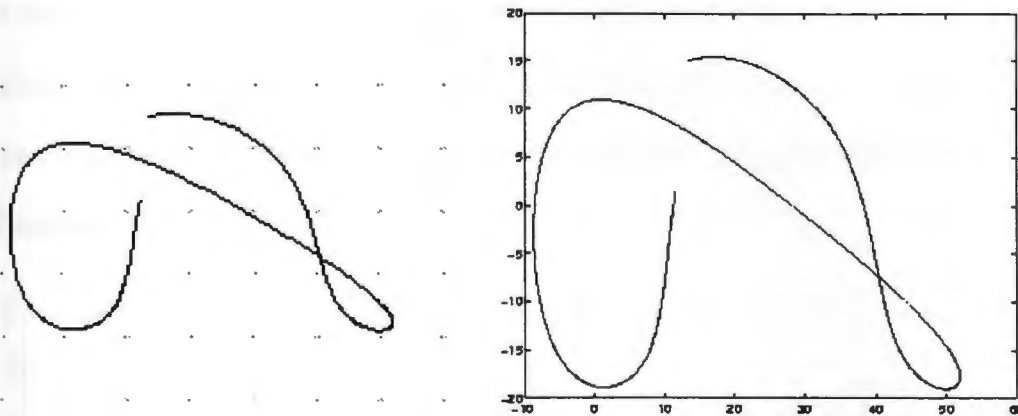
### 4.3 Evaluation and Results

The evaluation of the performance of an interactive application is somewhat difficult to objectify. For this particular application we are concerned with two predominant issues. One, the correctness of the rendering of B-spline curves at integral and fractional levels of resolution and, by implication, the correctness of the coarsening and refinement operations as applied to the representations of these curves. Two, the efficient performance of these coarsen and refine operations; are these operations capable of modeling shifts of resolution at interactive speeds?

As partial validation of the first issue, a number of the figures used in the illustration of this paper are screen captures of curves constructed using the `CurvEditor` application, which at least indicates that the curve rendering procedures actually function. A more precise diagnostic is supplied by the grid option of the display. We can, in practice, deploy a set of control points at explicit grid coordinates and compare the curve produced in the canvas environment with one similarly generated by a tool such as `MATLAB` and make a visual comparison of the results. A number of such sample curves produced discernably identical curves in both environments. Figure 26 demonstrates one such comparison. In terms of testing the coarsening and refinement operations, perhaps the best way to verify their accuracy is to construct a sample low-resolution curve, using a set of pre-determined grid



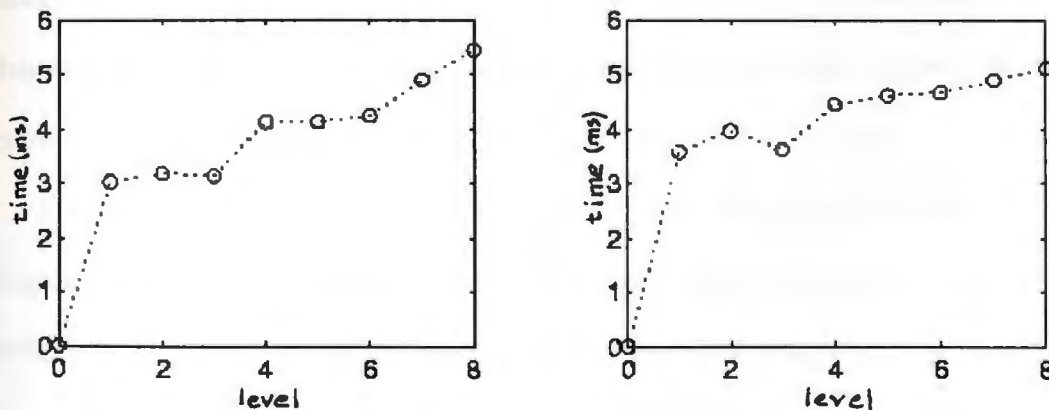
coordinates, refine this curve to the maximum setting, and then coarsen the curve back to its original low-resolution form. Aligning the mouse cursor with one of the resulting set of control points will give us the coordinates of that point shown in the "Position:" label of the interface. Any error introduced over the course of shifting the resolution of the curve will be evinced by a discrepancy between the new coordinate values of the point and the old. A number of such tests were conducted using several different curves and *no appreciable error* was found in any instance. The margin for error used was the width of the square visually identifying the control point which measures to about a value of 1.6 in the grid coordinate system.



**Figure 26: Sample comparison of curves produced by (a) CurvEditor and (b) MATLAB.**

The performance of the coarsen and refine operations was tested by taking a set of 1000 curves, with randomly instantiated control point information, and refining each of these curves to the maximum level of resolution, recording the execution time at each level and averaging the results over all 1000 curves. Next, the procedure was repeated, this time in the reverse direction timing the coarsening operation. A graph showing the plot of execution time vs. resolution level for both operations is shown in Figure 27, clearly demonstrating that

even for larger sets of control points, the amount of time spent in these respective operations is not a factor in the overall performance of the application. We had intended to generate another graph demonstrating the execution times of these operations plus that of the graphic redrawing to get an estimate on how much delay is attributable to the graphics operations versus the coarsening and refinement operations. Unfortunately, the implementation of the Java interpreter in the testing environment used in this research appears to consign graphic redrawing operations to a separate thread of execution. This makes it difficult to bracket the beginning and ending of a sequence of tasks, including a graphic redraw with timer checks, to determine its overall execution time. Anecdotally, the delay incurred from redrawing is much more noticeable at higher levels of resolution, due no doubt to the number of curve segments to plot and render. The inclusion of the grid steps this delay up even more so. However, the delay is not prohibitive and does not undermine the utility of basic editing operations such as moving a control point.



**Figure 27: Average execution time of (a) refine and (b) coarsen methods per resolution level (in milliseconds).**

We have already mentioned several components of the application that were not fully implemented at publication time, such as the procedure for converting a sketched curve to a

set of control points or the zoom feature. One other feature not implemented is the plotting of the change of orientation of the detail coefficients mentioned in section 3.2. The methodology for applying this change of coordinates calculation was still being researched at publication time and its implementation would constitute the first desirable revision of the present application. Another sub-performing feature, one belonging in this case to the Swing libraries, is the JSlider object. Like the graphics redrawing procedure, the event handler underlying the slider seems also to operate on a different thread of execution, making its manipulation somewhat difficult in practice. (The testing environment used for this application uses a touchpad instead of a mouse; the slider may conceivably perform better when a mouse is used.) We have circumvented this problem in this application by allowing the slider to be manipulated by the arrow keys on the keyboard.

Despite the aforementioned problems and deficiencies, the overall performance of this application is quite good and seems to provide the confirmation we sought regarding the ability of multiresolution curve representations to be modeled at interactive speeds. The fact that we have been able to confirm this using a language as notoriously compromised in its execution time as Java only serves to amplify the success of these concepts.

Avenues for future work on this material are plentiful. Since the publication of the original paper on multiresolution curves [Fink94], research on modeling multiresolution surfaces has already made significant headway, the most promising research being done in conjunction with Pixar Studios (see Stollnitz, et al. [Stol96]). The extension of the mathematics involved uses the well-researched construction of so-called Bezier and B-spline patches, which are the surface equivalents of curve segments [Bart87]. Among the preliminary conclusions of this research is that any surfaces topologically related to a two-dimensional mesh are candidates for multiresolution analysis [Stol96]. The present

application could conceivably be rewritten to model such surfaces using the Java 3D API which leverages a high degree of efficiency by mapping Java calls to DirectX or OpenGL libraries on the host system.

In its present form, the application is, by design, already extensible. Since the canvas is effectively decoupled from the application interface, it may already serve as a ready-made object capable of being plugged in to a larger, richer Java-based graphical editing application. Furthermore, since the basic data types in the system, the CurveState objects, are serializable, they may be exported to any data stream, which opens up the possibility for constructing a distributed, Java-based curve editing environment.



## 5. CONCLUSIONS

We have presented in this paper a unified representation that is capable of modeling a given two-dimensional curve at multiple scales of resolution. As outlined in the original research published by Finkelstein and Salesin [Fink94], this representation is based on the use of the wavelet transform. Such a curve representation consists of a set of control points defining a succession of piecewise cubic B-spline segments and a set of detail values marking or recording changes to, or events occurring on, this curve at higher levels of resolution. We have shown that the techniques for manipulating wavelet transforms used in multiresolution analysis are also applicable to this curve representation. Furthermore, when special care is taken to optimize the construction of the filter bank used to shift these representations between adjacent levels of resolution, we have shown that these operations may be performed in time linear with the size of the curve.

Besides operations for coarsening and refining the resolution of a curve, we have also described operations for rendering these curves at continuous levels of resolution, i.e.: at both integer and also at so-called fractional levels of resolution. This latter depiction is effected by performing a linear interpolation between neighboring integer level resolution curves. We have also defined operations for editing a curve's "sweep" and a curve's "character" independently by modifying the set of control points and the set of detail coefficients respectively within the curve representation. In the case of editing the "sweep", we have further defined operations for both integer and fractional level curves.

These operations form the basis of an interactive application for editing multiresolution curves. We have realized this application in the form of an independent object-oriented component, the Application Canvas, which is written in Java and may be embedded in any Java-based environment where these curve-editing operations are desirable. This object

makes use of a member object of its own, the multiresolution engine, to generate the filter bank and to provide the operations for coarsening and refining the set of control points forming the curve. Each set of control points is itself contained in another object, the CurveState, which is declared as Serializable so that it can be ported in its object state to any input/output stream, such as a file stream or a channel across a network. Lastly, we have placed this canvas object within a simple GUI-based application frame, forming a basic application for editing two-dimensional multiresolution curves. Through the use of this application, we have demonstrated that the above operations defined on multiresolution curves may indeed be performed in real time, i.e.: at interactive speeds, as was suggested in the original research [Fink94]. The result is an application component capable of modeling two-dimensional curves at multiple levels of resolution and available to operate in any environment supporting a Java Virtual Machine.

## REFERENCES

- [Amme98] Leen Ammeraal. *Computer Graphics for Java Programmers*. Chichester: John Wiley and Sons, 1998.
- [Ange97] Edward Angel. *Interactive Computer Graphics: a Top-Down Approach with OpenGL*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1997.
- [Bank90] M.J. Banks and E. Cohen. "Realtime spline curves from interactively sketched data." *Computer Graphics*. Vol. 24, no. 2: pp. 99-107, 1990.
- [Bart87] Richard H. Bartels, John C. Beatty, Brian A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Los Altos, California: Morgan Kaufman Publishers, Inc., 1987.
- [Chui92] Charles K. Chui. *An Introduction to Wavelets*. San Diego: Academic Press, 1992.
- [Fari88] Gerald Farin. *Curves and Surfaces for Computer-Aided Geometric Design*. Boston: Academic Press, 1988.
- [Fink94] Adam Finkelstein, David H. Salesin. "Multiresolution Curves". *Proceedings of the Special Interest Group on Computer Graphics (SIGGRAPH) 1994*, pp. 261-268. Association for Computing Machinery, New York, 1994.
- [Fole96] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. *Computer Graphics: Principles and Practice, Second Edition*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1996.
- [Fors88] David Forsey and Richard H. Bartels. "Hierarchical B-spline Refinement". *Computer Graphics*. Vol. 22, no. 4: pp. 205-212, 1988.
- [Hubb96] Barbara Burke Hubbard. *The World According to Wavelets*. Wellesley, Massachusetts: A. K. Peters, Ltd., 1996.
- [Kais94] Gerald Kaiser. *A Friendly Guide to Wavelets*. Boston, Massachusetts: Birkhauser, 1994.
- [Mall89] Stephane Mallat. "A Theory for Multiresolution Signal Decomposition: the Wavelet Representation." *Institute of Electrical and Electronic Engineers Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, no. 7: pp. 674-693. July, 1989.
- [Pres92] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge: Cambridge University Press, 1992.

[Stol96] Eric J. Stollnitz, Tony D. DeRose, David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. San Francisco: Morgan Kaufman Publishers, Inc., 1996.



## APPENDIX A: Endpoint-interpolating cubic B-spline matrices

Source: Eric J. Stollnitz, et al. *Wavelets and Computer Graphics: Theory and Applications*. San Francisco, California: Morgan Kaufmann Publishers, Inc., 1996. Pp. 214-216.

$$P^1 = \frac{1}{2} \begin{bmatrix} 2 & & & & & & \\ & 1 & 1 & & & & \\ & & 1 & 1 & & & \\ & & & 1 & 1 & & \\ & & & & 1 & 1 & \\ & & & & & 1 & 2 \end{bmatrix} \quad Q^1 = \sqrt{7} \begin{bmatrix} 1 \\ -2 \\ 3 \\ -2 \\ 1 \end{bmatrix}$$

$$P^2 = \frac{1}{16} \begin{bmatrix} 16 & & & & & & \\ & 8 & 8 & & & & \\ & & 12 & 4 & & & \\ & & & 3 & 10 & 3 & \\ & & & & 4 & 12 & \\ & & & & & 8 & 8 \\ & & & & & & 16 \end{bmatrix} \quad Q^2 = \sqrt{\frac{315}{31196288}} \begin{bmatrix} 1368 & & \\ -2064 & -240 & \\ 1793 & 691 & \\ -1053 & -1053 & \\ 691 & 1793 & \\ -240 & -2064 & \\ & & 1368 \end{bmatrix}$$

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

$$Q^3 = \begin{bmatrix} 6.311454 & & & & & & \\ -9.189342 & -1.543996 & & & & & \\ 7.334627 & 4.226722 & 0.087556 & & & & \\ -3.514553 & -5.585477 & -0.473604 & -0.000155 & & & \\ 1.271268 & 6.059557 & 1.903267 & 0.019190 & & & \\ -0.259914 & -4.367454 & -4.367454 & -0.259914 & & & \\ 0.019190 & 1.903267 & 6.059557 & 1.271268 & & & \\ -0.000155 & -0.473604 & -5.585477 & -3.514553 & & & \\ & 0.087556 & 4.226722 & 7.334627 & & & \\ & & -1.543996 & -9.189342 & & & \\ & & & 6.311454 & & & \end{bmatrix}$$

$$e^{i2\pi} = \sqrt{\frac{5 \cdot 2^i}{675221664}}$$

25931.200710		
-37755.271723	-6369.305453	
30135.003012	17429.266054	385.797044
-14439.869635	-23004.252368	-2086.545605 -1
5223.125428	24848.487871	8349.373420 124
-1067.879425	-17678.884301	-18743.473059 -1677 -1
78.842887	7394.685374	24291.795239 7904 124
-0.635830	-1561.868558	-18420.997597 -18482 -1677
	115.466347	7866.732009 24264 7904
	-0.931180	-1668.615872 -18482 -18482
		123.378671 7904 24264
		-0.994989 -1677 -18482 -1677 -0.994989
		124 7904 7904 123.378671
		-1 -1677 -18482 -1668.615872 -0.931180
		124 24264 7866.732009 115.466347
		-1 -18482 -18420.997597 -1561.868558 -0.635830
		7904 24291.795239 7394.685374 78.842887
		-1677 -18743.473059 -17678.884301 -1067.879425
		124 8349.373420 24848.487871 5223.125428
		-1 -2086.545605 -23004.252368 -14439.869635
		385.797044 17429.266054 30135.003012
		-6369.305453 -37755.271723
		25931.200710

## APPENDIX B: MATLAB code for B-spline wavelets.

Source: Eric J. Stollnitz, et al. *Wavelets and Computer Graphics: Theory and Applications*. San Francisco, California: Morgan Kaufmann Publishers, Inc., 1996. Pp. 217-222.

```
function P = FindP(d, j)
% P = FindP(d, j) returns the P matrix for B-spline scaling functions
% of degree d, level j.

d = fix(d);

if d < 0
    error('FindP: Must have d >= 0.');
```

```
end;

j = fix(j);

if j < 1
    error('FindP: Must have j >= 1.');
```

```
end;

if d==0
    P = [1; 1];
    for i = 2:j
        P = [P zeros(size(P)); zeros(size(P)) P];
    end;
else
    u = Knots(d, j-1);
    g = Greville(d,u);
    P = eye(2^(j - 1) + d);
    for k = 0:2^(j-1)-1
        [u, g, P] = InsKnot(d, u, g, P, (2*k+1)/2^j);
    end;
end;

return;

function x = Knots(d, j)

% x = Knots(d, j) returns a vector of knot values for B-spline scaling
% functions of degree d, level j.

x = [zeros(1, d-1) [0:2^j-1]/2^j ones(1, d)];
return;

function x = Greville(d, u)

% x = Greville(d, u) returns the vector of Greville abscissa values
% corresponding to degree d and knot vector u.
```

```

l = length(u);
x = u(1:l-d+1);

```

```

for k = 2:d
    x = x + u(k:l-d+k);
end;

```

```

x = x / d;
return;

```

```

function [uret, gret, pret] = InsKnot(d, u, g, p, unew)

```

```

% [uret, gret, pret] = InsKnot(d, u, g, p, unew) inserts a new knot
% at unew for B-spline scaling functions of degree d, thereby
% modifying knot vector u, Greville abscissas g, and synthesis matrix
% p.

