Adaptive Forward Differencing for Rendering Curves and Surfaces

Sheue-Ling Lien, Michael Shantz and Vaughan Pratt

Sun Microsystems, Inc.
2500 Garcia Avenue
Mountain View, CA 94043

Abstract

An adaptive forward differencing algorithm is presented for rapid rendering of cubic curves and bicubic surfaces. This method adjusts the forward difference step size so that approximately one pixel is generated along an ordinary or rational cubic curve for each forward difference step. The adjustment involves a simple linear transformation on the coefficients of the curve which can be accomplished with shifts and adds. This technique combines the advantages of traditional forward differencing and adaptive subdivision. A hardware implementation approach is described including the adaptive control of a forward difference engine. Surfaces are rendered by drawing many curves spaced closely enough together so that no pixels are left unpainted. A simple curve anti-aliasing algorithm is also presented in this paper. Anti-aliasing cubic curves is supported via tangent vector output at each forward difference step. The adaptive forward differencing algorithm is also suitable for software implementation.

CR Categories and Subject Descriptors: I.3.5 **[Computer Graphics]**: Computational Geometry and Object Modelling - Curve, surface, solid, and object representations; Geometric algorithms, and systems; I.3.3 **[Computer Graphics]**: Picture/Image Generation - Display algorithms; I.3.7 **[Computer Graphics]**: Three-dimensional Graphics and Realism - Color, shading, shadowing, and texture.

Additional Key Words and Phrases: image synthesis, adaptive forward differencing, parametric curves and surfaces.

 ACM-0-89791-227-6/87/007/0111 $00.75

Introduction

Parametric curves and curved surfaces are a common form of surface and object representation. In particular, non-uniform rational b-splines have gained popularity for mechanical CAD applications. Since high speed hardware capable of rendering vectors and polygons is widely available, high speed curve and surface rendering is usually done by subdividing and rendering them as straight lines or planar polygons. For conics, non-parametric, incremental solutions of the implicit equations[7, 8, 3, 1] are well known and a few hardware curve generators have been built. Less progress has been made on hardware techniques for rendering higher order curves and surfaces. Research has focused largely on subdivision methods for rendering and modelling.[2, 6] Recursive subdivision for curve and surface rendering is expensive to implement in hardware due to the high speed stack memory requirements and the fact that frame buffer memory access is easier to optimize if the pixels are being written to adjacent addresses.

Lane and others[5] developed scan line methods for rendering bicubic patches. They used Newton iteration to compute the intersections of the patch with the plane of the scanline. These approaches were not intended for, nor are they simple enough for hardware implementation.

Our adaptive forward difference (AFD) technique is an extension of well known[4] ordinary forward differencing and is related to the adaptive subdivision methods in that it adjusts the step size to the next pixel by transforming the equation of the curve to an identical curve with different parameterization. AFD differs from recursive subdivision or traditional forward differencing by generating points sequentially along the curve while adjusting the parameterization to give pixel sized steps. AFD allows a surprisingly simple hardware implementation, and is compatible with frame buffer memory interleaving for high performance.

This paper develops the theory of adaptive forward differencing and covers several related aspects and problem areas.

1) Reparameterization of cubic or rational cubic curves
2) Drawing surfaces by spacing curves δs apart
3) Generating anti-aliased curves
4) Trimming and image mapping on patches

With special purpose hardware for rendering these curves and surfaces directly, the usual subdivision overhead is reduced, and the appearance of the rendered objects is more accurate. The method lends itself to hardware fast shading techniques by functional approximations of the unit normal function over a patch.[9]

Principles

The method of adaptive forward differencing unifies the processes of recursive subdivision and forward differencing. In this section we present the principles underlying the method. The key insights are that these processes are both instances of linear substitution, and that efficiency is optimized by a choice of basis appropriate to the mix of substitutions.

We consider curves and surfaces in a space S, taken for the sake of illustration to be R^4 (homogeneous coordinates x,y,z,w). A *parametric object* in S is a function $f:X\rightarrow S$ where X is a set constituting the *parameter space*. The object is a *curve*, *segment*, *surface*, or *patch* when X is respectively the set R of reals, the real interval $[0,1]$, the real plane R^2, or the unit square $[0,1]^2$. We take s and t for the parameters, making f either $f(t)$ or $f(s,t)$.

