Rendering Cubic Curves and Surfaces with Integer Adaptive Forward Differencing

Sheue-Ling Chang, Michael Shantz and Robert Rocchetti

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

Abstract

For most compute environments, adaptive forward differencing is much more efficient when performed using integer arithmetic than when using floating point. Previously low precision integer methods suffered from serious precision problems due to the error accumulation inherent to forward differencing techniques. This paper proposes several different techniques for implementing adaptive forward differencing using integer arithmetic, and provides an error analysis of forward differencing which is useful as a guide for integer AFD implementation. The proposed technique using 32 bit integer values is capable of rendering curves having more than 4K forward steps with an accumulated error of less than one pixel and no overflow problems. A hybrid algorithm employing integer AFD is proposed for rendering antialiased, texture-mapped bicubic surfaces.

CR Categories and Subject Descriptors:
I.3.3 [**Computer Graphics**]: Picture/Image Generation - Display algorithms;
I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modelling - Curve, surface, and Geometric algorithms.

Additional Key Words and Phrases: adaptive forward differencing, parametric curve, and texture.

 ACM-0-89791-312-4/89/007/0157 $00.75

Introduction

Much progress has been made in recent years on techniques for computer aided geometrical design. Parametric curves and surfaces including non-uniform rational B-splines are commonly used to describe surfaces of objects being designed. Such objects have typically been rendered by tesselating to bilinear quadrilaterals or triangles which are then rendered using widely available special purpose hardware for polygons. Alternatively, isoparametric curves across the surface are rendered by tesselating to polylines which are rendered using special purpose hardware. These simple piecewise linear approximations are also used to drive numerically controlled milling machines.

Research has focused largely on subdivision methods for rendering and modelling [4, 11]. Less progress has been made on hardware techniques for direct rendering of higher order curves and surfaces. Recursive subdivision is expensive for hardware implementation due to the high speed stack memory requirements and the computational complexity increase over polygon rendering methods. Incremental solutions of the implicit equations have been developed for conics [2, 5, 13, 14], and a few hardware curve generators have been built.

Adaptive forward difference (AFD) is an incremental technique proposed previously for rendering parametric curves and surfaces [12, 17]. Abi-Ezzi [1] adapted the AFD technique to a new basis which has a convex hull property yet retains much of the efficiency of the forward difference basis. AFD is an extension of forward differencing and is similar to adaptive subdivision in its dynamic step size adjustment. AFD differs from recursive subdivision or ordinary forward differencing by generating points sequentially along the curve while adjusting the parametric increment to give pixel sized steps. AFD allows a surprisingly simple hardware implementation, and is compatible with frame buffer memory interleaving for high performance. With special purpose hardware for rendering these curves and surfaces directly, the overhead of subdivision and stack memory is eliminated and the quality of the rendered surface is as good.

AFD in software, with its high speed and low cost, is also attractive for parametric curve and surface rendering. Various implementation methods for integer adaptive forward differencing are discussed in this paper. One 32-bit scheme is proposed which offers advantages in performance, precision, and output format.

The error accumulation in forward differencing is analyzed, including a comparison of the error accumulation in different integer AFD schemes. Integer AFD may also be used for fast rendering of antialiased texture-mapped bicubic surfaces.

Integer Adaptive Forward Differencing

A parametric curve can be tesselated into n segments using equal parametric increments at $1/n$ spacing (assuming that the parameter of the function ranges from 0.0 to 1.0). This is done by first converting a polynomial function into the forward difference basis [8]. The points along the curve can then be generated incrementally with three additions per cycle in the case of a cubic.

The authors described three operators in adaptive forward differencing in the previous paper. Adjust-down is an operator for reducing the parametric increment by half and reducing the step size to approximately one pixel per step when the x,y screen step size is more than one pixel. The adjust-up operation doubles the parametric increment to increase the change in x,y coordinates when the step size is less than 1/2 pixel. The two adjustment operations are performed by transforming the coefficients of the coordinate functions by the the adjust-up matrix $[U]$ or the adjust-down matrix $[D]$:

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix} \quad \begin{cases} c' = (c<<1) + b; \\ b' = (a+b)<<2; \\ a' = a<<3; \end{cases}$$