```

```

uret = sort([u unew]);
gret = Greville(d, uret);
pret = PolyEval(g, p, gret);
return;

```

```

function pret = PolyEval(g, p, gnew)

```

```

% pret = PolyEval(g, p, gnew) returns the values of a control polygon
% defined by abscissas g and ordinates p, evaluated at gnew.

```

```

[m, n] = size(p);
if length(g) ~= m
    error('PolyEval: Length of g and rows of p must be the same.');
```

```

end;

for i = 1:length(gnew)
    row = max(find(g <= gnew(i)));
    if row == m
        pret(i,:) = p(m,:);
    else
        frac = (g(row+1) - gnew(i))/(g(row+1) - g(row));
        pret(i,:) = frac*p(row,:) + (1-frac)*p(row+1,:);
    end;
end;
return;

```

```

function I = Inner(d, j)

```

```

% I = Inner(d, j) returns the inner product matrix for B-spline
% scaling functions of degree d at level j.

```

```

IO = BernInnr(d);
n = 2^j + d;
I = zeros(n);
w = BernWts(d, j);

```



```

for k = 1:n
    w1 = reshape(w(:,k), d+1, 2^j);
    for l = k:n
        w2 = reshape(w(:,l), d+1, 2^j);
        I(k,l) = trace(w1'*IO*w2);
        I(l,k) = I(k,l);
    end;
end;

I = I / 2^j;
return;

function I = BernInnr(d)

% I = BernInnr(d) returns the matrix of inner products of
% Bernstein polynomials of degree d.

i = ones(d+1, 1)*[0:d];
j = i';
I = Choose(d, i).*Choose(d, j)./(Choose(2*d, i+j)*(2*d + 1));

return;

function c = Choose(n, r)

% c = Choose(n, r) returns (n choose r) = n! / (r! (n-r)!).

c = Fact(n)./(Fact(r).*Fact(n-r));
return;

function f = Fact(m)

% f = Fact(m) returns the matrix of factorials of entries of m.

[r,c] = size(m);
f = zeros(r,c);
for i = 1:r
    for j = 1:c
        f(i,j) = prod(2:m(i,j));
    end;
end;
return;

function w = BernWts(d, j)

% w = BernWts(d, j) returns a matrix of B-spline scaling
% function weights for Bernstein polynomials of degree d, level j.

w = eye(2^j + d);
if d == 0
    return

```

```

end;
u = Knots(d, j);
g = Greville(d, u);
for i = 1:2^j - 1
    for r = 1:d
        [u, g, w] = InsKnot(d, u, g, w, i/2^j);
    end;
end;
return;

```

```

function Q = FindQ(d, j, normalization)

```

```

% Q = FindQ(d, j, normalization) returns the Q matrix for B-spline
% scaling functions of degree d, level j. If normalization is 'min'
% (or is not specified) then the smallest entry in each column is made
% 1. If normalization is 'max' then the largest entry in each column
% is made 1. If normalization is 'L2' then the L^2 norm of each
% wavelet is made 1.

```

```

if nargin < 3
    normalization = 'min';
elseif ~strcmp(normalization, 'min') & ~strcmp(normalization, 'max')...
    & ~strcmp(normalization, 'L2')
    error('FindQ: normalization must be ''min'', ''max'', or ''L2''.');
end;

```

```

P = FindP(d, j);
I = Inner(d, j);
M = P'*I;
[m1, m2] = size(M);
n = m2 - rank(M);
Q = zeros(m2, n);
found = 0;
start_col = 0;

```

```

while (found < n/2) & (start_col < m2)
    start_col = start_col + 1 + (found > d);
    width = 0;
    rank_def = 0;
    while ~rank_def & (width < m2 - start_col + 1)
        width = width + 1;
        submatrix = M(:, start_col:start_col+width-1);
        rank_def = width - rank(submatrix);
    end;
    if rank_def
        % find nullspace of submatrix(should be just one column)
        q_col = null(submatrix);
        if strcmp(normalization, 'min')
            % normalize column so smallest nonzero entry has |1|
            q_col = q_col/min(abs(q_col) + 1e38*(abs(q_col)<1e-10));
        elseif strcmp(normalization, 'max')
            % normalize column so largest entry has magnitude 1
            q_col = q_col/max(abs(q_col));
        end;
    end;
end;

```

```

% change sign to give consistent orientation
q_col = q_col*(-1)^(start_col + floor((d+1)/2) + (q_col(1,1)>0));

% put column into left half of Q
found = found + 1;
Q(start_col:start_col + width-1, found) = q_col;

% use symmetry to put column into right half of Q in reverse
% order and negated if degree is even
Q(:, n-found+1) = flipud(Q(:, found))*(-1)^(d+1);
end;
end;

if strcmp(normalization, 'L2')
% normalize matrix so each column has L^2 norm of 1
ip = Q'*I*Q;
Q = Q*diag(1./sqrt(diag(ip)));
end;

return;

```

## APPENDIX C: CurvEditor code listing.

*Multires.java* Contains the code for the multiresolution engine. Creates the synthesis matrices  $P$  and  $Q$  for the first eight resolution levels. The matrices for the lowest three resolution levels are just hardcoded in place, as are the analysis filters  $A$  and  $B$  for these levels. Above level 3, the column vectors for the  $P$  and  $Q$  matrices are interleaved and the resulting matrices are reformed into more compact forms to support efficient banded-diagonal matrix multiplication operations. The corresponding analysis matrices at these levels are built by performing an  $LU$  decomposition on the  $PQ$  banded diagonal matrix, storing these resulting matrices and solving for the value of  $\mathbf{x}$  in the equation  $L(U(\mathbf{x}))=\mathbf{b}$  by backsubstitution.

Major methods: `coarsen`, `refine`, `editCurve`.

*CurvEditor.java*. Contains the application frame and the basic interfacing functionality for processing GUI-based events and requests and relaying these to its member `CvBspline` object, which processes all of the curve editing operations. Also contains functionality for handling file-based I/O of Vector objects representing curve and texture objects.

*CvBspline.java*. Contains all of the functionality for manipulating curve representations graphically. A number of methods are tied in directly with elements of the interface defined in the *CurvEditor* object. Major methods: the overloaded `shiftResolution` methods which make use of the `coarsen` and `refine` operations of its member *Multires* object; the overloaded `Bspline` drawing methods; and the various event handlers involved with curve editing operations.



*CurveState.java*. Contains the definition of the Cloneable and Serializable curve representation object used by the *CvB spline* object.

*Point2D.java*. Defines a lightweight data structure for modeling a two-dimensional coordinate that is also Cloneable and Serializable.

*UndoStack.java*. Augments the properties of a basic Stack object by setting a stack limit and keeping track only of the most recent additions to the stack up to that limit. Also used by the *CvB spline* object.

```

// File:    Multires.java
// Author:   Stephen Alberg
//
// This file contains the implementation of an object of type Multires.
// A Multires object performs two primary actions: coarsen and refine.
// The inputs to these operations are arrays of Point2D objects,
// representing the control points for a parametric cubic B-spline
// curve representation.
//
// The coarsen operation takes an input set of control points and
// performs a knot removal operation on the curve, removing half of
// the curve's segments. The returned array, of the same dimension,
// contains the reduced set of control points in the top half of the
// array and, in the bottom half, the difference coefficients obtained
// from the multi-resolution decomposition.
//
// The refine operation takes as inputs an array of control points and
// an array of difference coefficients and returns an enlarged array of
// control points doubling the number of curve segments featured in the
// input representation and incorporating any features conveyed by the
// input difference coefficients.

```

```
import java.util.*;
```

```

public class Multires
{ // Class constants
  static final float TINY = 1.0e-20f;
  static final int MAX = 1024;
  static final int LEVEL_MAX = 8;
  static final int SUB_DIAG_WIDTH = 5;
  static final int SUPER_DIAG_WIDTH = 5;
  static final int DIAG_WIDTH = SUB_DIAG_WIDTH + SUPER_DIAG_WIDTH + 1;
  static final int DOWN = 1;
  static final int UP = -1;

  // Small resolution synthesis and analysis matrices.
  // AB* matrices are the inverses of their corresponding PQ*
  // matrices.
  //
  // The contents of these matrices have been adapted from those
  // presented in Wavelets for Computer Graphics. Stollnitz, DeRose
  // and Salesin. Morgan Kaufmann, 1996. Specifically, these
  // matrices relate the sets of cubic B-spline scaling functions and
  // endpoint-interpolating cubic B-spline wavelets at neighboring,
  // integral levels of resolution.

  static final float[][] PQ1 = {
    { 1.0f, 0.0f, 0.0f, 0.0f, 2.645751f },
    { 0.5f, 0.5f, 0.0f, 0.0f, -5.291503f },
    { 0.0f, 0.5f, 0.5f, 0.0f, 7.937254f },
    { 0.0f, 0.0f, 0.5f, 0.5f, -5.291503f },
    { 0.0f, 0.0f, 0.0f, 1.0f, 2.645751f }};

  static final float[][] AB1 = {
    { 0.9375f, 0.1250f, -0.1250f, 0.1250f, -0.0625f },
    { -0.6875f, 1.3750f, 0.6250f, -0.6250f, 0.3125f },
    { 0.3125f, -0.6250f, 0.6250f, 1.3750f, -0.6875f },

```

```

    { -0.0625f,    0.1250f,   -0.1250f,    0.1250f,    0.9375f  },
    {  0.023623f, -0.047246f,  0.047246f, -0.047246f,  0.023623f  });

```

```

static final float[][] PQ2 = {
    { 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
      4.347003f, 0.0f  },
    { 0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
      -6.558636f, -0.762632f },
    { 0.0f, 0.75f, 0.25f, 0.0f, 0.0f,
      5.697497f, 2.195745f },
    { 0.0f, 0.1875f, 0.625f, 0.1875f, 0.0f,
      -3.346048f, -3.346048f },
    { 0.0f, 0.0f, 0.25f, 0.75f, 0.0f,
      2.195745f, 5.697497f },
    { 0.0f, 0.0f, 0.0f, 0.5f, 0.5f,
      -0.762632f, -6.558635f },
    { 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
      0.0f, 4.347003f  });

```

```

static final float[][] AB2 = {
    { 0.835156f, 0.329688f, -0.239583f, 0.079167f,
      0.041667f, -0.092188f, 0.046094f },
    { -0.353906f, 0.707812f, 0.947917f, -0.345833f,
      -0.083333f, 0.254688f, -0.127346f },
    { 0.290625f, -0.581250f, -0.015625f, 1.612500f,
      -0.015625f, -0.581250f, 0.290625f },
    { -0.127344f, 0.254688f, -0.083333f, -0.345833f,
      0.947917f, 0.707812f, -0.353906f },
    { 0.046094f, -0.092188f, 0.041667f, 0.079167f,
      -0.239583f, 0.329688f, 0.835156f },
    { 0.037921f, -0.075842f, 0.055115f, -0.018212f,
      -0.009585f, 0.021207f, -0.010604f },
    { -0.010604f, 0.021207f, -0.009585f, -0.018212f,
      0.055115f, -0.075842f, 0.037921f  });

```

```

static final float[][] PQ3 = {
    { 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
      6.311454f, 0.0f, 0.0f, 0.0f  },
    { 0.5f, 0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
      -9.189342f, -1.543996f, 0.0f, 0.0f  },
    { 0.0f, 0.75f, 0.25f, 0.0f, 0.0f, 0.0f, 0.0f,
      7.334627f, 4.226722f, 0.087556f, 0.0f  },
    { 0.0f, 0.1875f, 0.6875f, 0.125f, 0.0f, 0.0f, 0.0f,
      -3.514553f, -5.585477f, -0.473604f, -0.000155f },
    { 0.0f, 0.0f, 0.5f, 0.5f, 0.0f, 0.0f, 0.0f,
      1.271268f, 6.059557f, 1.903267f, 0.019190f },
    { 0.0f, 0.0f, 0.125f, 0.75f, 0.125f, 0.0f, 0.0f,
      -0.259914f, -4.367454f, -4.367454f, -0.259914f },
    { 0.0f, 0.0f, 0.0f, 0.5f, 0.5f, 0.0f, 0.0f,
      0.019190f, 1.903267f, 6.059557f, 1.271268f },
    { 0.0f, 0.0f, 0.0f, 0.125f, 0.6875f, 0.1875f, 0.0f,
      -0.000155f, -0.473604f, -5.585477f, -3.514553f },
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.25f, 0.75f, 0.0f,
      0.0f, 0.087556f, 4.226722f, 7.334627f },
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.5f, 0.5f,
      0.0f, 0.0f, -1.543996f, -9.189342f },
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,

```

```

0.0f,      0.0f,      0.0f,      6.311454f });

static final float[][] AB3 = {
{ 0.802757f, 0.394486f, -0.253324f, -0.038667f,
  0.202945f, -0.092461f, -0.071912f, 0.069304f,
  -0.000533f, -0.025190f, 0.012595f },
{ -0.271998f, 0.543996f, 0.983350f, -0.050721f,
  -0.483377f, 0.245774f, 0.169557f, -0.168642f,
  0.001764f, 0.060594f, -0.030297f },
{ 0.135699f, -0.271398f, -0.085737f, 1.066678f,
  0.705998f, -0.519247f, -0.266282f, 0.289940f,
  -0.005147f, -0.101007f, 0.050503f },
{ -0.067342f, 0.134684f, 0.014071f, -0.415441f,
  0.294364f, 1.079328f, 0.294364f, -0.415441f,
  0.014071f, 0.134684f, -0.067342f },
{ 0.050503f, -0.101007f, -0.005147f, 0.289940f,
  -0.266282f, -0.519247f, 0.705998f, 1.066678f,
  -0.085737f, -0.271398f, 0.135699f },
{ -0.030297f, 0.060594f, 0.001764f, -0.168642f,
  0.169557f, 0.245774f, -0.483377f, -0.050721f,
  0.983350f, 0.543996f, -0.271998f },
{ 0.012595f, -0.025190f, -0.000533f, 0.069304f,
  -0.071912f, -0.092461f, 0.202945f, -0.038667f,
  -0.253324f, 0.394486f, 0.802757f },
{ 0.031252f, -0.062503f, 0.040137f, 0.006126f,
  -0.032155f, 0.014650f, 0.011394f, -0.010981f,
  0.000084f, 0.003991f, -0.001996f },
{ -0.014120f, 0.028241f, -0.002475f, -0.065410f,
  0.100562f, -0.037543f, -0.036192f, 0.033185f,
  -0.000104f, -0.012288f, 0.006144f },
{ 0.006144f, -0.012288f, -0.000104f, 0.033185f,
  -0.036192f, -0.037543f, 0.100562f, -0.065410f,
  -0.002475f, 0.028241f, -0.014120f },
{ -0.001996f, 0.003991f, 0.000084f, -0.010981f,
  0.011394f, 0.014650f, -0.032155f, 0.006126f,
  0.040137f, -0.062503f, 0.031252f });

// Arrays for storing precomputed synthesis and analysis
// matrices up to level LEVEL_MAX. AB matrices are the
// LU decomposition matrices of the respective PQ matrices.
//
// Initialized by call to initMatrices().

float[][][] PQ = new float[LEVEL_MAX][][];
float[][][] AB = new float[LEVEL_MAX][2][][];
int[][] ABindx = new int[LEVEL_MAX][];

// Additional storage for matrices of level > LEVEL_MAX
Vector PQmore = new Vector();
Vector ABmore = new Vector();
Vector ABindxmore = new Vector();

// Repeating column vectors used in the creation of the synthesis
// matrices for resolution levels > 3.
static final float[] P1 = { 1.0f, 0.5f };
static final float[] P2 = { 0.0f, 0.5f, 0.75f, 0.1875f };