A *linear substitution* transforms $f(t)$ into $f(at+b)$ and $f(s,t)$ into $f(as+b,ct+d)$, expressible as the composition of f with a linear or bilinear function respectively. The geometric effect of linear substitution is to translate and scale a segment or patch within the curve or surface containing it. Any segment of a curve can be mapped to any other segment of the same curve by some linear substitution, and likewise for patches.

Let us denote by L the linear substitution $t/2$ and by R the linear substitution $(t+1)/2$. Then L and R act on a segment C to yield the "left" and "right" halves LC and RC of C. These are the transformations associated with recursive subdivision; they may be applied recursively to subdivide a curve segment into quarters LLC,LRC,RLC,RRC (Figure 1(a)), eighths, etc.

Let us denote by E the linear substitution $t+1$. Then E acts on a segment C to yield its "right neighbor" EC. One use for E is as the forward difference operator. To render a long segment C, start with a very small initial segment D of C (e.g. $D=LL\cdots LC$) and generate the remaining (also small) segments of C by $ED,EED,EEED,\cdots$. This process is usually called forward differencing.

Another use for E is as a substitute for R in recursive subdivision: we may represent R as EL, as illustrated by the top half of Figure 1(b). However, rather than computing LC and RC separately we can compute LC once and then apply E to LC to get the right half, allowing us to discard C after applying L to it and so avoiding a "stack pop" when the time comes to apply R. The lower half of Figure 1(b) shows that this can be extended down another layer of recursion: we can get to RLC, LRC, and RRC by starting from LLC and repeatedly applying E, thanks to the additional identity $ER=LE$ (i.e. $EEL=LE$) which allows E to make the jump from RLC to LRC. This ability of E to run across the whole tree holds at any depth. At sufficient depth the method turns into ordinary forward differencing as per the previous paragraph.

A disadvantage of forward differencing is that it may not traverse C with uniform velocity. Recursive subdivision avoids this difficulty by stopping at different depths in different parts of the recursion tree. We may transfer this advantage of recursive subdivision to forward differencing by inserting an occasional L or L^{-1} (the substitution $2t$) into the stream of E's whenever the velocity is too great or too small respectively. This has the effect of changing our level in the recursion tree as we forward-difference across it. We call this technique *adaptive forward differencing*.

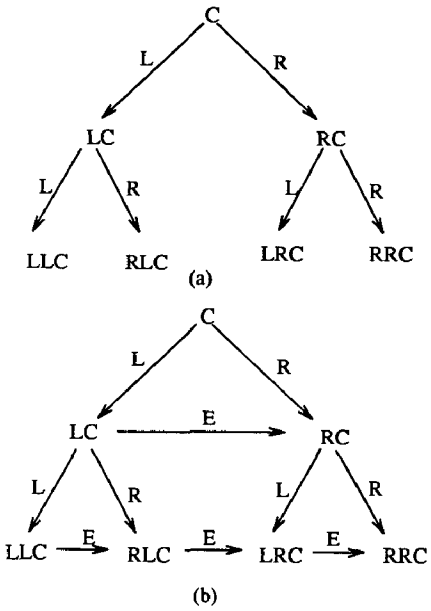

Figure 1. Relationship of linear substitutions L, R, and E.

In order to implement the above we require concrete representations for C, L, E, etc. We do this in the usual way: independently for each dimension of S take C to be a polynomial in t (and s), regard the polynomial as a point in a vector space of dimension one more than its degree, regard linear substitutions as a particular kind of linear transformation of this space, and perform the transformations in an appropriate basis for the space. One key property of linear substitution is that it does not increase polynomial degree, the other is that its action on a polynomial viewed as a vector is indeed that of a linear transformation.

While any basis will do, certain bases favor certain transformations. For example the total number of 1's in the binary representation of a particular transformation may be quite small in a particular basis, permitting the transformation to be carried out with just a few shifts and adds. Catmull [2] gives a basis for which L and R can be cheaply computed with only three adds and four shifts.

$$L_c = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/4 & 0 & 0 \\ 1/2 & -1/8 & 1/2 & -1/8 \\ 0 & 1/8 & 0 & 1/8 \end{bmatrix}$$

$$R_c = \begin{bmatrix} 1/2 & -1/8 & 1/2 & -1/8 \\ 0 & 1/8 & 0 & 1/8 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/4 \end{bmatrix}$$

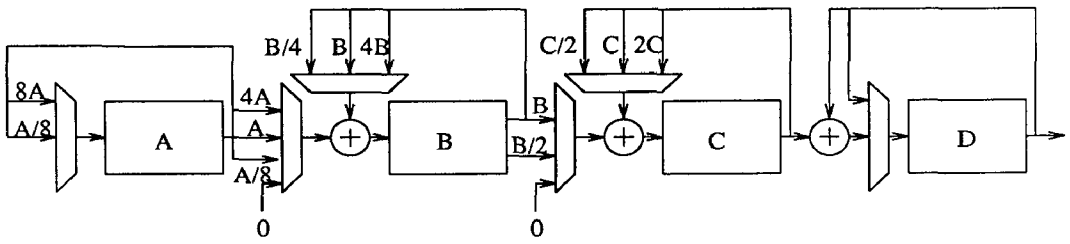

Figure 2. Block diagram of an AFD unit.