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & -1/8 & 1/16 \\ 0 & 0 & 1/4 & -1/8 \\ 0 & 0 & 0 & 1/8 \end{bmatrix} \quad \begin{cases} a' = a>>3; \\ b' = (b>>2) - a'; \\ c' = (c - b')>>1; \end{cases}$$

where a, b, c, and d are the coefficients of the curve represented in forward difference basis. The notations "a<<3" and "a>>3" indicate a left or right shift of coefficient a by three bits. When the step size is within the desired range, the forward-step matrix $[F]$ is applied to move one step forward.

$$F = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{cases} output\ (d); \\ d' = d+c; \\ c' = c+b; \\ b' = b+a; \end{cases}$$

Adaptive forward differencing can be implemented in either floating point or fixed point arithmetic. Floating point implementation offers more precision, but is usually more expensive. Higher performance can be achieved if AFD is performed using integer additions and shifts. There are two straight forward integer AFD implementations. The 64-bit integer AFD, with two 32-bit integers, provides adequate precision for most displays, but the computational cost is more than doubled over 32 bit integer operations due to the carry. The 32-bit integer AFD shown in Figure 1 has an advantage in performance but a disadvantage in precision. It can handle only a very small number of forward steps. Subdivisions are required for large curves to avoid excessive error accumulation. In the error analysis section we show that this scheme can only handle cubic curves involving up to 100 forward steps on a 1Kx1K screen. The overflow protection bit is used here to avoid clipping a curve exactly to the display windown boundary which is quite expensive compared to vector clipping.

a	12.20
b	12.20
c	12.20
d	12.20

Figure 1. An integer forward difference implementation with a sign bit, an overflow-protection bit, 10 integer and 20 fractional bits.

The approach in Figure 2 has the a, b, and c registers aligned and the d register in a 16.16 fract format. A forward step is performed by outputing (d>>16) where

$$d' = d + (c>>8);\ c' = c + b;\ b' = b + a;$$

Compared to the first method where all registers are aligned, this approach provides eight more fractional bits which increases the maximum number of forward steps to be 512, at a cost of one extra shift per forward step.

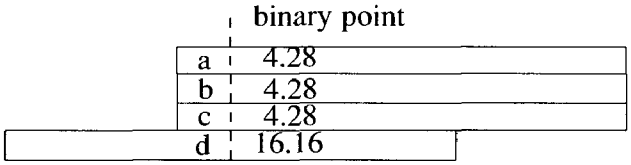

Figure 2. An integer forward difference implementation with the a, b, and c registers aligned and the d register in 16.16 format. Register a has 28 fractional bits.

Bartels etc. [3] discussed using successive guard bits in the forward difference registers to achieve higher precision with less bits. Each register except a contains n successive guard bits for processing up to 2^n forward steps, Figure 3.

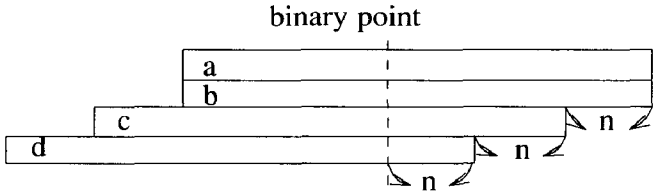

Figure 3. Forward difference registers with n guard bits.

A forward step operation is performed with n guard bits truncated before adding a register:

$$F_i = \begin{bmatrix} 1 & 2^{-n} & 0 & 0 \\ 0 & 1 & 2^{-n} & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{cases} output\ (d>>n); \\ d' = d + (c>>n); \\ c' = c + (b>>n); \\ b' = b + a; \end{cases}$$

The constant 2^{-n} indicates a truncation of n guard bits. The use of guard bits significantly increases the number of forward steps allowable with 32-bit integer forward differencing. Register initialization is also simpler with guard bits. A parametric curve $f(t) = At^3 + Bt^2 + Ct + D$ is converted to the forward difference basis with the following transformation where A, B, C, and D are the control points in polynomial basis, a, b, c, and d are the control points in forward difference basis, $\delta = 1/n$ is the parametric increment:

$$\begin{bmatrix} d \\ c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \delta & \delta^2 & \delta^3 \\ 0 & 0 & 2\delta^2 & 6\delta^3 \\ 0 & 0 & 0 & 6\delta^3 \end{bmatrix} \begin{bmatrix} D \\ C \\ B \\ A \end{bmatrix}$$

The forward difference basis functions are

$$B_3 = \frac{t(t-1)(t-2)}{6,} \quad B_2 = \frac{t(t-1)}{2}, \quad B_1 = t, \quad B_0 = 1$$