```



```

static final float[] P3 = { 0.0f, 0.0f, 0.25f, 0.6875f, 0.5f, 0.125f
};
static final float[] PC = { 0.125f, 0.5f, 0.75f, 0.5f, 0.125f };

// Wavelet space synthesis matrix columns must be multiplied
// by the factor sqrt( 5 * 2^j / 675221664 )
static final float[] Q1 =
{ 25931.200710f, -37755.271723f, 30135.003012f,
-14439.869635f, 5223.125428f, -1067.879425f,
78.842887f, -0.635830f };
static final float[] Q2 =
{ 0.0f, -6369.305453f, 17429.266054f,
-23004.252368f, 24848.487871f, -17678.884301f,
7394.685374f, -1561.868558f, 115.466347f,
-0.931180f };
static final float[] Q3 =
{ 0.0f, 385.797044f,
-2086.545605f, 8349.373420f, -18743.473059f,
24291.795239f, -18420.997597f, 7866.732009f,
-1668.615872f, 123.378671f, -0.994989f};
static final float[] QC = { -1.0f, 124.0f, -1677.0f, 7904.0f,
-18482.0f, 24264.0f, -18482.0f,
7904.0f, -1677.0f, 124.0f, -1.0f };

// Constructor:
Multires()
{ initMatrices();
}

// Initializes synthesis and analysis matrices for first
// LEVEL_MAX levels of resolution.
void initMatrices()
{ // The first three levels are the statically declared matrices
PQ[0] = PQ1; PQ[1] = PQ2; PQ[2] = PQ3;
AB[0][0] = AB1; AB[1][0] = AB2; AB[2][0] = AB3;

// Next, create the banded diagonal matrices for the remaining
// levels. This will be done by interleaving column vectors
// from the respective P and Q analysis matrices, using the
// static P# and Q# vectors declared above. For each such
// banded matrix created, we will also perform an LU
// decomposition on the matrix for use in the analysis operation
// (coarsen).

int htBase = 16; // start at output = 2^4 + 3 control points

for (int j = 3; j < LEVEL_MAX; ++j)
{ //
// first, create the banded diagonal matrix
//

int height = htBase + 3;
PQ[j] = new float[height][DIAG_WIDTH];
float[][] M = new float[height][DIAG_WIDTH];

// compute multiplier and create local copies of

```

```

// Q column vectors
float q = (float)(Math.sqrt((double)(5 * htBase) /
                           (double)(675221664) ));

float[] tQ1 = scalarMult(Q1, q);
float[] tQ2 = scalarMult(Q2, q);
float[] tQ3 = scalarMult(Q3, q);
float[] tQC = scalarMult(QC, q);

// first seven columns of matrix:
int row = 0, col = SUB_DIAG_WIDTH;

// use the static vectors to create the banded diagonal
// matrix one column at a time...
populateDiagonal(PQ[j], M, P1, row, col, DOWN);
populateDiagonal(PQ[j], M, P2, row, ++col, DOWN);
populateDiagonal(PQ[j], M, tQ1, row, ++col, DOWN);
populateDiagonal(PQ[j], M, P3, row, ++col, DOWN);
populateDiagonal(PQ[j], M, tQ2, row, ++col, DOWN);
populateDiagonal(PQ[j], M, PC, row+3, SUB_DIAG_WIDTH + 2,
                 DOWN);
populateDiagonal(PQ[j], M, tQ3, row+1, DIAG_WIDTH-1, DOWN);

// next, populate the middle column vectors:
for (row = 7; row < height - 7; row++)
{ if (row % 2 == 1)
    populateDiagonal(PQ[j], M, PC, row-2, SUB_DIAG_WIDTH+2,
                    DOWN);
  else
    populateDiagonal(PQ[j], M, tQC, row-5, DIAG_WIDTH-1,
                    DOWN);
}

// lastly, populate the last seven column vectors:
row = height-1; col = 0;
populateDiagonal(PQ[j], M, tQ3, row-1, col, UP);
populateDiagonal(PQ[j], M, PC, row-3, SUB_DIAG_WIDTH-2, UP);
populateDiagonal(PQ[j], M, tQ2, row, ++col, UP);
populateDiagonal(PQ[j], M, P3, row, ++col, UP);
populateDiagonal(PQ[j], M, tQ1, row, ++col, UP);
populateDiagonal(PQ[j], M, P2, row, ++col, UP);
populateDiagonal(PQ[j], M, P1, row, ++col, UP);

//
// next, create the LU decomposition of this matrix
//

AB[j][0] = new float[height][SUB_DIAG_WIDTH]; // lower
AB[j][1] = M; // upper
ABindx[j] = new int[height];

bandec(AB[j][1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
       AB[j][0], ABindx[j]);

htBase *= 2;
}
}

```

```

// Performs a simple scalar multiplication on an input vector.
float[] scalarMult(float[] v, float scalar)
{ float[] x = new float[v.length];
  for (int i = 0; i < v.length; ++i)
    x[i] = v[i] * scalar;
  return x;
}

// Performs a simple matrix multiplication on an input
// matrix and vector.
// MODIFIED (3/9/99): input x contains elements that are
// not multiplied by the matrix and need to be copied
// over into the solution vector.
float[] matrixMult(float[][] A, float[] x, int n)
{ int m = A.length; //, n = x.length;
  if (m <= 0 || A[0].length != n)
    return null;

  float[] b = new float[x.length]; //m);
  int i = 0;
  for (; i < m; ++i)
  { b[i] = 0.0f;
    for (int j = 0; j < n; ++j)
      b[i] += A[i][j] * x[j];
  }

  // copy over any remaining elements in the vector
  for (; i < x.length; ++i)
    b[i] = x[i];

  return b;
}

// Used by initMatrices to create synthesis matrices.
void populateDiagonal(float[][] A, float[][] M, float[] v, int row,
                     int col, int dir)
{ int n = v.length;
  for (int i = 0; i < n; ++i)
  { A[row][col] = M[row][col] = v[i];
    if (dir == DOWN)
    { row++; col--;
    }
    else
    { row--; col++;
    }
  }
}

// Reduces resolution of cubic B-spline curve representation.
//
// Input: set of control points  $c^j$  | <difference coeffs.  $d^j$ ,
//         $j+$ , ...>
// Output: set of control points  $c^{(j-1)}$  | difference coeffs.
//          $d^{(j-1)}$  | <difference coeffs.  $d^j$ ,  $j+$ , ...>
//
public synchronized Point2D[] coarsen(Point2D[] P, int j)

```

```

{ // convert points to arrays of floats
  float[][] xy = Point2DToXY(P);

  // for level j, index into stored matrices is j-1
  if (j <= 3)
  { // use simple matrix multiplication
    xy[0] = matrixMult(AB[j-1][0], xy[0], numPoints(j));
    xy[1] = matrixMult(AB[j-1][0], xy[1], numPoints(j));
  }
  else if (j <= LEVEL_MAX)
  { // use LU decomposition of banded diagonal matrices
    banbks(AB[j-1][1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
           AB[j-1][0], ABindx[j-1], xy[0]);
    banbks(AB[j-1][1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
           AB[j-1][0], ABindx[j-1], xy[1]);

    // uninterleave the output vectors
    int n = numPoints(j);
    xy[0] = unleave(xy[0], n);
    xy[1] = unleave(xy[1], n);
  }
  else
  { // check if matrices up to and including this level
    // have been constructed...
    if (j - LEVEL_MAX < ABmore.size())
    { // build new matrices up to level j...
    }
  }
}

return XYToPoint2D(xy);
}

// Increases resolution of cubic B-spline curve representation.
//
// Input:  set of control points  $c^{(j-1)}$  | difference coeffs.
//          $d^{(j-1)}$  | <difference coeffs.  $d^j, j+, \dots$ >
// Output: set of control points  $c^j$  | <difference coeffs.  $d^j,$ 
//          $j+, \dots$ >
//
public synchronized Point2D[] refine(Point2D[] P, int j)
{ // convert points to arrays of floats
  float[][] xy = Point2DToXY(P);

  // for level j, index into stored matrices is j-1
  if (j <= 3)
  { // use simple matrix multiplication
    xy[0] = matrixMult(PQ[j-1], xy[0], numPoints(j));
    xy[1] = matrixMult(PQ[j-1], xy[1], numPoints(j));
  }
  else if (j <= LEVEL_MAX)
  { // use banded diagonal multiplication
    int n = numPoints(j);
    xy[0] = interleave(xy[0], n);
    xy[1] = interleave(xy[1], n);

    xy[0] = banmul(PQ[j-1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
                  xy[0]);

```



```

        xy[1] = banmul(PQ[j-1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
                      xy[1]);
    }
    else
    { // check if matrices up to and including this level
      // have been constructed...
      if (j - LEVEL_MAX < PQmore.size())
      { // build new matrices up to level j...
        }
      }
    }

    return XYToPoint2D(xy);
}

// Computes changes to set of control points at level j after
// an edit is applied to the point index at level (j + mu).
//
// P is the set of control points at level j,
// j is the floor of the current level of resolution,
// mu is a number on the interval (0,1].
// deltaX, deltaY are the edit changes to control point index
// at level (j + mu),
// index is the index of the edited control point at level (j+mu).
//
public Point2D[] editCurve(Point2D[] P, int j, float mu,
                          float deltaX, float deltaY, int index)
{ float g = mu * mu; // monotonically increasing function on mu
  // for gradational propagation of edits to
  // control points.

  if(mu > 0) j--;

  int n = numPoints(j), nPlus = numPoints(j+1);

  float[][] xy = new float[2][nPlus]; // creates array of zeros

  xy[0][index] = deltaX; // put deltas in array
  xy[1][index] = deltaY;

  if(j < 3)
  {
    xy[0] = matrixMult(AB[j][0], xy[0], nPlus);
    xy[1] = matrixMult(AB[j][0], xy[1], nPlus);
  }
  else if(j < LEVEL_MAX)
  { banbks(AB[j][1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
          AB[j][0], ABindx[j], xy[0]);
    banbks(AB[j][1], SUB_DIAG_WIDTH, SUPER_DIAG_WIDTH,
          AB[j][0], ABindx[j], xy[1]);

    xy[0] = unleave(xy[0], nPlus);
    xy[1] = unleave(xy[1], nPlus);
  }
  // else j is larger: implementation must wait until larger
  // matrices are created and stored

  // multiply control point deltas at level j by g and difference

```

```

// coefficient deltas at level j by g/mu
int i = 0;
while(i < n)
{ xy[0][i] *= g;
  xy[1][i] *= g;
  i++;
}
while(i < nPlus)
{ xy[0][i] *= mu; // (mu * mu)/ mu
  xy[1][i] *= mu;
  i++;
}

// apply deltaC_prime to k-th index of control points
float[][] deltaCprime = new float[2][2];
int k = computeDeltaCprime(deltaCprime, j, index, deltaX,
                          deltaY);

// multiply by (1 - g)
for(i = 0; i < 2; ++i)
  for(int ii = 0; ii < 2; ++ii)
    deltaCprime[i][ii] *= (1.0 - g);

// ...and add to array of deltas at index k
xy[0][k] += deltaCprime[0][0];
if(k+1 < n)
  xy[0][k+1] += deltaCprime[0][1];

xy[1][k] += deltaCprime[1][0];
if(k+1 < n)
  xy[1][k+1] += deltaCprime[1][1];

// Lastly, apply these deltas to the array of control points
int limit = (P.length < xy[0].length) ? P.length : xy[0].length;

for(i = 0; i < nPlus; ++i) //limit; ++i)
{ P[i].x += xy[0][i];
  P[i].y += xy[1][i];
}

return P;
}

// Computes the changes to a subset of control points at level j
// as a result of an edit at a fractional resolution. Returns
// the index of the set of control points which is affected.
int computeDeltaCprime(float[][] dCprime, int j, int index,
                      float deltaX, float deltaY)
{ int k = -1; // the index to be returned
  int n = numPoints(j+1);

  if(j+1 == 1)
  { switch(index)
    {
      case 0: k = 0;
              dCprime[0][0] = deltaX; dCprime[0][1] = 0.0f;
              dCprime[1][0] = deltaY; dCprime[1][1] = 0.0f;
    }
  }
}

```

```

        break;

    case 1: k = 0;
           dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
           dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
           break;

    case 2: k = 1;
           dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
           dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
           break;

    case 3: k = 2;
           dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
           dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
           break;

    case 4: k = 3;
           dCprime[0][0] = deltaX; dCprime[0][1] = 0.0f;
           dCprime[1][0] = deltaY; dCprime[1][1] = 0.0f;
           break;
    };
}
else if(j+1 == 2)
{ switch(index)
  {
    case 0: k = 0;
           dCprime[0][0] = deltaX; dCprime[0][1] = 0.0f;
           dCprime[1][0] = deltaY; dCprime[1][1] = 0.0f;
           break;

    case 1: k = 0;
           dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
           dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
           break;

    case 2: k = 1;
           dCprime[0][0] = deltaX/0.75f; dCprime[0][1] = 0.0f;
           dCprime[1][0] = deltaY/0.75f; dCprime[1][1] = 0.0f;
           break;

    case 3: k = 2;
           dCprime[0][0] = deltaX/0.625f; dCprime[0][1] = 0.0f;
           dCprime[1][0] = deltaY/0.625f; dCprime[1][1] = 0.0f;
           break;

    case 4: k = 3;
           dCprime[0][0] = deltaX/0.75f; dCprime[0][1] = 0.0f;
           dCprime[1][0] = deltaY/0.75f; dCprime[1][1] = 0.0f;
           break;

    case 5: k = 3;
           dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
           dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
           break;

    case 6: k = 4;

```

```

        dCprime[0][0] = deltaX; dCprime[0][1] = 0.0f;
        dCprime[1][0] = deltaY; dCprime[1][1] = 0.0f;
        break;
    };
}
else if(j+1 >= 3)
{
    if(index < 4)
    {
        switch(index)
        {
            case 0: k = 0;
                    dCprime[0][0] = deltaX; dCprime[0][1] = 0.0f;
                    dCprime[1][0] = deltaY; dCprime[1][1] = 0.0f;
                    break;

            case 1: k = 0;
                    dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
                    dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
                    break;

            case 2: k = 1;
                    dCprime[0][0] = deltaX/0.75f; dCprime[0][1]= 0.0f;
                    dCprime[1][0] = deltaY/0.75f; dCprime[1][1]= 0.0f;
                    break;

            case 3: k = 2;
                    dCprime[0][0] = deltaX/0.6875f;
                    dCprime[0][1] = 0.0f;
                    dCprime[1][0] = deltaY/0.6875f;
                    dCprime[1][1] = 0.0f;
                    break;
        };
    }
    else if(index > n-5)
    {
        int m = numPoints(j);

        if(index == n-4)
        {
            k = m - 3;
            dCprime[0][0] = deltaX/0.6875f; dCprime[0][1] = 0.0f;
            dCprime[1][0] = deltaY/0.6875f; dCprime[1][1] = 0.0f;
        }
        else if(index == n-3)
        {
            k = m - 2;
            dCprime[0][0] = deltaX/0.75f; dCprime[0][1] = 0.0f;
            dCprime[1][0] = deltaY/0.75f; dCprime[1][1] = 0.0f;
        }
        else if(index == n-2)
        {
            k = m - 2;
            dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
            dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
        }
        else if(index == n-1)
        {
            k = m - 1;
            dCprime[0][0] = deltaX; dCprime[0][1] = 0.0f;
            dCprime[1][0] = deltaY; dCprime[1][1] = 0.0f;
        }
    }
}

```



```

else
{  if(index % 2 == 0)  // multiple influence
  {  k = index/2;
    dCprime[0][0] = deltaX; dCprime[0][1] = deltaX;
    dCprime[1][0] = deltaY; dCprime[1][1] = deltaY;
  }
  else  // single influence
  {  k = index/2 + 1;
    dCprime[0][0] = deltaX/0.75f; dCprime[0][1] = deltaX;
    dCprime[1][0] = deltaY/0.75f; dCprime[1][1] = deltaY;
  }
}
}

return k;
}

// Interleaves first n elements of p for use with interleaved
// banded diagonal synthesis matrices.
float[] interleave(float[] p, int n)
{  float[] v = new float[p.length];

  // distribute scaling function input
  v[0] = p[0]; v[1] = p[1];
  int i = 3, k = 2;
  while (i < n-1)
  {  v[i] = p[k];
    k++;
    i += 2;
  }
  v[n-1] = p[k];

  // now distribute wavelet input
  i = 2; k++;
  while (i < n-1)
  {  v[i] = p[k];
    k++;
    i += 2;
  }

  i++;

  // copy remaining vector, if any
  while (i < p.length)
  {  v[i] = p[i];
    i++;
  }

  return v;
}

// Reverses interleaving of first n elements of p.
float[] unleave(float[] p, int n)
{  float[] v = new float[p.length];

  // redistribute scaling function output
  v[0] = p[0]; v[1] = p[1];