However, for forward differencing neither Catmull's basis for L and R nor any of the other bases usually considered for recursive subdivision are particularly well suited to the matrix representation of E. The best basis for E is the forward difference basis which allows parallel additions suitable for a pipeline implementation.

Adaptive Forward Difference Algorithm

For adaptive forward differencing we require a basis that works well with L and L^{-1}, especially with E, on the ground that E occurs significantly more often than L in practice. The following set is the forward difference basis which is considered to be the most appropriate.

$$B_3 = \frac{1}{6}(t^3-3t^2+2t) = \frac{1}{6}t(t-1)(t-2)$$

$$B_2 = \frac{1}{2}(t^2-t) = \frac{1}{2}t(t-1)$$

$$B_1 = t$$

$$B_0 = 1$$

The E matrix of this basis requires only three adds which can be done in parallel.

$$E = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The L and L^{-1} matrices can be implemented with simple shifts and adds.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & -1/8 & 1/16 \\ 0 & 0 & 1/4 & -1/8 \\ 0 & 0 & 0 & 1/8 \end{bmatrix} \quad L^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

This algorithm is implemented in hardware called an AFD unit. An AFD unit is a third order digital differential analyzer which implements an adaptive forward difference solution to a parametric cubic function of t. The parameter t varies from 0 to 1 along the curve. The dt step size for t is adaptively adjusted so that the curve steps along in approximately one pixel steps in screen coordinates. Figure 2 shows a block diagram of an AFD unit. †

† Sun Microsystems, Inc. is pursuing patent protection in the United States and abroad on the technology described in this paper.

Four AFDUs can be used to generate the x,y,z,w values of the pixels along a cubic curve. Figure 3 shows the 4 required AFDUs and the divide by w circuit necessary for rendering rational curves. The filter unit is the controller for the adaptive step size, and performs other functions.

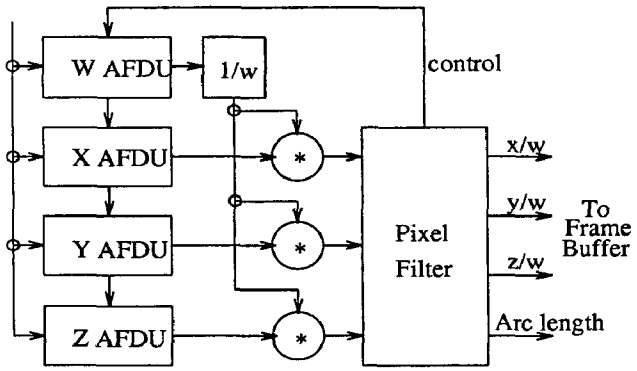

Figure 3. Block diagram of the AFD hardware. Each AFDU computes a 3rd order parametric function.

1. Reparameterization

A parametric cubic function f(t) can be represented in forward difference basis as

$$f = aB_3(t)+bB_2(t)+cB_1(t)+dB_0(t)$$

A cubic curve is defined by four cubic functions x(t), y(t), z(t), and w(t), each implemented by a separate AFD unit.

$$x(t) = a_xB_3+b_xB_2+c_xB_1+d_xB_0$$

$$y(t) = a_yB_3+b_yB_2+c_yB_1+d_yB_0$$

$$z(t) = a_zB_3+b_zB_2+c_zB_1+d_zB_0$$

$$w(t) = a_wB_3+b_wB_2+c_wB_1+d_wB_0$$

The coefficients a, b, c, and d are loaded into the 4 coefficient registers of each AFD unit. At each clock event the parameter t increases by dt and the four AFDUs generate the coordinates of one pixel.

If the x,y address step, corresponding to the dt step, is more than one pixel, dt is divided by two (adjusted down) so that each clock generates approximately one pixel along the curve. If the x,y address step is less than 1/2 pixel then dt is doubled (adjusted up) to increase the change in x,y coordinates.