This transformation can be performed using integer additions and shifts if the tesselation number is a power of two, ie., the scaling factor $\delta = 2^{-n}$ as follows:

$$\begin{cases} d = D \\ c = (C + (B + (A >> n)) >> n) >> n \\ b = (2B >> 2n) + (6A >> 3n) \\ a = 6A \gg 3n \end{cases}$$

With n successive guard bits in each register, the c coefficient is scaled by 2^n and the a and b coefficients are scaled by 2^{2n}. The initialization is simplied as follows:

$$\begin{bmatrix} d \\ c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & \delta & \delta^2 \\ 0 & 0 & 2 & 6\delta \\ 0 & 0 & 0 & 6\delta \end{bmatrix} \begin{bmatrix} D \\ C \\ B \\ A \end{bmatrix} \begin{cases} d = D \\ c = C + (B + (A >> n)) >> n \\ b = 2B + (6A >> n) \\ a = 6A \gg n \end{cases}$$

To utilize a forward difference register set with n successive guard bits in integer AFD the adjust-up $[U]$ and adjust-down $[D]$ operators can be modified as follows

$$U_i = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 2^{-n} & 0 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix} \begin{cases} c' = (c << 1) + (b >> n); \\ b' = (a+b) << 2; \\ a' = a << 3; \end{cases}$$

$$D_i = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & -2^{-n}/8 & 2^{-n}/16 \\ 0 & 0 & 1/4 & -1/8 \\ 0 & 0 & 0 & 1/8 \end{bmatrix} \begin{cases} a' = a >> 3; \\ b' = (b >> 2) - a'; \\ c' = (c - (b' >> n)) >> 1 \end{cases}$$

to reflect the alignment of the binary points of the registers. The factor 2^{-n} indicates the existence of the guard bits. The number of guard bits remains unchanged throughout the process.

Victor Klassen [9, 10] extended this pseudo floating concept to the integer AFD implementation. He modified the adjust-up and adjust-down operators $[U_K]$ and $[D_K]$ to incorporate the usage of guard bits in the AFD registers and thus to vary the number of guard bits following each adjustment operation. One guard bit is added before an adjust down and eliminated after an adjust up:

$$U_K = \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 2^{-n-2} & 0 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad D_K = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & -2^{-n} & 2^{-n-1} \\ 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When the registers acquire one extra guard bit after an adjust-down and lose one guard bit after an adjust-up, matrix $[U_K]$ can be derived from $[U_i]$ by scaling the first row of $[U_i]$ by 1/2, the second row by 1/4, and the third and fourth rows by 1/8. Similarly, $[D_K]$ can be derived by scaling the first row of $[D_i]$ by 2, the second row by 4, and the third and fourth rows by 8. We notice that this technique keeps the a register constant and only shifts the contents of b, c and d registers during an adjustment.

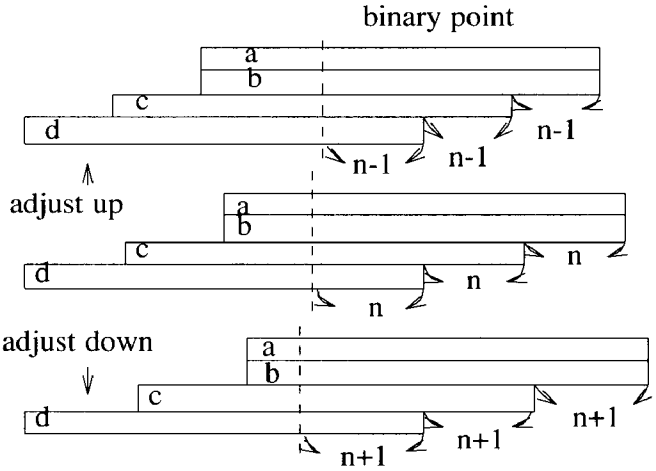

Figure 4. The re-alignment of forward difference registers in Klassen's technique associated with an adjust-up or adjust-down operation. The contents of the a register remains constant during the entire process.

There are different pseudo floating implementations for integer AFD depending on the application. Figure 5 illustrates the register format of our technique where the d register stays constant in an adjustment, and the a, b and c registers are shifted instead.