```

```

int i = 2, k = 3;
while (k < n-1)
{ v[i] = p[k];
  i++;
  k += 2;
}
v[i] = p[n-1];

// now, redistribute wavelet output
i++;
k = 2;
while (k < n-1)
{ v[i] = p[k];
  i++;
  k += 2;
}

// copy remaining vector, if any
while (i < p.length)
{ v[i] = p[i];
  i++;
}

return v;
}

// Returns the number of control points in a B-spline
// curve of resolution level j.
int numPoints(int j)
{ return (int)(Math.pow(2, j)) + 3;
}

// This function is adapted from Numerical Recipes in C,
// Press, Teukolsky, et al. Cambridge Univ. Press, 1992.
//
// Given an n*n matrix a, this routine replaces it by the LU
// decomposition of a rowwise permutation of itself. a is output
// with the U portion of the matrix the upper triangle and main
// diagonal of a and the L portion the lower triangle part (L's main
// diagonal values are all 1.) indx is an output vector that
// records the row permutation effected by the partial pivoting;
// the return value is + or - 1, depending on whether the number of
// row interchanges was even or odd, respectively. Use this routine
// in combination with lubksb to solve linear equations or invert
// a matrix.
public float ludcmp(float[][] a, int[] indx)
{ int imax = -1, n = a.length;
  float d, big, dum, sum, temp;
  float[] vv = new float[n];

  d = 1.0f;
  for(int i = 0; i < n; ++i) // no row interchanges yet
  { big = 0.0f; // loop over rows to get
    for(int j = 0; j < n; ++j) // scaling information
    { temp = Math.abs(a[i][j]);
      if (temp > big)
        big = temp;
    }
  }
}

```

```

    }
    if (big == 0.0f)
    { System.out.println("ERROR: ludcmp: singular matrix");
      return 0.0f;
    }
    vv[i] = 1.0f/big;          // save the scaling
}

// next, loop over the columns per Crout's algorithm:
for(int j = 0; j < n; ++j)
{ for(int i = 0; i < j; ++i)
  { sum = a[i][j];
    for(int k = 0; k < i; ++k)
      sum -= a[i][k] * a[k][j];
    a[i][j] = sum;
  }
  big = 0.0f;                // init. for the search for the
                             // largest pivot element

  for(int i = j; i < n; ++i)
  { sum = a[i][j];
    for (int k = 0; k < j; ++k)
      sum -= a[i][k] * a[k][j];
    a[i][j] = sum;
    dum = vv[i] * Math.abs(sum);
    if (dum >= big)
    { // is the figure of merit for the pivot better than the
      // best so far?
      big = dum;
      imax = i;
    }
  }
  if (j != imax)             // do we need to interchange rows?
  { for (int k = 0; k < n; ++k) // Yes, do so...
    { dum = a[imax][k];
      a[imax][k] = a[j][k];
      a[j][k] = dum;
    }
    d = -1.0f*d;            // ...and change the parity of d
    vv[imax] = vv[j];      // and interchange scale factor.
  }
  indx[j] = imax;
  if (a[j][j] == 0.0f)
    a[j][j] = TINY;

  // if the pivot element is zero the matrix is singular (at
  // least to the precision of the algorithm). For some
  // applications on singular matrices, it is desirable to
  // substitute TINY for 0.0.

  if (j != n)               // now, divide by the pivot element.
  { dum = 1.0f/(a[j][j]);
    for (int i = j+1; i < n; ++i)
      a[i][j] *= dum;
  }
}

return d;

```

```

}

// This function is adapted from Numerical Recipes in C,
// Press, Teukolsky, et al. Cambridge Univ. Press, 1992.
//
// Solves the set of n linear equations A*x = b. Here a is an input
// n*n matrix, not as the matrix a but as its LU decomposition as
// returned from ludcmp. indx is input as the permutation vector
// returned from ludcmp. b is input as the right-hand side vector
// and returns with the solution vector x. a and indx are not
// modified by this routine and can be left in place for successive
// calls with different right-hand sides b.
public void lubksb(float[][] a, int[] indx, float[] b)
{ int n = a.length, ii = -1, ip;
  float sum;

  for (int i = 0; i < n; ++i) // When ii is set to a pos. value,
  { ip = indx[i];           // it will become the index of the
    sum = b[ip];           // first nonvanishing element of b.
    b[ip] = b[i];         // We now do the forward
                          // substitution.

    if (ii > -1)
    { for (int j = ii; j <= i-1; ++j)
      sum -= a[i][j] * b[j];
    }
    else if (sum > 0.0f) // A nonzero element was found,
      ii = i;           // so now we do the sums in the loop.
    b[i] = sum;
  }
  for (int i = n-1; i >= 0; --i) // Now, backsubstitute...
  { sum = b[i];
    for (int j = i+1; j < n; j++)
      sum -= a[i][j] * b[j];
    b[i] = sum/a[i][i]; // Store a component of the sol. x.
  }
}

// This function is adapted from Numerical Recipes in C,
// Press, Teukolsky, et al. Cambridge University Press, 1992.
//
// Matrix multiply b = A*x, where A is a band diagonal with m1 rows
// below the diagonal and m2 rows above. The input vector x and
// output vector b are length n. The array a is n*(m1+m2+1) with
// the diagonal elements in column vector index m1. Subdiagonals
// are in the values a[j..(n-1)][0.(m1-1)], with j > 1 appropriate
// to the number of elements on each subdiagonal. Superdiagonal
// elements are in a[1..j][(m1+1)..(m1+m2)] with j < (n-1)
// appropriate to the number of elements on each superdiagonal.
// MODIFIED (3/14/99): length of vector x may be longer than width
// and so the length of b will be longer than the height of a.
// Therefore, copy any remaining elements from x to b after
// multiplication.
float[] banmul(float[][] a, int m1, int m2, float[] x)
{ int tmploop, n = a.length;
  float[] b = new float[x.length]; //n];

  for (int i = 0; i < n; ++i)

```



```

    { int k = i - m1;
      int w = m1 + m2, u = n - k - 1;
      tmploop = (w < u) ? w : u;
      b[i] = 0.0f;
      for (int j = (0 > -k) ? 0 : -k; j <= tmploop; ++j)
        b[i] += a[i][j] * x[j+k];
    }

    // now, copy over elements from x to b, if any.
    for (int i = n; i < x.length; ++i)
      b[i] = x[i];

    return b;
}

// This function is adapted from Numerical Recipes in C,
// Press, Teukolsky, et al. Cambridge University Press, 1992.
//
// Given an n*n band diagonal matrix A with m1 subdiagonal rows and
// m2 superdiagonal rows, compactly stored in the n*(m1+m2+1) array
// a as described for the method banmul above, this method
// constructs an LU decomposition of a rowwise permutation of A.
// The upper triangular matrix replaces a, while the lower
// triangular matrix is returned in the n*m1 array al. indx is an
// output vector which records the row permutation effected by the
// partial pivoting; d is output as +/- 1 depending on whether the
// number of row interchanges was even or odd, respectively. Use
// this in combination with the method banbks below.
void bandec(float[][] a, int m1, int m2, float[][] al, int[] indx)
{ int mm = m1 + m2 + 1, l = m1, n = a.length;
  float dum;

  // rearrange the storage: left justify the top m1 rows
  for (int i = 0; i < m1; ++i)
  { for (int j = m1-i; j < mm; ++j)
      a[i][j-1] = a[i][j];
    l--;
    for (int j = mm-l-1; j < mm; ++j)
      a[i][j] = 0.0f;
  }
  l = m1;
  for (int k = 0; k < n; ++k) // For each row...
  { dum = a[k][0];
    int i = k;
    if (l < n)
      l++;
    for (int j = k+1; j < l; ++j) // find the pivot element.
    { if (Math.abs(a[j][0]) > Math.abs(dum))
        { dum = a[j][0];
          i = j;
        }
    }
    indx[k] = i;
    if (dum == 0.0) // Matrix is algorithmically singular
      a[k][0] = TINY; // but proceed anyway w/ TINY pivot.
    if (i != k)
      { for (int j = 0; j < mm; ++j) // interchange rows

```

```

        { float temp = a[k][j];
          a[k][j] = a[i][j];
          a[i][j] = temp;
        }
    }
    for (i = k+1; i < l; ++i) // do the elimination
    { dum = a[i][0]/a[k][0];
      al[k][i-k-1] = dum;
      for (int j = 1; j < mm; ++j)
          a[i][j-1] = a[i][j] - dum * a[k][j];
      a[i][mm-1] = 0.0f;
    }
}

// This function is adapted from Numerical Recipes in C,
// Press, Teukolsky, et al. Cambridge Univ. Press, 1992.
//
// Given the arrays a, al and indx, as obtained from bandec, above,
// and given a right-hand side vector b, solves the band diagonal
// linear equations A*x = b. The solution vector x overwrites b.
// The other input arrays are not modified, and can be left in place
// for successive calls with different right-hand sides.
void banbks(float[][] a, int m1, int m2,
            float[][] al, int[] indx, float[] b)
{ int n = a.length, mm = m1 + m2 + 1, l = m1;
  float dum;

  // Forward substitution, unscrambling the permuted rows as we go.
  for (int k = 0; k < n; ++k)
  { int i = indx[k];
    if (i != k)
    { float temp = b[k];
      b[k] = b[i];
      b[i] = temp;
    }
    if (l < n)
        l++;
    for (i = k+1; i < l; ++i)
        b[i] -= al[k][i-k-1] * b[k];
  }

  l = 1;

  // Backsubstitution.
  for (int i = n-1; i >= 0; i--)
  { dum = b[i];
    for (int k = 1; k < l; ++k)
        dum -= a[i][k] * b[k+i];
    b[i] = dum / a[i][0];
    if (l < mm)
        l++;
  }
}

// Converts an array of Point2D to a 2D array of floats.
float[][] Point2DToXY(Point2D[] P)

```

```

{ int n = P.length;
  float[][] points = new float[2][n];

  for (int i = 0; i < n; ++i)
  { points[0][i] = P[i].x;
    points[1][i] = P[i].y;
  }

  return points;
}

// Converts a 2D array of floats to an array of Point2D.
Point2D[] XYToPoint2D(float[][] P)
{ int n = P[0].length;
  Point2D[] points = new Point2D[n];

  for (int i = 0; i < n; ++i)
    points[i] = new Point2D(P[0][i], P[1][i]);

  return points;
}
}

```

```

// File:   CurvEditor.java
// Author: Stephen Alberg
// Uses:   CvBspline and Swing classes
//
// This file the class definition for an interactive multiresolution
// curve editor.  The CurvEditor class invokes components of the Swing
// library to provide a fluent interface to the program.  The CvBspline
// class is a canvas object within which multiple curves may be edited.
// When a user request to edit the resolution of a curve is made, the
// CvBspline object calls upon the coarsen/refine methods of its member
// Multires object, receiving as a return value the modified set of
// control points for the curve
//
// TO DO:  scribble translation interface, zoom control, knot editing
// vs. control point editing (using Hermite|Bspline conversion).
//
// Original source for the curve entering and drawing functionality:
// Leen Ammeraal, Computer Graphics for Java Programmers.  Wiley, 1998.

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class CurvEditor extends JFrame
{ public static void main(String[] args)
  { // use system look and feel
    try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    } catch(Exception e){}
    new CurvEditor();
  }

  static JFrame instance;

  JSlider slider;
  CvBspline canvas;

  JToggleButton edit = new JToggleButton();
  JToggleButton scribble = new JToggleButton();
  JToggleButton draw = new JToggleButton();
  JToggleButton zoom = new JToggleButton();
  JPopupMenu zoomPopup = new JPopupMenu();
  JToggleButton zoomPlus = new JToggleButton();
  JToggleButton zoomMinus = new JToggleButton();
  JToggleButton zoomOne = new JToggleButton();
  JButton polygon = new JButton();
  JButton points = new JButton();
  JButton knots = new JButton();
  JButton grid = new JButton();
  JButton coords = new JButton();
  JButton clear = new JButton();

```



```

JLabel resLabel = new JLabel("Resolution value: 0.0");
JLabel coordsLabel = new JLabel("Position: ");

// image icons for buttons
ImageIcon editUnselected, editRollover, editSelected;
ImageIcon scribbleUnselected, scribbleRollover, scribbleSelected;
ImageIcon drawUnselected, drawRollover, drawSelected;
ImageIcon zoomUnselected, zoomRollover, zoomSelected;
ImageIcon zoomPlusUnselected, zoomPlusRollover, zoomPlusSelected;
ImageIcon zoomMinusUnselected, zoomMinusRollover, zoomMinusSelected;
ImageIcon zoomOneUnselected, zoomOneRollover, zoomOneSelected;
ImageIcon polygonUnselected, polygonRollover, polygonSelected;
ImageIcon pointsUnselected, pointsRollover, pointsSelected;
ImageIcon knotsUnselected, knotsRollover, knotsSelected;
ImageIcon gridUnselected, gridRollover, gridSelected;
ImageIcon coordsUnselected, coordsRollover, coordsSelected;
ImageIcon clearUnselected, clearRollover;
ImageIcon canvasBG;
ImageIcon crvIcon, difIcon;
ImageIcon textureIcon1, textureIcon2, textureIcon3;

// menu items for menubar
JMenuItem exit;
JMenuItem undo, redo, select, sketch, build;
JMenuItem zoomIn, zoomOut, clearScreen;
JMenuItem shPolygon, shPoints, shKnots, shGrid, shCoords;
JMenuItem importTexture, gridSpacing, undoStack;
JMenuItem about;

// dialog boxes
JDialog aboutBox;

boolean shiftAllowed = true;

// Constructor
CurvEditor()
{ super("CurvEditor v.1.1");
  instance = this;

  addWindowListener(new WindowAdapter()
  { public void windowClosing(WindowEvent e)
    { System.exit(0);
    }
  });

  setSize(800, 600);
  getContentPane().setLayout(new BorderLayout());

  slider = new JSlider(JSlider.HORIZONTAL, 0, 80, 0);
  canvas = new CvBspline(slider);
  canvasBG = new ImageIcon("images/bigwhite.gif");
  canvas.setIcon(canvasBG);

  JScrollPane pane = new JScrollPane(canvas);
  getContentPane().add(pane, BorderLayout.CENTER);

```

```

loadImages();

JMenuBar menubar = createMenuBar();
setJMenuBar(menubar);

JToolBar toolbar = new JToolBar(JToolBar.VERTICAL);
initButtons(toolbar);
getContentPane().add(toolbar, BorderLayout.WEST);

slider.setPaintTicks(true);
slider.setMajorTickSpacing(10);
slider.setMinorTickSpacing(1);
slider.setSnapToTicks(true);
slider.setPaintLabels(false);
slider.setBorder(new BevelBorder(BevelBorder.LOWERED));
slider.addChangeListener(new ChangeListener()
{ public void stateChanged(ChangeEvent e)
  { if (shiftAllowed)
    { shiftAllowed = false;
      slider.setEnabled(false);
      canvas.shiftResolution(
        (float)(slider.getValue())/10.0f);
      resLabel.setText("Resolution value: " +
        (double)(slider.getValue())/10.0);
      slider.setEnabled(true);
      shiftAllowed = true;
    }
  }
});
this.addKeyListener(new KeyListener()
{ public void keyPressed(KeyEvent e)
  { if (e.getKeyCode() == KeyEvent.VK_LEFT)
    slider.setValue(slider.getValue() - 1);

    else if(e.getKeyCode() == KeyEvent.VK_RIGHT)
    slider.setValue(slider.getValue() + 1);

    canvas.shiftResolution((float)(slider.getValue())/10.0f);
    resLabel.setText("Resolution value: " +
      (double)(slider.getValue())/10.0);
  }

  // dummy functions necessary to implement abstract class
  public void keyReleased(KeyEvent e){}
  public void keyTyped(KeyEvent e){}
});

// Set layout of bottom of interface
canvas.setCoordsLabel(coordsLabel);

JPanel p = new JPanel();
p.setLayout(new BorderLayout(p, BorderLayout.Y_AXIS));
p.setBackground(Color.lightGray);
getContentPane().add(p, BorderLayout.SOUTH);

JPanel q = new JPanel();
q.setLayout(new BorderLayout(q, BorderLayout.X_AXIS));