To reduce dt by half, we transform the cubic functions x(t), y(t), z(t), w(t) by applying the L matrix:

$$x'(t) = x(\frac{t}{2}) = a'_x B_3 + b'_x B_2 + c'_x B_1 + d'_x B_0$$

$$y'(t) = y(\frac{t}{2}) = a'_y B_3 + b'_y B_2 + c'_y B_1 + d'_y B_0$$

$$z'(t) = z(\frac{t}{2})$$

$$w'(t) = w(\frac{t}{2})$$

The coefficients of the two sets of cubic functions are related by

$$a' = \frac{1}{8}a$$

$$b' = \frac{1}{4}b - \frac{1}{8}a$$

$$c' = \frac{1}{2}c - \frac{1}{8}b + \frac{1}{16}a$$

$$d' = d$$

To double dt, we transform the cubic functions by applying the L^{-1} matrix:

$$x'(t) = x(2t)$$

$$y'(t) = y(2t)$$

$$z'(t) = z(2t)$$

$$w'(t) = w(2t)$$

Here the coefficient transformation is

$$a' = 8a$$

$$b' = 4b + 4a$$

$$c' = 2c + b$$

$$d' = d$$

If the step size is correct then we apply the E matrix.

$$x'(t) = x(t+1)$$

$$y'(t) = y(t+1)$$

$$z'(t) = z(t+1)$$

$$w'(t) = w(t+1)$$

The AFD units in this case generate a new pixel and advance to the next pixel with the corresponding coefficients transformed by

$$a' = a$$

$$b' = b + a$$

$$c' = c + b$$

$$d' = d + c$$

The adaptive forward differencing mechanism is illustrated below.

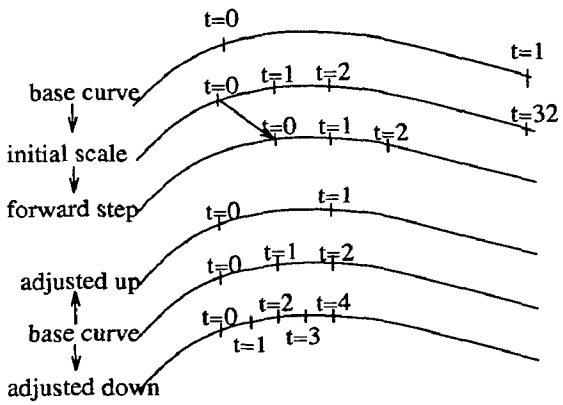

Figure 4 Operations of "adjust up", "adjust down", and "forward step".

We have measured the percentage of steps requiring an adjust up or adjust down using the Utah teapot at various scale factors. Drawing the cubic curves comprising a wire mesh on the bicubic patches making up the surface involved 73,000 forward steps, and 600 adjustment steps. The overhead for the adaptive nature of the forward difference scheme is therefore quite small. It increases when the curves being drawn have large accelerations.

2. Initial Setup

To render a cubic curve C we first convert to the forward difference basis. We then start with a small initial segment D of the curve by applying $L^n = L.L.....L$ to the curve C. The initial scale down is not really required. However, if not done, the adaptive mechanism may adjust down many times until the pixel step size is approximately one pixel before it starts rendering. In practice the parameterization is scaled down before loading into the AFD units to be within the hardware register precision. The AFDUs adjust from there.

3. Pixel filtering

The pixel filter performs five functions. 1) It compares the current pixel coordinates with the previous pixel coordinates generated by the AFD units and tells the AFD units whether they should adjust up, adjust down, or step forward to the next pixel. 2) The pixel filter also detects and replaces "elbow" sequences of the form x,y to x,y+1 to x+1,y+1 with a diagonal move x,y to x+1,y+1. This is done to improve the appearance of generated curves. 3) It also generates arc length along the curve generated by the AFD units. It adds 1 to the arc length if the curve steps either horizontally or vertically and adds 1.414 if the curve steps diagonally. The output arc length is used to address the pattern memory for mapping texture along curves. 4) The filter unit performs clipping on t, x, y, and z. t is clipped between a tmin register and a tmax register to assist in rendering trimmed patches. x, y, and z are clipped to their respective min and max register values. 5) The filter generates the instantaneous tangent and normal vectors for the purpose of anti-aliasing curves.

The pixel filter thus acts as the controller for the AFD units and also computes arc length, and antialias weights.