```

```

q.add(Box.createRigidArea(new Dimension(75,20)));
resLabel.setPreferredSize(new Dimension(140, 25));
q.add("West", resLabel);
q.add(Box.createRigidArea(new Dimension(75,20)));

coordsLabel.setPreferredSize(new Dimension(240, 25));
q.add("West", coordsLabel);
q.add(Box.createGlue());

p.add(q);

q = new JPanel();
q.setLayout(new BorderLayout(q, BorderLayout.X_AXIS));

q.add(Box.createRigidArea(new Dimension(75,25)));
slider.setPreferredSize(new Dimension(525, 50));
q.add("West", slider);
q.add(Box.createRigidArea(new Dimension(75,25)));

p.add(q);
p.add(Box.createRigidArea(new Dimension(75,25)));

show();
}

public static JFrame sharedInstance()
{ return instance;
}

void loadImages()
{ editUnselected = new ImageIcon("images/hand.gif");
  editRollover   = new ImageIcon("images/hand1.gif");
  editSelected   = new ImageIcon("images/hand2.gif");
  scribbleUnselected = new ImageIcon("images/pencil.gif");
  scribbleRollover  = new ImageIcon("images/pencil1.gif");
  scribbleSelected  = new ImageIcon("images/pencil2.gif");
  drawUnselected   = new ImageIcon("images/curve.gif");
  drawRollover     = new ImageIcon("images/curver.gif");
  drawSelected     = new ImageIcon("images/curve2.gif");
  zoomUnselected   = new ImageIcon("images/zoom.gif");
  zoomRollover     = new ImageIcon("images/zoom1.gif");
  zoomSelected     = new ImageIcon("images/zoom2.gif");
  zoomPlusUnselected = new ImageIcon("images/zoomplus.gif");
  zoomPlusRollover  = new ImageIcon("images/zoomplus1.gif");
  zoomPlusSelected  = new ImageIcon("images/zoomplus2.gif");
  zoomMinusUnselected = new ImageIcon("images/zoomminus.gif");
  zoomMinusRollover  = new ImageIcon("images/zoomminus1.gif");
  zoomMinusSelected  = new ImageIcon("images/zoomminus2.gif");
  zoomOneUnselected = new ImageIcon("images/zoomone.gif");
  zoomOneRollover   = new ImageIcon("images/zoomone1.gif");
  zoomOneSelected   = new ImageIcon("images/zoomone2.gif");
  polygonUnselected = new ImageIcon("images/polygon.gif");
  polygonRollover   = new ImageIcon("images/polygon1.gif");
  polygonSelected   = new ImageIcon("images/polygon2.gif");
  pointsUnselected  = new ImageIcon("images/points.gif");
  pointsRollover    = new ImageIcon("images/points1.gif");
}

```

```

pointsSelected = new ImageIcon("images/points2.gif");
knotsUnselected = new ImageIcon("images/knots.gif");
knotsRollover = new ImageIcon("images/knots1.gif");
knotsSelected = new ImageIcon("images/knots3.gif");
gridUnselected = new ImageIcon("images/grid.gif");
gridRollover = new ImageIcon("images/grid1.gif");
gridSelected = new ImageIcon("images/grid2.gif");
coordsUnselected = new ImageIcon("images/coords.gif");
coordsRollover = new ImageIcon("images/coords1.gif");
coordsSelected = new ImageIcon("images/coords2.gif");
clearUnselected = new ImageIcon("images/clear.gif");
clearRollover = new ImageIcon("images/clear1.gif");
crvIcon = new ImageIcon("images/crvicon.gif");
difIcon = new ImageIcon("images/dificon.gif");
textureIcon1 = new ImageIcon("images/tx1.gif");
textureIcon2 = new ImageIcon("images/tx2.gif");
textureIcon3 = new ImageIcon("images/tx3.gif");
}

```

```

void initButtons(JToolBar toolbar)
{
    ButtonGroup bg = new ButtonGroup();
    Insets margin = new Insets(0,0,0,0);

    edit.setIcon(editUnselected);
    edit.setRolloverIcon(editRollover);
    edit.setSelectedIcon(editSelected);
    edit.setSelected(true);
    edit.setToolTipText("Edit/Select Curve");
    edit.setMargin(margin);
    bg.add(edit);
    toolbar.add(edit);

    scribble.setIcon(scribbleUnselected);
    scribble.setRolloverIcon(scribbleRollover);
    scribble.setSelectedIcon(scribbleSelected);
    scribble.setToolTipText("Write Curve");
    scribble.setMargin(margin);
    bg.add(scribble);
    toolbar.add(scribble);

    draw.setIcon(drawUnselected);
    draw.setRolloverIcon(drawRollover);
    draw.setSelectedIcon(drawSelected);
    draw.setToolTipText("Build Curve");
    draw.setMargin(margin);
    bg.add(draw);
    toolbar.add(draw);

    zoom.setIcon(zoomUnselected);
    zoom.setRolloverIcon(zoomRollover);
    zoom.setSelectedIcon(zoomSelected);
    zoom.setToolTipText("Zoom");
    zoom.setMargin(margin);
    bg.add(zoom);
    toolbar.add(zoom);

    bg = new ButtonGroup();
}

```



```

zoomPlus.setIcon(zoomPlusUnselected);
zoomPlus.setRolloverIcon(zoomPlusRollover);
zoomPlus.setSelectedIcon(zoomPlusSelected);
zoomPlus.setToolTipText("Zoom in");
zoomPlus.setMargin(margin);
bg.add(zoomPlus);
zoomPopup.add(zoomPlus);

zoomMinus.setIcon(zoomMinusUnselected);
zoomMinus.setRolloverIcon(zoomMinusRollover);
zoomMinus.setSelectedIcon(zoomMinusSelected);
zoomMinus.setToolTipText("Zoom out");
zoomMinus.setMargin(margin);
bg.add(zoomMinus);
zoomPopup.add(zoomMinus);

zoomOne.setIcon(zoomOneUnselected);
zoomOne.setRolloverIcon(zoomOneRollover);
zoomOne.setSelectedIcon(zoomOneSelected);
zoomOne.setToolTipText("Zoom default");
zoomOne.setMargin(margin);
bg.add(zoomOne);
zoomPopup.add(zoomOne);

toolbar.addSeparator(new Dimension(5,5));

polygon.setIcon(polygonUnselected);
polygon.setRolloverIcon(polygonRollover);
polygon.setToolTipText("Hide Polygon");
polygon.setMargin(margin);
toolbar.add(polygon);

points.setIcon(pointsUnselected);
points.setRolloverIcon(pointsRollover);
points.setToolTipText("Hide Points");
points.setMargin(margin);
toolbar.add(points);

knots.setIcon(knotsSelected);
knots.setRolloverIcon(knotsRollover);
knots.setToolTipText("Show Knots");
knots.setMargin(margin);
toolbar.add(knots);

grid.setIcon(gridUnselected);
grid.setRolloverIcon(gridRollover);
grid.setToolTipText("Show Grid");
grid.setMargin(margin);
toolbar.add(grid);

coords.setIcon(coordsUnselected);
coords.setRolloverIcon(coordsRollover);
coords.setToolTipText("Show Coordinates");
coords.setMargin(margin);
toolbar.add(coords);

clear.setIcon(clearUnselected);

```

```

clear.setRolloverIcon(clearRollover);
clear.setToolTipText("Clear Screen");
clear.setMargin(margin);
toolbar.add(clear);

edit.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.setOpState(CvBspline.EDIT);
  }
});

scribble.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.setOpState(CvBspline.SKETCH);
  }
});

draw.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.setOpState(CvBspline.DRAW);
  }
});

zoom.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { //canvas.setOpState(CvBspline.ZOOM);
    zoomPopup.show(zoom, zoom.getWidth(), 10);
  }
});

zoomPlus.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.setOpState(CvBspline.ZOOM);
    // perform the zoom...

    // reset the system to edit mode
    edit.setSelected(true);
    //canvas.setOpState(CvBspline.EDIT);
    zoomPlus.setSelected(false);
    repaint();
  }
});

zoomMinus.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.setOpState(CvBspline.ZOOM);
    // perform the zoom...

    // reset the system to edit mode
    edit.setSelected(true);
    //canvas.setOpState(CvBspline.EDIT);
    zoomMinus.setSelected(false);
    repaint();
  }
});

zoomOne.addActionListener(new ActionListener()

```

```

    { public void actionPerformed(ActionEvent e)
      { canvas.setOpState(CvBspline.ZOOM);
        // perform the zoom...

        // reset the system to edit mode
        edit.setSelected(true);
        //canvas.setOpState(CvBspline.EDIT);
        zoomOne.setSelected(false);
        repaint();
      }
    });

    clear.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.clear();
        edit.setSelected(true);
        repaint();
      }
    });

    knots.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.toggleKnots();
        swapKnotsLabel();
      }
    });

    polygon.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.togglePolygon();
        swapPolygonLabel();
      }
    });

    points.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.togglePoints();
        swapPointsLabel();
      }
    });

    grid.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.toggleGrid();
        swapGridLabel();
      }
    });

    coords.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.toggleCoords();
        swapCoordsLabel();
      }
    });
}

JMenuBar createMenuBar()

```

```

{ JMenuBar menubar = new JMenuBar();

// Create File menu
JMenu file = new JMenu("File");
file.setMnemonic('F');
JMenuItem mi;
mi = new JMenuItem("New...");
mi.setMnemonic('N');
mi.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { // show dialog box asking if we want to save
    // the current drawing (set of curves) first
    Vector currState = canvas.saveCurve();

    if (currState.size() > 0)
    { //if answer from dialog box is yes
      int result = JOptionPane.showConfirmDialog(
        CurvEditor.sharedInstance(),
        "Do you wish to save the present drawing?"
      );
      if (result == JOptionPane.YES_OPTION)
        saveFile();
      if (result != JOptionPane.CANCEL_OPTION)
        canvas.clear();
    }
    else
      canvas.clear();
  }
});

void saveFile()
{ JFileChooser saveChooser = new JFileChooser();
  ExampleFileFilter filter = new ExampleFileFilter(
    new String[] { "crv" }, "CurvEditor files"
  );
  ExampleFileView fileView = new ExampleFileView();
  fileView.putIcon("crv", crvIcon);
  saveChooser.setFileView(fileView);
  saveChooser.addChoosableFileFilter(filter);
  saveChooser.setFileFilter(filter);
  saveChooser.setCurrentDirectory(
    new File("CurvEditor.class"));

  int retval = saveChooser.showSaveDialog(
    CurvEditor.sharedInstance());

  if (retval == 0)
  { File theFile = saveChooser.getSelectedFile();
    // Load selected curve into environment
    if (theFile != null)
    { try
      { ObjectOutputStream os = new ObjectOutputStream(
        new FileOutputStream(theFile)
      );
      Vector newState = canvas.saveCurve();
      os.writeObject(newState);
      os.close();
    } catch(Exception ex){
      System.out.println(ex);
    }
  }
}

```



```

        }
    }
}
));
file.add(mi);
mi = new JMenuItem("Open...");
mi.setMnemonic('O');
mi.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        //JFileChooser chooser = new JFileChooser();
        JFileChooser openChooser = new JFileChooser();
        ExampleFileFilter filter = new ExampleFileFilter(
            new String[] { "crv" }, "CurvEditor files"
        );
        ExampleFileView fileView = new ExampleFileView();
        fileView.putIcon("crv", crvIcon);
        openChooser.setFileView(fileView);
        openChooser.addChoosableFileFilter(filter);
        openChooser.setFileFilter(filter);
        openChooser.setCurrentDirectory(
            new File("CurvEditor.class"));

        int retval = openChooser.showOpenDialog(
            CurvEditor.sharedInstance());

        if (retval == 0)
        {
            File theFile = openChooser.getSelectedFile();
            // Load selected curve into environment
            if (theFile != null)
            {
                try
                {
                    ObjectInputStream os = new ObjectInputStream(
                        new FileInputStream(theFile)
                    );
                    Vector newState = (Vector)os.readObject();
                    canvas.loadCurve(newState);
                    os.close();
                }
                catch (Exception ex) {
                    System.out.println(ex);
                }
            }
        }
    }
});
file.add(mi);
mi = new JMenuItem("Save");
mi.setMnemonic('S');
mi.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        JFileChooser saveChooser = new JFileChooser();
        ExampleFileFilter filter = new ExampleFileFilter(
            new String[] { "crv" }, "CurvEditor files"
        );
        ExampleFileView fileView = new ExampleFileView();
        fileView.putIcon("crv", crvIcon);
        saveChooser.setFileView(fileView);
        saveChooser.addChoosableFileFilter(filter);
    }
});

```

```

saveChooser.setFileFilter(filter);
saveChooser.setCurrentDirectory(
    new File("CurvEditor.class"));

int retval = saveChooser.showSaveDialog(
    CurvEditor.sharedInstance());
if (retval == 0)
{
    File theFile = saveChooser.getSelectedFile();
    // Load selected curve into environment
    if (theFile != null)
    {
        try
        {
            ObjectOutputStream os = new ObjectOutputStream(
                new FileOutputStream(theFile)
            );
            Vector newState = canvas.saveCurve();
            os.writeObject(newState);
            os.close();
        }
        catch (Exception ex) {
            System.out.println(ex);
        }
    }
}
});
file.add(mi);
mi = new JMenuItem("Save As...");
mi.setMnemonic('A');
mi.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        JFileChooser saveChooser = new JFileChooser();
        ExampleFileFilter filter = new ExampleFileFilter(
            new String[] { "*" }, "All files"
        );
        ExampleFileView fileView = new ExampleFileView();
        fileView.putIcon("crv", crvIcon);
        saveChooser.setFileView(fileView);
        saveChooser.addChoosableFileFilter(filter);
        saveChooser.setFileFilter(filter);
        saveChooser.setCurrentDirectory(
            new File("CurvEditor.class"));

        int retval = saveChooser.showSaveDialog(
            CurvEditor.sharedInstance());
        if (retval == 0)
        {
            File theFile = saveChooser.getSelectedFile();
            // Load selected curve into environment
            if (theFile != null)
            {
                try
                {
                    ObjectOutputStream os = new ObjectOutputStream(
                        new FileOutputStream(theFile)
                    );
                    Vector newState = canvas.saveCurve();
                    os.writeObject(newState);
                    os.close();
                }
                catch (Exception ex) {
                    System.out.println(ex);
                }
            }
        }
    }
});

```

```

    }
    }
    }
    });
    file.add(mi);
    file.add(new JSeparator());
    importTexture = new JMenu("Import Texture");
    importTexture.setMnemonic('I');
    mi = new JMenuItem(textureIcon1);
    mi.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { loadTexture(textureIcon1);
      }
    });
    importTexture.add(mi);
    mi = new JMenuItem(textureIcon2);
    mi.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { loadTexture(textureIcon2);
      }
    });
    importTexture.add(mi);
    mi = new JMenuItem(textureIcon3);
    mi.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { loadTexture(textureIcon3);
      }
    });
    importTexture.add(mi);
    importTexture.add(new JSeparator());
    mi = new JMenuItem("User-defined Texture...");
    mi.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { loadTexture(null);
      }
    });
    importTexture.add(mi);
    file.add(importTexture);
    mi = new JMenuItem("Save As Texture...");
    mi.setMnemonic('T');
    mi.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      {
        JFileChooser saveChooser = new JFileChooser();
        ExampleFileFilter filter = new ExampleFileFilter(
          new String[] { "dif" }, "Curve Texture files"
        );
        ExampleFileView fileView = new ExampleFileView();
        fileView.putIcon("dif", difIcon);
        saveChooser.setFileView(fileView);
        saveChooser.addChoosableFileFilter(filter);
        saveChooser.setFileFilter(filter);
        saveChooser.setCurrentDirectory(
          new File("CurvEditor.class"));

        int retval = saveChooser.showSaveDialog(

```

```

CurvEditor.sharedInstance());
if (retval == 0)
{
    File theFile = saveChooser.getSelectedFile();
    // Load selected curve into environment
    if (theFile != null)
    {
        try
        {
            ObjectOutputStream os = new ObjectOutputStream(
                new FileOutputStream(theFile)
            );
            Vector newTexture = canvas.saveTexture();
            os.writeObject(newTexture);
            os.close();
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }
}
});
file.add(mi);
file.add(new JSeparator());
exit = new JMenuItem("Exit");
exit.setMnemonic('x');
exit.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
file.add(exit);
menubar.add(file);

// Create Edit menu
JMenu mEdit = new JMenu("Edit");
mEdit.setMnemonic('E');
undo = new JMenuItem("Undo");
undo.setMnemonic('U');
undo.setEnabled(false);
undo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        canvas.undoLastMove();
        repaint();
    }
});
mEdit.add(undo);
redo = new JMenuItem("Redo");
redo.setMnemonic('R');
redo.setEnabled(false);
redo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        canvas.redoLastMove();
        repaint();
    }
});
mEdit.add(redo);
mEdit.add(new JSeparator());
select = new JMenuItem("Select");
select.setMnemonic('l');

```



```

select.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        canvas.setOpState(CvBspline.EDIT);
        edit.setSelected(true);
    }
});
mEdit.add(select);
sketch = new JMenuItem("Sketch");
sketch.setMnemonic('k');
sketch.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        canvas.setOpState(CvBspline.SKETCH);
        scribble.setSelected(true);
    }
});
mEdit.add(sketch);
build = new JMenuItem("Build");
build.setMnemonic('B');
build.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        canvas.setOpState(CvBspline.DRAW);
        draw.setSelected(true);
    }
});
mEdit.add(build);
mEdit.add(new JSeparator());
zoomIn = new JMenuItem("Zoom In");
zoomIn.setMnemonic('Z');
mEdit.add(zoomIn);
zoomOut = new JMenuItem("Zoom Out");
zoomOut.setMnemonic('o');
mEdit.add(zoomOut);
clearScreen = new JMenuItem("Clear Screen");
clearScreen.setMnemonic('C');
clearScreen.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        canvas.clear();
        edit.setSelected(true);
        repaint();
    }
});
mEdit.add(clearScreen);
mEdit.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        if (canvas.undoStackEmpty())
            undo.setEnabled(false);
        else
            undo.setEnabled(true);

        if (canvas.redoStackEmpty())
            redo.setEnabled(false);
        else
            redo.setEnabled(true);
    }
});
menubar.add(mEdit);

```

```

// Create Options menu
JMenu options = new JMenu("Options");
options.setMnemonic('p');
shPolygon = new JMenuItem("Hide Polygon");
shPolygon.setMnemonic('y');
shPolygon.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.togglePolygon();
    swapPolygonLabel();
  }
});
options.add(shPolygon);
shPoints = new JMenuItem("Hide Points");
shPoints.setMnemonic('i');
shPoints.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.togglePoints();
    swapPointsLabel();
  }
});
options.add(shPoints);
shKnots = new JMenuItem("Show Knots");
shKnots.setMnemonic('K');
shKnots.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.toggleKnots();
    swapKnotsLabel();
  }
});
options.add(shKnots);
shGrid = new JMenuItem("Show Grid");
shGrid.setMnemonic('G');
shGrid.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.toggleGrid();
    swapGridLabel();
  }
});
options.add(shGrid);
shCoords = new JMenuItem("Show Coords");
shCoords.setMnemonic('d');
shCoords.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)
  { canvas.toggleCoords();
    swapCoordsLabel();
  }
});
options.add(shCoords);
options.add(new JSeparator());
gridSpacing = new JMenu("Grid Spacing");
JCheckBoxMenuItem temp;
ButtonGroup bg = new ButtonGroup();
temp = (JCheckBoxMenuItem) gridSpacing.add(
  new JCheckBoxMenuItem("10"));
temp.setSelected(true);
temp.addActionListener(new ActionListener()
{ public void actionPerformed(ActionEvent e)

```

```

        { canvas.setGridSpacing(10.0f);
        }
    });
    bg.add(temp);
    temp = (JCheckBoxMenuItem) gridSpacing.add(
        new JCheckBoxMenuItem("20"));
    temp.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.setGridSpacing(20.0f);
      }
    });
    bg.add(temp);
    temp = (JCheckBoxMenuItem) gridSpacing.add(
        new JCheckBoxMenuItem("25"));
    temp.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.setGridSpacing(25.0f);
      }
    });
    bg.add(temp);
    options.add(gridSpacing);
    undoStack = new JMenu("Undo Stack Depth");
    bg = new ButtonGroup();
    temp = (JCheckBoxMenuItem) undoStack.add(
        new JCheckBoxMenuItem("1"));
    temp.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.setUndoLimit(1);
      }
    });
    bg.add(temp);
    temp = (JCheckBoxMenuItem) undoStack.add(
        new JCheckBoxMenuItem("5"));
    temp.setSelected(true);
    temp.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.setUndoLimit(5);
      }
    });
    bg.add(temp);
    temp = (JCheckBoxMenuItem) undoStack.add(
        new JCheckBoxMenuItem("10"));
    temp.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.setUndoLimit(10);
      }
    });
    bg.add(temp);
    temp = (JCheckBoxMenuItem) undoStack.add(
        new JCheckBoxMenuItem("15"));
    temp.addActionListener(new ActionListener()
    { public void actionPerformed(ActionEvent e)
      { canvas.setUndoLimit(15);
      }
    });
    bg.add(temp);
    options.add(undoStack);

```

```

menubar.add(options);

// Create Help menu
JMenu help = new JMenu("Help");
help.setMnemonic('H');
about = new JMenuItem("About...");
about.setMnemonic('t');
about.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        if (aboutBox == null)
        {
            aboutBox = new JDialog(CurvEditor.sharedInstance(),
                "About CurvEditor v1.1", false);
            JPanel authorPanel = new JPanel(new BorderLayout());
            ImageIcon author =
                new ImageIcon("images/copyright.gif");
            aboutBox.getContentPane().add(authorPanel,
                BorderLayout.CENTER);
            JLabel authorLabel = new JLabel(author);
            authorPanel.add(authorLabel, BorderLayout.CENTER);
            JPanel buttonPanel = new JPanel(true);
            authorPanel.add(buttonPanel, BorderLayout.SOUTH);
            JButton button = (JButton) buttonPanel.add(
                new JButton("OK"));
            button.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    aboutBox.setVisible(false);
                }
            });
        }
        aboutBox.pack();
        aboutBox.show();
    }
});
help.add(about);
menubar.add(help);

return menubar;
}

void loadTexture(ImageIcon icon)
{
    File theFile = null;

    if(icon == textureIcon1)
        theFile = new File("tex1.dif");
    else if(icon == textureIcon2)
        theFile = new File("tex2.dif");
    else if(icon == textureIcon3)
        theFile = new File("tex3.dif");
    else
    {
        JFileChooser chooser = new JFileChooser();
        ExampleFileFilter filter = new ExampleFileFilter(
            new String[] { "dif" }, "Curve Texture files"
        );
        ExampleFileView fileView = new ExampleFileView();
        fileView.putIcon("dif", difIcon);
        chooser.setFileView(fileView);
        chooser.addChoosableFileFilter(filter);
    }
}

```



```

chooser.setFileFilter(filter);
chooser.setCurrentDirectory(new File("CurvEditor.class"));

int retval =
    chooser.showOpenDialog(CurvEditor.sharedInstance());

if(retval == 0)
    theFile = chooser.getSelectedFile();
}

if(theFile != null)
{ try
  { ObjectInputStream os = new ObjectInputStream(
      new FileInputStream(theFile)
    );
    Vector newTexture = (Vector)os.readObject();
    canvas.importTexture(newTexture);
    os.close();
  } catch(Exception ex){
    System.out.println(ex);
  }
}
}

void swapKnotsLabel()
{ if (knots.getIcon() == knotsUnselected)
  { knots.setIcon(knotsSelected);
    knots.setToolTipText("Show Knots");
    shKnots.setText("Show Knots");
  }
  else
  { knots.setIcon(knotsUnselected);
    knots.setToolTipText("Hide Knots");
    shKnots.setText("Hide Knots");
  }
}

void swapPolygonLabel()
{ if (polygon.getIcon() == polygonUnselected)
  { polygon.setIcon(polygonSelected);
    polygon.setToolTipText("Show Polygon");
    shPolygon.setText("Show Polygon");
  }
  else
  { polygon.setIcon(polygonUnselected);
    polygon.setToolTipText("Hide Polygon");
    shPolygon.setText("Hide Polygon");
  }
}

void swapPointsLabel()
{ if (points.getIcon() == pointsUnselected)
  { points.setIcon(pointsSelected);
    points.setToolTipText("Show Points");
    shPoints.setText("Show Points");
  }
  else

```

```

    { points.setIcon(pointsUnselected);
      points.setToolTipText("Hide Points");
      shPoints.setText("Hide Points");
    }
}

void swapGridLabel()
{ if (grid.getIcon() == gridUnselected)
  { grid.setIcon(gridSelected);
    grid.setToolTipText("Hide Grid");
    shGrid.setText("Hide Grid");
  }
  else
  { grid.setIcon(gridUnselected);
    grid.setToolTipText("Show Grid");
    shGrid.setText("Show Grid");
  }
}

void swapCoordsLabel()
{ if (coords.getIcon() == coordsUnselected)
  { coords.setIcon(coordsSelected);
    coords.setToolTipText("Hide Coordinates");
    shCoords.setText("Hide Coordinates");
  }
  else
  { coords.setIcon(coordsUnselected);
    coords.setToolTipText("Show Coordinates");
    shCoords.setText("Show Coordinates");
  }
}
}

```

```

// File:   CvBspline.java
// Author: Stephen Alberg
// Uses:   Point2D, Multires, UndoStack, CurveState and Swing classes
//
// This file the class definition for an interactive multiresolution
// curve editor.  The CvBspline class is a scrollable client canvas
// object.  It inherits from the JViewport object which forms part of
// the JScrollPane in a CurvEditor object.  This canvas-type object
// relates mouse inputs and selections from the CurvEditor interface to
// the member Multires object for processing shifts in curve
// resolution.  Requires that a JSlider object is instantiated in the
// interface object prior to this object's instantiation.
//
// TO DO:  scribble translation interface, zoom control, knot editing
// vs. control point editing (using Hermite|Bspline conversion).
//
// Original source for the curve entering and drawing functionality:
// Leen Ammeraal, Computer Graphics for Java Programmers.  Wiley, 1998.

import java.awt.*;
import java.awt.event.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class CvBspline extends JLabel
{ // constants
  static final Cursor EDIT_CURSOR =
    Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR);
  static final Cursor DRAW_CURSOR =
    Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR);
  static final Cursor WAIT_CURSOR =
    Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);

  // system state IDs
  static final int EDIT = 1;
  static final int SKETCH = 2;
  static final int DRAW = 3;
  static final int ZOOM = 4;

  // C is the currently selected CurveState
  CurveState C;

  // State contains all of the curves in the present environment.
  Vector State = new Vector();

  // Scribble contains a set of data points used for saving
  // a sketched line in the editing system.
  Vector Scribble = new Vector();

  // Two stacks, one for preserving the last undoLimit moves, the
  // other for saving moves as undo operations are applied.
  UndoStack undo = new UndoStack();
  UndoStack redo = new UndoStack();

```

```

// Instance variables
int centerX, centerY,      // coordinates of screen center
    index = -1,           // index of selected control point
    stateIndex = -1;      // index of current curve in State vector

float rWidth = 1000.0F, rHeight = 750.0F,
    eps = rWidth/100F, pixelSize;
float zoomX = 0.0f, zoomY = 0.0f, zoomXt = 0.0f, zoomYt = 0.0f;

int opState = EDIT;
int undoLimit = 5;
float gridSpacing = 10.0f;

float mouseX = rWidth, mouseY = rHeight; // used in EDIT operations
float lastX = 0.0f, lastY = 0.0f;       // used by Scribble vector

boolean showCoords = false,
    showGrid = false,
    showKnots = false,
    showRect = true,
    showPolygon = true,
    showPoints = true,
    editCurve = false,
    mouseDown = false,
    shiftAllowed = true,
    paintFirstTime = true;

Multires multires = new Multires();
JSlider slider;
JLabel coordsLabel;

// Constructor
CvBspline(JSlider s)
{ super();

    this.slider = s;
    this.slider.setValue(0);
    this.slider.setEnabled(false);
    TitledBorder tb = (TitledBorder)this.slider.getBorder();

    setCursor(EDIT_CURSOR);

    addMouseListener(new MouseAdapter()
    { public void mousePressed(MouseEvent evt)
      { float x = fx(evt.getX()), y = fy(evt.getY());
        mouseX = x; mouseY = y;

        switch(opState)
        {
        case DRAW:
            // If there is no current curve, start a new one
            if (C == null)
            { C = new CurveState(State.size());
              State.addElement(C);
            }
            else // otherwise, save the curve to the undo stack
              undo.push(C.clone());
        }
      }
    });
}

```



```

// empty the redo stack
redo.removeAllElements();

C.V.addElement(new Point2D(x, y));
C.np++;
if (C.np == C.nextNPsize)
{ C.jCurr++;
  C.nextNPsize = multires.numPoints(C.jCurr + 1);

  // Add coarser curve rep to CurveState
  if (C.jCurr > 0)
  { Point2D[] P = new Point2D[C.V.size()];
    copyFrom(C.V, P);
    P = multires.coarsen(P, C.jCurr);
    copyBack(C.Vl, P);
  }

  //slider.setValue(C.jCurr*10);
  resetSliderValue();
}
break;
case SKETCH:
  // Reset the Scribble vector
  Scribble = new Vector();
  Scribble.addElement(new Point2D(x, y));
  break;
case EDIT:
  // If no curve or control point was clicked on,
  // set current curve to null
  stateIndex = -1;
  for (int j = 0; j < State.size(); ++j)
  { CurveState cTemp = (CurveState) (State.elementAt(j));
    if (index < 0)
    { for (int i = 0; i < cTemp.V.size(); ++i)
      { if (onPoint((Point2D) (cTemp.V.elementAt(i)),
                    x, y))
        { index = i; i = cTemp.V.size();
          C = cTemp;
          stateIndex = j; j = State.size();
        }
      }
    }
  }
}

// If we deselected a curve, save the last
// version of it to the undo stack.
if (C != null)
{ undo.push(C.clone());
  redo.removeAllElements();
}
if (stateIndex < 0)
{ C = null;
  slider.setValue(0);
}

break;

```

```

    case ZOOM:
        zoomX = zoomXt = x; zoomY = zoomYt = y;
        break;
    };

    repaint();
}

public void mouseReleased(MouseEvent evt)
{
    switch(opState)
    {
        case DRAW:
            // If number of segments in curve is not a power of 2,
            // insert knots into curve until next power of 2 is
            // reached.
            // DON'T CHECK THIS HERE! Wait until the opState is
            // changed and update it then.
            break;
        case SKETCH:
            float sx = fx(evt.getX()), sy = fy(evt.getY());
            Scribble.addElement(new Point2D(sx, sy));

            // Convert scribble to CurveState and add to State
            //C = ScribbleToCurve(Scribble, State.size());
            //State.addElement(C);
            //Scribble = new Vector();
            repaint();
            break;
        case EDIT:
            if (stateIndex < 0)
                C = null;
            else
            {
                C = (CurveState) (State.elementAt(stateIndex));
                //slider.setValue(C.jCurr*10);
                resetSliderValue();
                if (index >= 0)
                {
                    float x = fx(evt.getX()), y = fy(evt.getY());

                    // Need to modify this for fractional editing
                    Point2D p = (Point2D) (C.V.elementAt(index));
                    Point2D[] P;

                    if (C.mu == 0) //C.jCurr == 0)
                    {
                        p.x = x; p.y = y;

                        // Coarsen the low-res vector
                        if (C.jCurr > 0)
                        {
                            P = new Point2D[C.V.size()];
                            copyFrom(C.V, P);
                            P = multires.coarsen(P, C.jCurr);
                            copyBack(C.V1, P);
                        }
                    }
                    else
                    {
                        // Determine influence on control points
                        float deltaX = x - p.x, deltaY = y - p.y;

```

```

        P = new Point2D[C.V.size()];
        copyFrom(C.V1, P);
        P = multires.editCurve(P, C.jCurr, C.mu,
                               deltaX, deltaY, index);
        copyBack(C.V1, P);

        // now, re-refine the control points
        P = new Point2D[C.V.size()];
        copyFrom(C.V1, P);
        P = multires.refine(P, C.jCurr);
        copyBack(C.V, P);
    }

    // reset global value index
    index = -1;
}
}
editCurve = false;
repaint();
break;
case ZOOM:
    // map zoom rectangle to overall screen...
    repaint();

    // now, reset zoom rectangle
    zoomX = zoomY = zoomXt = zoomYt = 0.0f;
    setOpState(EDIT);
    break;
};

mouseX = mouseY = rWidth;
}
});

addMouseListener(new MouseMotionAdapter()
{ public void mouseDragged(MouseEvent evt)
  { float x = fx(evt.getX()), y = fy(evt.getY());
    mouseX = evt.getX(); mouseY = evt.getY();

    if (showCoords)
        coordsLabel.setText("Position: " + round(x) +
                             ", " + round(y));

    switch(opState)
    {
    case DRAW:
        break;
    case SKETCH:
        Scribble.addElement(new Point2D(x, y));
        repaint();
        break;
    case EDIT:
        //if (stateIndex >= 0)
        //  C = (CurveState)(State.elementAt(stateIndex));
        if (C != null && index >= 0)
            { // Need to modify this for fractional editing
              Point2D p = (Point2D)(C.V.elementAt(index));