4. Rendering conics and rational cubics

One of the AFD units generates the homogeneous coordinate w as a parametric cubic function of t. For rational cubic curves x, y, and z must be divided by w at each point. This is accomplished by using a reciprocal unit which computes a truncated Taylor series approximation of 1/w.

The reciprocal 1/w is computed as follows which can be easily implemented with look-up tables, adders and mutipliers.

$$\frac{1}{w} = \frac{1}{w_0} - \frac{\delta}{w_0^2}$$

The following example shows how to set up AFDUs to draw an ellipse with radius r_x, r_y centered at $<x_0, y_0>$ and rotated by an angle θ. A half ellipse with radius r_x, r_y can be defined in parametric form as

$$x_1(t) = r_x \frac{t(1-t)}{t^2-t+0.5}$$

$$y_1(t) = r_y \frac{0.5-t}{t^2-t+0.5}$$

We can get the other half of the ellipse by mirroring the image. By rotating the ellipse by an angle θ and then translating it to $<x_0, y_0>$, we get a set of cubic functions which describe an ellipse with radius r_x, r_y centered at x_0, y_0 and rotated by an angle θ:

$$\begin{aligned} x(t) &= r_x t(1-t)\cos\theta + r_y(0.5-t)\sin\theta + x_0(t^2-t+0.5) \\ &= (x_0 - r_x\cos\theta)t^2 + (r_x\cos\theta - r_y\sin\theta - x_0)t \\ &\quad + 0.5(r_y\sin\theta + x_0) \\ y(t) &= -r_x t(1-t)\sin\theta + r_y(0.5-t)\cos\theta + y_0(t^2-t+0.5) \\ &= (y_0 + r_x\sin\theta)t^2 - (r_x\sin\theta + r_y\cos\theta - y_0)t \\ &\quad + 0.5(r_y\cos\theta + y_0) \\ w(t) &= t^2-t+0.5 \end{aligned}$$

By converting the above cubic functions to DDA basis, we get a set of coefficients

$$\begin{aligned} a_x &= 0 \\ b_x &= 2(x_0 - r_x\cos\theta) \\ c_x &= (x_0 - r_x\cos\theta) + (r_x\cos\theta - r_y\sin\theta - x_0) \\ d_x &= 0.5(r_y\sin\theta + x_0) \\ a_y &= 0 \\ b_y &= 2(y_0 + r_x\sin\theta) \\ c_y &= (y_0 + r_x\sin\theta) - (r_x\sin\theta + r_y\cos\theta - y_0) \\ d_y &= 0.5(r_y\cos\theta + y_0) \\ a_w &= 0.0 \\ b_w &= 2.0 \\ c_w &= 0.0 \\ d_w &= 0.5 \end{aligned}$$

We can set up the AFD units with the above coefficients for drawing the ellipse (see Figure 6).

5. Anti-aliasing cubic curves

AFD gives a simple means of generating the instantaneous tangent vector $<t_x, t_y>$ along the curve by simply subtracting the last point from the previous one. The instantaneous tangent vector gives an indication of whether the current pixel is in an x_major ($t_x > t_y$) or y_major ($t_x < t_y$) slope. An approximate distance of the current pixel away from the center curve is computed from this tangent and the fractional portion of the pixel addresses as follows. Here α is the ratio of variation of intensity, and α is used to blend the curve color and the background color. This is a rather crude approximation but gives surprisingly improved curve appearance. Figure 5 shows the result of this curve anti-aliasing method drawn with the software simulation.

If the tangent vector indicates x_major, we compute

$$\alpha = (f_y - 0.5) + \frac{t_y}{t_x}(f_x - 0.5)$$

where $<t_x, t_y>$ is the tangent and f_x and f_y are the fractional portion of the pixel x and y address. If α is positive, then the intensity of pixel <x,y> is blended by (1.0–α), and pixel <x,y+1> by α. In case of a negative α, pixel <x,y> is blended by (1.0+α), and pixel <x,y-1> by $-\alpha$. For y_major, α is

$$\alpha = (f_x - 0.5) + \frac{t_x}{t_y}(f_y - 0.5)$$

In this case, pixel <x,y> is blended by (1.0–α), and pixel <x+1,y> by α if α is positive; otherwise pixel <x,y> by (1.0+α) and pixel <x-1,y> by $-\alpha$.

One advantage of this anti-aliasing scheme is that it applies as well to both nonrational and rational curves. Figure 6 shows a set of anti-aliased rational curves rendered with this scheme.

Shading Bicubic Patches

The AFD technique can be used to render shaded, curved, trimmed patches, generate anti-aliased curves, and map texture and imagery onto curves and surfaces as a function of either arc-length or parameter.

Shading and image mapping onto bicubic or rational bicubic surface patches is performed by drawing many curves very close to each other. Each curve is a cubic in t formed by setting s at a constant $s=s_i$. We therefore need to find the spacing δs from one curve to the next so that no pixel gaps exist in between them. To compute the spacing δs in between the current curve $f(s=s_i,t)$ and the next curve, we run a series of testing curves in the orthogonal direction (i.e. s direction) at $t = (0.0, 0.125, \ldots, 1.0)$ and examine the step size used by those curves at the positions $s=s_i$. The minimum size used is then chosen as the spacing for the next curve $f(s_i+\delta s, t)$. When the δs gets smaller, it indicates that the next curve should be filled in closer to the current one; when δs increases, the next curve can be a little less close.

We explain next how AFD is used to adaptively adjust the spacing in between curves. For a bicubic patch $F(s,t)$ represented in forward difference basis,

$$F(s,t) = < f_x(s,t), f_y(s,t), f_z(s,t), f_w(s,t) >$$

each $f(s,t)$ is a bicubic function of s and t. For example the x component

$$f_x(s,t)=\left[B_0(t),B_1(t),B_2(t),B_3(t)\right]\begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix}\begin{bmatrix} B_0(s) \\ B_1(s) \\ B_2(s) \\ B_3(s) \end{bmatrix}$$

where $x_{i,j}$ are the x coordinates of the control points of the patch. A curve at a constant s, $f(s=s_i,t)$, is a cubic function represented in forward difference basis as

$$f(s=s_i,t)=dB_0(t)+cB_1(t)+bB_2(t)+aB_3(t)$$

where the four coefficients a,b,c,d are cubic functions of s in forward difference basis:

$$d(s)=x_{00}B_0(s)+x_{01}B_1(s)+x_{02}B_2(s)+x_{03}B_3(s)$$

$$c(s)=x_{10}B_0(s)+x_{11}B_1(s)+x_{12}B_2(s)+x_{13}B_3(s)$$

$$b(s)=x_{20}B_0(s)+x_{21}B_1(s)+x_{22}B_2(s)+x_{23}B_3(s)$$

$$a(s)=x_{30}B_0(s)+x_{31}B_1(s)+x_{32}B_2(s)+x_{33}B_3(s)$$

We apply AFD to these four cubic functions to generate the value of coefficients for the next curve. When the spacing δs for the next curve is the same as the previous one, the E matrix is applied. If the spacing for the next curve doubles, L^{-1} and then the E matrix are applied to double the spacing. If the spacing halves, we apply L and then the E matrix to reduce it. We are still examining methods for minimizing this redundancy through subdivision and tuning of the adjustment criterion.

Figure 7 shows a Phong shaded Utah teapot rendered on a 1152 x 900 screen with 80 patches using the AFD technique, for comparison with the equivalent polygon shaded version in Figure 8 containing 4060 triangles.

Trimmed patches are rendered by scan converting the trimming region in s,t space using the δs scanline width. (Here a scanline in s,t space is different from a scanline in the screen space.) This produces a "scanline" curve segment at each constant s_i bounded by one or more tmin, tmax pairs. These curve segments are rendered with clipping to the appropriate tmin and tmax. Figure 9 shows a shaded, image mapped, bicubic patch trimmed with a SUN logo.

Discussion

In rendering curves we set the threshhold of adjustment to be 0.5 and 1.0, i.e. we adjust up if x and y step by less than .5 pixel and we adjust down if x or y step by greater than 1 pixel. Using this threshhold we do not overpaint too many pixels, and neither do we leave gaps between pixels. Patches are rendered by filling many curves very close together. However, using the 0.5 and 1.0 threshhold in rendering a patch we tend to get missing pixels in the patch. This problem is solved by reducing the pixel adjustment threshold down to 0.35 and 0.7, instead of by reducing the spacing in between adjacent curves. We are currently trying to establish the optimal adjustment criterion for ensuring no pixel gaps.