```

```

Point2D[] P;

if(C.mu == 0) //jCurr == 0)
{ p.x = x; p.y = y;

    // Coarsen the low-res vector
    if (C.jCurr > 0)
    { P = new Point2D[C.V.size()];
      copyFrom(C.V, P);
      P = multires.coarsen(P, C.jCurr);
      copyBack(C.Vl, P);
    }
}
else
{ // Determine influence on control points
  float deltaX = x - p.x, deltaY = y - p.y;
  P = new Point2D[C.V.size()];
  copyFrom(C.Vl, P);
  P = multires.editCurve(P, C.jCurr, C.mu,
                        deltaX, deltaY, index);

  copyBack(C.Vl, P);

  // now, re-refine the control points
  P = new Point2D[C.V.size()];
  copyFrom(C.Vl, P);
  P = multires.refine(P, C.jCurr);
  copyBack(C.V, P);
}

repaint();
}
break;
case ZOOM:
  zoomXt = x; zoomYt = y;
  repaint();
  break;
};
}

public void mouseMoved(MouseEvent evt)
{ float x = fx(evt.getX()), y = fy(evt.getY());

  if (showCoords)
    coordsLabel.setText("Position: " + round(x) +
                        ", " + round(y));
}
});

setOpaque(true);
super.setOpaque(true);
setDoubleBuffered(true);
setBackground(Color.white);
}

// Interface relay functions
public void clear()
{ Graphics g = getGraphics();

```



```

// clear all curves from system
State.removeAllElements();
Scribble.removeAllElements();
undo.removeAllElements();
redo.removeAllElements();
C = null;

// reset interface
slider.setValue(0);
setCursor(DRAW_CURSOR);
opState = EDIT;
index = stateIndex = -1;

// clear the screen
JViewport jvp = (JViewport)getParent();
if (jvp != null)
{
    g.setColor(getBackground());
    g.fillRect(jvp.getViewPosition().x, jvp.getViewPosition().y,
               jvp.getExtentSize().width,
               jvp.getExtentSize().height);
    jvp.setViewPosition(new Point(plotX(jvp), plotY(jvp)));
}

repaint();
}

public void loadCurve(Vector newState)
{
    clear();
    State = newState;
    repaint();
}

public Vector saveCurve()
{
    return State;
}

public void importTexture(Vector texture)
{
    // texture format: first 4 entries are control points,
    // the remaining are difference (wavelet) coefficients
    int nd = texture.size() - 4;

    if(nd < 0) return;

    // MODIFIED 4/29/99: take current resolution of curve and
    // apply texture from the next finer level on up to this
    // curve, expanding its length if necessary.

    // reduce the resolution of the current curve
    //Point2D[] P = new Point2D[C.V.size()];
    //
    //while(C.jCurr > 0)
    //{
    //    copyFrom(C.V, P);
    //    P = multires.coarsen(P, C.jCurr);
    //    copyBack(C.V, P);
    //    C.jCurr--;
    //}

```

```

// test if the texture can even make a dent in the current curve:
if(texture.size() > multires.numPoints(C.jCurr))
{
    // copy the texture into the current curve
    //for(int i = 0; i < 4; ++i)
    for(int i = 0; i < multires.numPoints(C.jCurr); ++i)
    { texture.removeElementAt(i);
      texture.insertElementAt(((Point2D)
                              (C.V.elementAt(i))).clone(),i);
    }

    C.V = texture;

    // bring curve back to previous resolution, or to resolution
    // of texture, whichever is greater.
    int np = 4;
    int limit = (C.np > texture.size()) ? C.np : texture.size();
    Point2D[] P = new Point2D[limit];

    while(np < limit)
    { np = multires.numPoints(++C.jCurr);
      copyFrom(C.V, P);
      P = multires.refine(P, C.jCurr);
      copyBack(C.V, P);
    }

    // lastly, create the low-res curve in the current curve
    C.np = limit;
    copyFrom(C.V, P);
    P = multires.coarsen(P, C.jCurr);
    copyBack(C.Vl, P);

    resetSliderValue();
    repaint();
}

public Vector saveTexture()
{ // saving the selected texture means
  // saving the set of control points of the
  // current CurveState

  CurveState temp = (CurveState)(C.clone());
  Point2D[] P = new Point2D[temp.V.size()];

  // reduce resolution of texture to level 0
  while(temp.jCurr > 0)
  { copyFrom(temp.V, P);
    P = multires.coarsen(P, temp.jCurr);
    copyBack(temp.V, P);
    temp.jCurr--;
  }

  return temp.V;
}

```

```

public void setCoordsLabel(JLabel coordsLabel)
{ this.coordsLabel = coordsLabel;
}

public void setOpState(int opId)
{ opState = opId;
  if (opState == EDIT)
  { setCursor(EDIT_CURSOR);
    slider.setEnabled(true);

    // Check current curve C to see if the
    // number of segments is a power of 2 and,
    // if not, insert knots until it is.

    // MODIFIED 4/29/99: added this capability
    if(C != null && C.np > 4 &&
        C.np > multires.numPoints(C.jCurr))
        C = nextLevelCurve(C);
  }
  //else      //if(opState == SKETCH)
  //{
  //}
  else if(opState == DRAW)
  { setCursor(DRAW_CURSOR);
    slider.setEnabled(false);

    // promote fractional value to next higher
    // integral level if we're adding to the curve
    if(C != null)
    { C.mu = 0;
      resetSliderValue();
    }
  }
  //else if(opState == ZOOM)
  //{
  //}
  else
  { setCursor(DRAW_CURSOR);
    slider.setEnabled(false);
  }
  repaint();
}

public void undoLastMove()
{ if (!undoStackEmpty())
  { // First, save current move to redo stack
    redo.push(C.clone());

    // Next, pop the undo stack and make this the current curve
    C = (CurveState)undo.pop();
    State.setElementAt(C, C.stateIndex);
    resetSliderValue();

    // ... and repaint
    repaint();
  }
}

```

```

}

public void redoLastMove()
{
    if (!redo.empty())
    {
        // Save current state back to undo stack
        undo.push(C.clone());

        // Next, pop the redo stack and make this the current curve
        C = (CurveState)(redo.pop());
        State.setElementAt(C, C.stateIndex);
        resetSliderValue();

        // ... and repaint
        repaint();
    }
}

public boolean undoStackEmpty()
{
    return undo.empty();
}

public boolean redoStackEmpty()
{
    return redo.empty();
}

public void setUndoLimit(int Limit)
{
    undo.setStackLimit(Limit);
    redo.setStackLimit(Limit);
}

public void setGridSpacing(float gridSpacing)
{
    this.gridSpacing = gridSpacing;
    repaint();
}

public void toggleKnots()
{
    showKnots = !showKnots;
    repaint();
}

public void togglePolygon()
{
    showPolygon = !showPolygon;
    repaint();
}

public void togglePoints()
{
    showPoints = !showPoints;
    repaint();
}

public void toggleRect()
{
    showRect = !showRect;
    repaint();
}

public void toggleGrid()
{
    showGrid = !showGrid;
}

```



```

    repaint();
}

public void toggleCoords()
{ showCoords = !showCoords;
  if (!showCoords)
    coordsLabel.setText("Position: ");
  repaint();
}

// Integer level resolution shifting
public void shiftResolution(int jNew)
{ int n = multires.numPoints(jNew);

  // guard against overlapping concurrent calls
  // from event handler
  if (shiftAllowed && C != null)
  { shiftAllowed = false;

    // no interpolation for now
    if (multires.numPoints(C.jCurr) == C.np) {
      if (jNew > C.jCurr)
      { Point2D[] P;

        if (n <= C.V.size())
          P = new Point2D[C.V.size()];
        else
          P = new Point2D[n];
        copyFrom(C.V, P);
        P = multires.refine(P, jNew);
        copyBack(C.V, P);
        C.np = n; C.jCurr = jNew;
      }
    }
    else if(jNew < C.jCurr)
    { Point2D[] P = new Point2D[C.V.size()];
      copyFrom(C.V, P);
      P = multires.coarsen(P, C.jCurr);
      copyBack(C.V, P);
      C.np = n; C.jCurr = jNew;
    }

    repaint();
  }
  //shiftAllowed = true;
}

// Fractional level resolution shifting
public void shiftResolution(float jNew)
{ int jLow = (int)jNew, jHigh = jLow + 1;
  int nLow = multires.numPoints(jLow),
    nHigh = multires.numPoints(jHigh);

  float mu = (float)((int)(jNew*10) % 10)/10;
  Point2D[] P;

  // All fractional-level shifting occurs when the level of

```

```

// resolution either goes from j.0 to j.1 or down from j.1 to
// j.0. We test for these two cases below.
if (C != null)
{ if (jNew - (C.jCurr + C.mu) > 0 && C.mu == 0.0f &&
    jLow == C.jCurr && mu >= 0.1f)
    { // save the current low-res curve to V1
      P = new Point2D[C.V.size()];
      copyFrom(C.V, P);
      copyBack(C.V1, P);

      // refine the current curve
      shiftResolution(jHigh);
    }
  else if (jNew - (C.jCurr + C.mu) < 0 && C.mu >= 0.1f &&
    jLow < C.jCurr && mu == 0.0f)
    { // reset current curve to low-res curve
      P = new Point2D[C.V1.size()];
      copyFrom(C.V1, P);
      copyBack(C.V, P);
      C.jCurr--;
      C.np = multires.numPoints(C.jCurr);

      // if jCurr > 0, coarsen current curve and save to V1
      if (C.jCurr > 0)
        { P = new Point2D[C.V.size()];
          copyFrom(C.V, P);
          P = multires.coarsen(P, C.jCurr);
          copyBack(C.V1, P);
        }
      else
        C.V1 = new Vector();
    }

  // Reset C's mu value
  if (C != null)
    C.mu = mu;

  repaint();
}
}

// internal utility functions
void initgr()
{ Dimension d = getSize();
  int maxX = d.width - 1, maxY = d.height - 1;
  pixelSize = Math.max(rWidth/maxX, rHeight/maxY);
  centerX = maxX/2; centerY = maxY/2;
}

int iX(float x){return Math.round(centerX + x/pixelSize);}
int iY(float y){return Math.round(centerY - y/pixelSize);}
float fx(int X){return (X - centerX) * pixelSize;}
float fy(int Y){return (centerY - Y) * pixelSize;}

int plotX(JViewport jvp)
{ return getSize().width/2 - jvp.getExtentSize().width/2;
}

```

```

int plotY(JViewport jvp)
{ return getSize().height/2 - jvp.getExtentSize().height/2;
}

float round(float f)
{ return (float)(Math.round(f * 1000.0f)) / 1000.0f;
}

void copyFrom(Vector v, Point2D[] P)
{ int i = 0;
  for (; i < v.size(); ++i)
    P[i] = (Point2D)(v.elementAt(i));
  for (; i < P.length; ++i)
    P[i] = new Point2D(0.0f, 0.0f);
}

void copyBack(Vector v, Point2D[] P)
{ v.removeAllElements();
  for (int i = 0; i < P.length; ++i)
    v.insertElementAt(P[i], i);
}

CurveState ScribbleToCurveState(Vector scribble, int index)
{ // Converts an input series of Point2D objects into
  // a more compact curve representation using a derivative
  // of the least-squares matching method. NOT IMPLEMENTED YET.
  return null;
}

// Adds control points to a CurveState until the number of
// segments is the next power of 2.
CurveState nextLevelCurve(CurveState C)
{ // Keep inserting knots until we get the
  // right number of segments.
  Point2D[] Q = new Point2D[C.V.size()];
  copyFrom(C.V, Q);

  while(C.np < multires.numPoints(C.jCurr+1))
  { // Build current knot sequence
    float[] knots = knotSequence(C.np - 3);

    // Build old abscissa values
    float[] oldAbscissa = computeAbscissae(knots);

    // Find next knot insertion point
    int pointIndex = findMaxInterval(C.V);
    int knotIndex = pointIndex + 1;

    float newKnot = (knots[knotIndex] + knots[knotIndex-1]) / 2;

    // Recalculate the new Greville abscissa values for
    // points at pointIndex and pointIndex - 1 and newPoint
    float[] abscissa = new float[3];

    abscissa[0] = (knots[knotIndex - 2] +
                  knots[knotIndex - 1] +

```

```

        newKnot) / 3;
abscissa[1] = (knots[knotIndex - 1] +
             newKnot +
             knots[knotIndex]) / 3;
abscissa[2] = (newKnot +
             knots[knotIndex] +
             knots[knotIndex + 1]) / 3;

// Now, recalculate the ordinate values for the x and y co-
// ordinates of the affected control points and the new point.
Point2D[] oldPoints = new Point2D[4];
for(int i = pointIndex - 2, j=0; j < 4; ++i, j++)
{
    if(i < 0)
        oldPoints[j] = (Point2D)(C.V.elementAt(i+1));
    else if(i >= C.V.size())
        oldPoints[j] = (Point2D)(C.V.elementAt(i-1));
    else
        oldPoints[j] = (Point2D)(C.V.elementAt(i));
}

Point2D[] newPoints = new Point2D[3];
for(int i = 0; i < 3; ++i)
{
    float interval;
    if(pointIndex == 1 && i == 0)
        interval = 0;
    else if(pointIndex == C.V.size() - 1 && i == 2)
        interval = 0;
    else
        interval = oldAbscissa[pointIndex + i - 1] -
                   oldAbscissa[pointIndex + i - 2];
    float xSlope = (interval == 0) ? 0 :
                   (oldPoints[i+1].x -
                    oldPoints[i].x)/interval;
    float ySlope = (interval == 0) ? 0 :
                   (oldPoints[i+1].y -
                    oldPoints[i].y) / interval;

    int oldAbsIndex = (pointIndex == C.V.size()-1 && i == 2) ?
                      pointIndex : pointIndex + i - 1;

    float newX = xSlope * abscissa[i] +
                 oldPoints[i+1].x -
                 xSlope * oldAbscissa[oldAbsIndex];
    float newY = ySlope * abscissa[i] +
                 oldPoints[i+1].y -
                 ySlope * oldAbscissa[oldAbsIndex];

    newPoints[i] = new Point2D(newX, newY);
}

// Reset the two old points in the control set and add
// the new control point.
C.V.setElementAt(newPoints[0], pointIndex - 1);
C.V.setElementAt(newPoints[2], pointIndex);

C.V.insertElementAt(newPoints[1], pointIndex);

```



```

        // Increment point counter
        C.np++;
    }

    Q = new Point2D[C.V.size()];
    copyFrom(C.V, Q);

    // Prepare remaining components of CurveState
    C.jCurr++;
    C.nextNPsize = multires.numPoints(C.jCurr + 1);

    Point2D[] P = new Point2D[C.V.size()];
    copyFrom(C.V, P);
    P = multires.coarsen(P, C.jCurr);
    copyBack(C.V1, P);

    resetSliderValue();

    return C;
}

// Returns a uniform knot sequence of size numSegments.
float[] knotSequence(int numSegments)
{
    int n = numSegments + 5;
    float[] knots = new float[n];

    for(int i = 0; i < n; ++i)
        knots[i] = (float)((float)(i)/n); //numSegments);

    return knots;
}

// Returns the sequence of Greville abscissae related to the
// input knot sequence for an endpoint-interpolating B-spline.
float[] computeAbscissae(float[] knots)
{
    float[] abscissae = new float[knots.length - 2];

    for(int i = 0; i < knots.length - 2; ++i)
        abscissae[i] = (knots[i] + knots[i+1] + knots[i+2]) / 3;

    return abscissae;
}

// Finds the largest distance between adjacent pairs
// of x,y points in P and returns the upper index.
int findMaxInterval(Vector V)
{
    Point2D[] P = new Point2D[V.size()];
    copyFrom(V, P);

    float max = distance(P[0], P[1]);
    int maxIndex = 1;

    for(int i = 2; i < V.size(); ++i)
    {
        float newMax = distance(P[i-1], P[i]);

        if(max < newMax)
            max = newMax;
    }
}