For performance comparison purposes, we used the following three schemes to render a wireframe mesh of curves for a piece of teapot handle on a 512 x 512 screen with ten curves in each direction: (1) ordinary forward differencing, (2) adaptive subdivision, and (3) adaptive forward differencing. In this test, the ordinary forward differencing technique took 8192 forward steps, the adaptive subdivision technique took 3887 subdivisions, and AFD took 49 adjust_up's, 36 adjust_down's and 3910 forward steps. It is obvious that the ordinary forward differencing technique usually requires more forward steps than our technique because it uses the smallest step required for no gaps and cannot adjust to a longer step when appropriate. Each forward operation takes three adds, each adjust_up or adjust_down takes 3 adds and 2 multiplies. The first scheme required a total of 24516 adds. Our technique required 11900 adds and 255 multiplies. The subdivision technique took a total of 11661 adds and 15548 multiplies, where a single subdivision requires 3 adds and 4 multiplies.

We used the adaptive subdivision technique and our patch rendering technique to compare the patch rendering performance on rendering a piece of teapot body and a piece of teapot handle on a 512 x 512 screen. The termination condition we used in the subdivision technique was to constrain the minimum bounding box of the control points of a Bezier patch within 1.0 by 1.0. The subdivision technique on the teapot body required 86380 subdivisions to fill the entire patch. Since a subdivision takes 36 adds and 48 multiplies, it requires approximately 3 million adds and 4 million multiplies. Our technique required 708 adjust up, 513 adjust down, and 218513 forward step operations. Thus AFD used approximately 0.66 million adds and 3700 multiplies. Our method has a curve set up overhead of 12 adds per curve - a total of 6000 adds in this test case, which is negligible. In the case of the teapot handle, it took 143379 subdivisions with adaptive subdivision, whereas the new method performed 537 adjust up, 727 adjust down, and 187386 forward step operations, i.e., 5.16 million adds and 6.88 million multiplies against 0.56 million adds and 2528 multiplies.

Clearly, both subdivision and AFD can be implemented with integer arithmetic given sufficient precision. In both methods the above multiplies can be performed using simple shifts. The shifts required for the L and L^{-1} matrices can be implemented with "wires" in hardware since all elements are integer powers of 2. A complete error analysis of a fixed point integer implementation of AFD is currently being conducted.

The relatively poor performance of adaptive subdivision is due to the fact that a subdivision operation takes significantly more computation than a forward difference operation. This new method has the advantage of producing picture quality equivalent to adaptive subdivision without the memory stack management overhead of recursive subdivision and is thus more suitable for hardware implementation. AFD also makes patch rendering performance competitive with polygon rendering. When doing image mapping and patch trimming, our technique operates in s,t space but polygon methods operate in the screen scanline order, therefore, our method does not require a transformation from screen space to image coordinates as the polygon method does.

Acknowledgements

The following people contributed greatly to the ideas, simulations, and design of these algorithms: Jerry Evans, David Elrod, Nola Donato, Bob Rocchetti, Sue Carrie, Serdar Ergene, Jim Van Loo, Paul Tien, and Mark Moyer.

References

1. Jerry Van Aken and Mark Novak, "Curve-Drawing Algorithms for Raster Displays," *ACM Transactions on Graphics*, vol. 4, no. 2, pp. 147-169, April 1985.
2. Edwin Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Thesis in Computer Science, University of Utah, UTEC-CSc-74-133, 1974.
3. George M. Chaikin, "An Algorithm for High Speed Curve Generation," *Computer Graphics and Image Processing*, vol. 3, pp. 346-349, 1974.
4. Steven A. Coons, *Surfaces for Computer-Aided Design of Space Forms*, Project MAC, MIT, MAC-TR-41, June 1967.
5. Jeffrey Lane, Loren Carpenter, Turner Whitted, and James Blinn, "Scan Line Methods for Displaying Parametrically Defined Surfaces," *CACM*, vol. 23, no. 1, January 1980.
6. Jeffrey M. Lane and Richard F. Riesenfeld, "A Theoretical Development for the Computer Generation of Piecewise Polynomial Surfaces," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-2, no. 1, pp. 35-46, January 1980.
7. M. L. V. Pitteway, "Algorithm for drawing ellipses or hyperbolae with a digital plotter," *Computer Journal*, vol. 10, no. 3, pp. 282-289, Nov. 1967.
8. Vaughan Pratt, "Techniques for Conic Splines," *Computer Graphics*, vol. 19, no. 3, July 1985.
9. Michael Shantz and Sheue-Ling Lien, "Shading Bicubic Patches," *Computer Graphics*, vol. 21, no. 4, July 1987.