```

```

        maxIndex = i;
    }
}

return maxIndex;
}

float distance(Point2D p, Point2D q)
{ return (float)Math.sqrt((q.x - p.x)*(q.x - p.x) +
    (q.y - p.y)*(q.y - p.y));
}

boolean onPoint(Point2D p, float x, float y)
{ return Math.abs(iX(p.x) - iX(x)) < 2 &&
    Math.abs(iY(p.y) - iY(y)) < 2;
}

void resetSliderValue()
{ if (C != null)
    { if (C.mu == 0)
        slider.setValue(C.jCurr * 10);
      else
        slider.setValue((int)((C.jCurr-1 + C.mu) * 10));
    }
    if (opState != EDIT)
        slider.setEnabled(false);
}

void drawGrid(Graphics g)
{ JViewport jvp = (JViewport)getParent();
  if (jvp != null)
    { Point p = jvp.getViewPosition();
      Dimension d = jvp.getExtentSize();

      float gx = (float)Math.ceil(fx(p.x)/gridSpacing) *
          gridSpacing,
          gy = (float)Math.ceil(fy(p.y)/gridSpacing) *
          gridSpacing;

      float gxMax = fx(p.x + d.width),
          gyMin = fy(p.y + d.height);

      g.setColor(Color.lightGray);
      for (float f = gx; f < gxMax; f += gridSpacing)
          for (float h = gy; h > gyMin; h -= gridSpacing)
              { g.fillOval(iX(f), iY(h), 2, 2);
                }

      g.setColor(Color.black);
    }
}

// Integral-level resolution version
void bspline(Graphics g, Point2D[] P, int n, int thisIndex)
{ int m = 50; //, n = np; //P.length;
  float xA, yA, xB, yB, xC, yC, xD, yD,
    a0, a1, a2, a3, b0, b1, b2, b3, x = 0, y = 0, x0, y0;
  boolean first = true;

```

```

for (int i = 1; i < n-2; ++i)
{  xA = P[i-1].x; xB = P[i].x; xC = P[i+1].x; xD = P[i+2].x;
  yA = P[i-1].y; yB = P[i].y; yC = P[i+1].y; yD = P[i+2].y;
  a3 = (-xA+3*(xB-xC)+xD)/6;  b3 = (-yA+3*(yB-yC)+yD)/6;
  a2 = (xA-2*xB+xC)/2;        b2 = (yA-2*yB+yC)/2;
  a1 = (xC-xA)/2;             b1 = (yC-yA)/2;
  a0 = (xA+4*xB+xC)/6;        b0 = (yA+4*yB+yC)/6;
  for (int j = 0; j <= m; ++j)
  {  x0 = x;
     y0 = y;
     float t = (float)j/(float)m;
     x = ((a3*t+a2)*t+a1)*t+a0;
     y = ((b3*t+b2)*t+b1)*t+b0;

     if (opState == EDIT &&
         onPoint(new Point2D(x, y), mouseX, mouseY))
         stateIndex = thisIndex;

     if (showKnots)
     {  if(j == 0 || j == m)
        g.fillOval(iX(x)-2, iY(y)-2, 4, 4);
      }

     if (first) first = false;
     else
       g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
  }
}
}

// Fractional-level resolution version
void bspline(Graphics g, Point2D[] P, int n, Point2D[] PL, float mu,
             int thisIndex)
{ // If the curve is of integral resolution,
  // call the other B-spline plot.
  if (mu == 0.0f)
    bspline(g, P, n, thisIndex);

  // Otherwise, plot the curve that exists at some fractional
  // level mu between the higher resolution curve P and the
  // lower resolution curve PL.
  else
  {  int m = 25; //50;
     float xA, yA, xB, yB, xC, yC, xD, yD, xE, yE,
           a0, a1, a2, a3, b0, b1, b2, b3,
           c0, c1, c2, c3, d0, d1, d2, d3, x = 0, y = 0, x0, y0;
     float xAL, yAL, xBL, yBL, xCL, yCL, xDL, yDL,
           a0L, a1L, a2L, a3L, b0L, b1L, b2L, b3L;
     boolean first = true;

     for (int i = 1, iL = 1; i < n-3; iL++, i+=2)
     { // Save first 5 control points of high-res curve
       xA = P[i-1].x; xB = P[i].x; xC = P[i+1].x; xD = P[i+2].x;
       yA = P[i-1].y; yB = P[i].y; yC = P[i+1].y; yD = P[i+2].y;
       xE = P[i+3].x; yE = P[i+3].y;

       // Save first 4 control points of low-res curve

```

```

xAL=PL[iL-1].x; xBL=PL[iL].x;
xCL=PL[iL+1].x; xDL=PL[iL+2].x;
yAL=PL[iL-1].y; yBL=PL[iL].y;
yCL=PL[iL+1].y; yDL=PL[iL+2].y;

// Save coefficients of first part of high-res curve...
a3 = (-xA+3*(xB-xC)+xD)/6; b3 = (-yA+3*(yB-yC)+yD)/6;
a2 = (xA-2*xB+xC)/2;      b2 = (yA-2*yB+yC)/2;
a1 = (xC-xA)/2;          b1 = (yC-yA)/2;
a0 = (xA+4*xB+xC)/6;     b0 = (yA+4*yB+yC)/6;

// ...as well as the second part of the high-res curve...
c3 = (-xB+3*(xC-xD)+xE)/6; d3 = (-yB+3*(yC-yD)+yE)/6;
c2 = (xB-2*xC+xD)/2;     d2 = (yB-2*yC+yD)/2;
c1 = (xD-xB)/2;         d1 = (yD-yB)/2;
c0 = (xB+4*xC+xD)/6;     d0 = (yB+4*yC+yD)/6;

// ...and lastly the coefficients of the low-res curve.
a3L = (-xAL+3*(xBL-xCL)+xDL)/6;
b3L = (-yAL+3*(yBL-yCL)+yDL)/6;
a2L = (xAL-2*xBL+xCL)/2;      b2L = (yAL-2*yBL+yCL)/2;
a1L = (xCL-xAL)/2;           b1L = (yCL-yAL)/2;
a0L = (xAL+4*xBL+xCL)/6;     b0L = (yAL+4*yBL+yCL)/6;

// Interpolate first half of curve...
for (int j = 0; j <= m; ++j)
{
    x0 = x;
    y0 = y;
    float t = (float)j/(float)m;
    float tL = (float)j/(float)m*0.5f;
    x = mu * (((a3*t+a2)*t+a1)*t+a0) +
        (1.0f - mu)*(((a3L*tL+a2L)*tL+a1L)*tL+a0L);
    y = mu * (((b3*t+b2)*t+b1)*t+b0) +
        (1.0f - mu)*(((b3L*tL+b2L)*tL+b1L)*tL+b0L);

    if (opState == EDIT &&
        onPoint(new Point2D(x, y), mouseX, mouseY))
        stateIndex = thisIndex;

    if (showKnots)
    {
        if (j == 0 || j == m)
            g.fillOval(iX(x)-2, iY(y)-2, 4, 4);
    }

    if (first) first = false;
    else
        g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
}

// ...and then the second half of the curve
for (int j = 0; j <= m; ++j)
{
    x0 = x;
    y0 = y;
    float t = (float)j/(float)m;
    float tL = (float)j/(float)m*0.5f + 0.5f;
    x = mu * ((c3*t+c2)*t+c1)*t+c0) +
        (1.0f - mu)*(((a3L*tL+a2L)*tL+a1L)*tL+a0L);

```

```

        y = mu * (((d3*t+d2)*t+d1)*t+d0) +
            (1.0f - mu)*(((b3L*tL+b2L)*tL+b1L)*tL+b0L);

        if (opState == EDIT &&
            onPoint(new Point2D(x, y), mouseX, mouseY))
            stateIndex = thisIndex;

        if (showKnots)
        {   if(j == 0 || j == m)
            g.fillOval(iX(x)-2, iY(y)-2, 4, 4);
        }

        //if (first) first = false;
        //else
            g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
    }
}
}

public void paintComponent(Graphics g)
{   super.paintComponent(g);

    if (paintFirstTime)
    {   paintFirstTime = false;
        JViewport jvp = (JViewport)getParent();
        if (jvp != null)
            jvp.setViewPosition(new Point(plotX(jvp), plotY(jvp)));
    }

    initgr();
    int left = iX(-rWidth/2), right = iX(rWidth/2),
        bottom = iY(-rHeight/2), top = iY(rHeight/2);

    // Draw grid and zoom rectangle
    if (showGrid)
        drawGrid(g);
    if (opState == ZOOM && zoomX != zoomXt && zoomY != zoomYt)
    {   int zLeft = iX(zoomX), zRight = iX(zoomXt),
        zBottom = iY(zoomYt), zTop = iY(zoomY);
        if (zRight >= zLeft)
        {   if (zBottom >= zTop)
            g.drawRect(zLeft, zTop, zRight - zLeft, zBottom - zTop);
            else
            g.drawRect(zLeft, zBottom, zRight-zLeft, zTop-zBottom);
        }
        else
        {   if (zBottom >= zTop)
            g.drawRect(zRight, zTop, zLeft-zRight, zBottom - zTop);
            else
            g.drawRect(zRight, zBottom, zLeft-zRight, zTop-zBottom);
        }
    }

    Point2D[] P;
    Point2D[] PL;
}

```



```

// Draw the scribble
if (Scribble != null)
{ P = new Point2D[Scribble.size()];
  Scribble.copyInto(P);
  for (int k = 0; k < Scribble.size(); ++k)
  { if (k > 0)
      g.drawLine(iX(P[k-1].x), iY(P[k-1].y),
                 iX(P[k].x), iY(P[k].y));
  }
}

// Repeat for each curve in State:
for (int k = 0; k < State.size(); ++k)
{ CurveState cTemp = (CurveState)(State.elementAt(k));
  P = new Point2D[cTemp.V.size()];
  PL = new Point2D[cTemp.Vl.size()];
  cTemp.V.copyInto(P);
  cTemp.Vl.copyInto(PL);
  g.setColor(Color.gray);
  for (int i = 0; i < cTemp.np; i++)
  { // if we're moving a control point, find
    // which curve it belongs to
    if (opState == EDIT &&
        onPoint(P[i], mouseX, mouseY))
        stateIndex = k;

    // Show tiny rectangle around point:
    if (showPoints)
    { if(k == stateIndex)
        g.fillRect(iX(P[i].x)-2, iY(P[i].y)-2, 4, 4);
      else
        g.drawRect(iX(P[i].x)-2, iY(P[i].y)-2, 4, 4);
    }
    if (i > 0 && showPolygon)
      // Draw line P[i-1]P[i]:
      g.drawLine(iX(P[i-1].x), iY(P[i-1].y),
                 iX(P[i].x), iY(P[i].y));
  }
  g.setColor(Color.black);
  if (cTemp.np >= 4) bspline(g, P, cTemp.np, PL, cTemp.mu, k);
}

shiftAllowed = true;
}
}

```

```

// File: CurveState.java
// Author: Stephen Alberg
// Uses: Point2D class
//
// This file contains the class definition for a CurveState object.
// The CurveState class performs the duty of registering the data
// constituting each curve presently drawn in a multiresolution curve
// editing system. It may also be used to save the last edit
// information onto an undo stack.

import java.io.*;
import java.util.*;

public class CurveState implements Cloneable, Serializable
{ public Vector V;          // Records the set of control points
                                // and any difference coefficients
  public Vector Vl;        // The set of control points at resolution
                                // jCurr. Used for fractional curve editing
  public int np;          // The number of control points
  public int stateIndex;  // The "id" of the curve in the environment
  public int jCurr;       // Current level of resolution
  public int nextNPsize;  // Number of control points in next res.
  public float mu;        // Fractional offset from jCurr, if any

  CurveState(Vector V, Vector Vl, int np, int stateIndex, int jCurr,
              int nextNPsize, float mu)
  { this.V = V; this.Vl = Vl; this.np = np;
    this.stateIndex = stateIndex;
    this.jCurr = jCurr; this.nextNPsize = nextNPsize;
    this.mu = mu;
  }

  CurveState(int stateIndex)
  { V = new Vector(); Vl = new Vector(); np = 0;
    this.stateIndex = stateIndex;
    jCurr = -1; nextNPsize = 4; mu = 0.0f;
  }

  public synchronized Object clone() // overrides Object
  { return new CurveState((Vector)(V.clone()), (Vector)(Vl.clone()),
                          np, stateIndex, jCurr,
                          nextNPsize, mu);
  }
}

```

```
// Point2D.java: Class for points in logical coordinates.

import java.io.*;

public class Point2D implements Cloneable, Serializable
{   float x, y;
    Point2D(float x, float y){this.x = x; this.y = y;}

    public synchronized Object clone() // overrides Object
    {   return new Point2D(x, y);
    }
}
```

```

// File: UndoStack.java
// Author: Stephen Alberg
//
// This file contains the class definition of an UndoStack object. An
// UndoStack extends the basic properties of the Java Stack object by
// by providing more direct manipulation and limit checking of the
// stack size.

import java.util.Stack;

public class UndoStack extends Stack
{
    int stackLimit;

    UndoStack()
    {
        super(); // instantiate the parent object
        stackLimit = 5; // default stack size
    }

    UndoStack(int stackLimit)
    {
        super();
        this.stackLimit = stackLimit;
    }

    // Overrides parent method push()
    public Object push(Object item)
    {
        Object o = super.push(item);

        removeExtraObjects();
        return o;
    }

    public void setStackLimit(int newLimit)
    {
        stackLimit = newLimit;
        removeExtraObjects();
    }

    public int getStackLimit()
    {
        return stackLimit;
    }

    void removeExtraObjects()
    {
        // The Stack object evidently appends the last insertion
        // to the end of the parent Vector, rather than inserting
        // at the beginning. Therefore, to trim the undoStack,
        // keep removing elements at index 0.
        while (size() > stackLimit)
            super.removeElementAt(0);
    }
}

```

## BIBLIOGRAPHY

- Ammeraal, Leen. *Computer Graphics for Java Programmers*. Chichester: John Wiley and Sons, 1998.
- Angel, Edward. *Interactive Computer Graphics: a Top-Down Approach with OpenGL*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1997.
- Banks, Michael J. and Elaine Cohen. "Realtime spline curves from interactively sketched data." *Computer Graphics*. Vol. 24, no. 2: pp. 99-107, 1990.
- Bartels, Richard H., John C. Beatty, Brian A Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Los Altos, California: Morgan Kaufman Publishers, Inc., 1987.
- Chui, Charles K.. *An Introduction to Wavelets*. San Diego: Academic Press, 1992.
- Farin, Gerald. *Curves and Surfaces for Computer-Aided Geometric Design*. Boston: Academic Press, 1988.
- Finkelstein, Adam and David H Salesin. "Multiresolution Curves". *Proceedings of the Special Interest Group on Computer Graphics (SIGGRAPH) 1994*, pp. 261-268. Association for Computing Machinery, New York, 1994.
- Foley, James D., Andries van Dam, Steven K. Feiner, John F. Hughes. *Computer Graphics: Principles and Practice, Second Edition*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1996.
- Forsey, David and Richard H. Bartels. "Hierarchical B-spline Refinement". *Computer Graphics*. Vol. 22, no. 4: pp. 205-212, 1988.
- Hubbard, Barbara Burke. *The World According to Wavelets*. Wellesley, Massachusetts: A. K. Peters, Ltd., 1996.
- Kaiser, Gerald. *A Friendly Guide to Wavelets*. Boston, Massachusetts: Birkhauser, 1994.
- Mallat, Stephane. "A Theory for Multiresolution Signal Decomposition: the Wavelet Representation." *Institute of Electrical and Electronic Engineers Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, no. 7: pp. 674-693. July, 1989.
- OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis. *OpenGL Programming Guide, Second Edition*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1997.



Press, William H., Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge: Cambridge University Press, 1992.

Salisbury, Michael P., Sean E. Anderson, Ronen Barzel, David H. Salesin. "Interactive Pen-and-Ink Illustration". *Proceedings of the Special Interest Group on Computer Graphics (SIGGRAPH) 1994*, pp. 101-108. Association for Computing Machinery, New York, 1994.

Stollnitz, Eric J., Tony D. DeRose, David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. San Francisco: Morgan Kaufman Publishers, Inc., 1996.