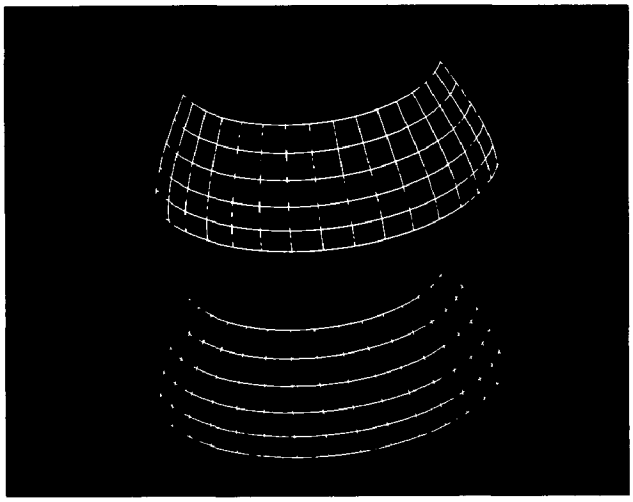

Figure 5. Comparison of antialiased and ordinary non_rational cubic curves rendered with adaptive forward difference scheme.

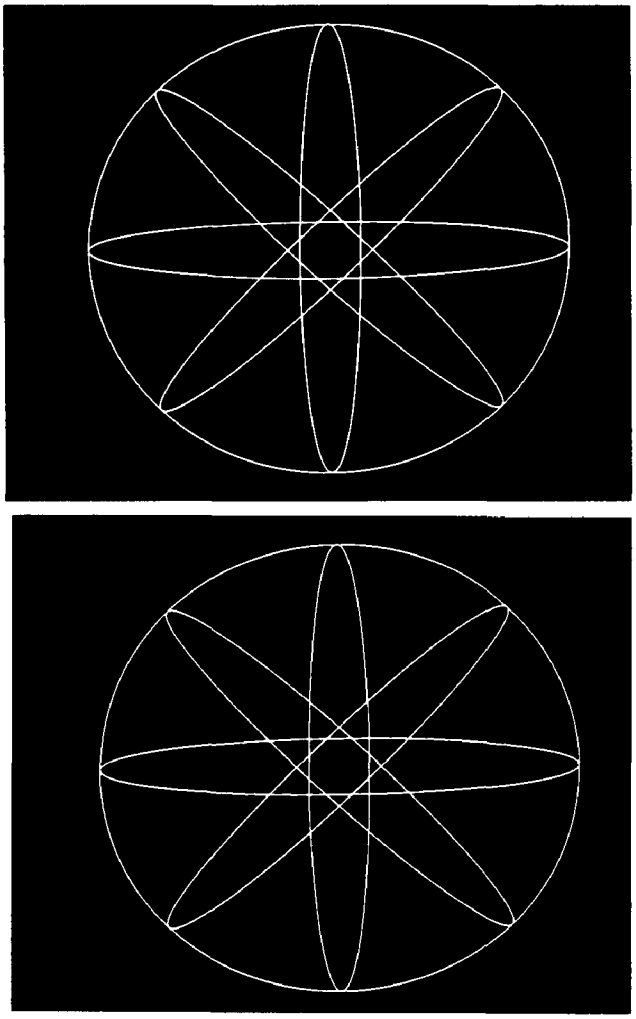

Figure 6. Comparison of antialiased and ordinary conics rendered with adaptive forward difference scheme.

Figure 7. Adaptive forward difference rendering of the Utah teapot using 80 patches.

Figure 8. Classical polygon rendering of the Utah teapot using 4060 triangles.

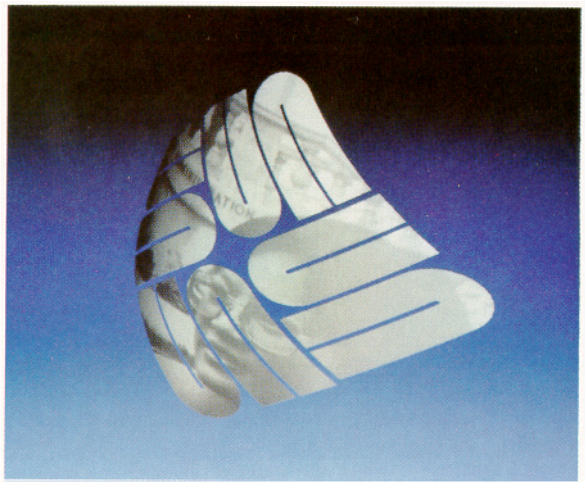

Figure 9. An image_mapped bicubic patch trimmed with a SUN logo.