The new adjust-up matrix $[U_C]$ is derived from $[U_i]$ by scaling the first row of the matrix by 1, the second row by 1/2, and the third and fourth rows by 1/4.

$$U_C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 2^{-n-1} & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 \end{bmatrix} \begin{cases} c' = c + (b >> (n+1)); \\ b' = b + a; \\ a' = a << 1; \\ n' = n - 1; \end{cases}$$

The adjust-down matrix $[D_C]$ is derived from $[D_i]$ by scaling the first row by 1, the second row by 2, and the third and fourth rows by 4.

$$D_C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -2^{-n-2} & 2^{-n-3} \\ 0 & 0 & 1 & -1/2 \\ 0 & 0 & 0 & 1/2 \end{bmatrix} \begin{cases} a' = a >> 1; \\ b' = b - a'; \\ c' = c - (b' >> (n+2)); \\ n' = n + 1; \end{cases}$$

These transformations preserve the contents of the d register in an adjustment while eliminating one guard bit successively from the a, b and c registers during an adjust-up, and adding one guard bit successively to the a, b and c registers during an adjust-down.

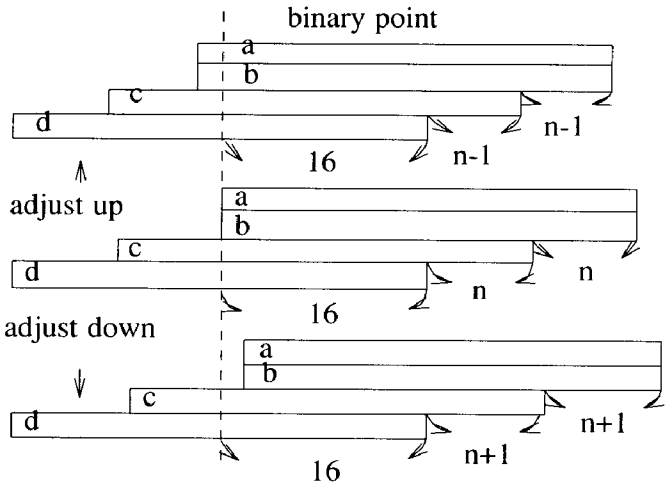

Figure 5. The re-alignment of the forward difference registers in our technique during an adjust-up or adjust-down operation. The binary point of the d register remains fixed.

Pseudo floating point is very important in order to make the most effective use of fixed point arithmetic and fixed width registers. Both methods described above vary the number of guard bits in an adjustment thus giving the effect of a floating binary point. Since single float variables in computers contain only 24 bits of mantissa, these 32-bit pseudo floating point AFD implementations could actually provide more precision than single precision float implementation when the registers are initialized with double precision floating point values.

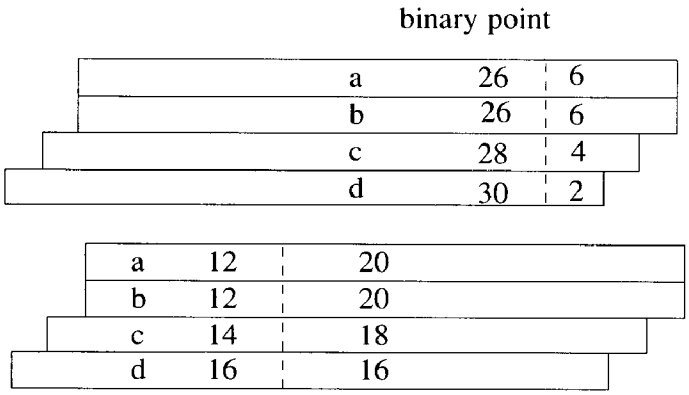

Figure 6. A comparison of the register layout of the two methods.

By comparing Figures 4 and 5, we can see that the difference between the two methods is that Klassen's method produces an output with a floating binary point while our method produces a fixed point 16.16 fract format output. The new method also offers several advantages including

(1) an easier and less expensive initialization computation for fract format inputs and outputs,

(2) fewer operations in the adjust up and down operations,

(3) a 16.16 fract format output which better suits subsequent integer instructions, and

(4) higher precision when the velocity of a curve decreases and the parametric step size is adjusted severely upward.

When the curve velocity is approximately 4.0 and the parametric increment is at 2^{-2}, Klassen's registers are shifted so far to the left that only few fractional bits remain, whereas the registers in our technique always retain at least 16 fractional bits even in regions of minimum curve velocity, as shown in Figure 6.

Error Analysis on Forward Differencing

While forward differencing has been a popular and inexpensive method for incrementally evaluating points along a parametric function, its rapid error accumulation has been a problem. The user must carefully analyze the error accumulation characteristics of the method to ensure accuracy. In this section the error accumulation properties of forward differencing are analyzed and guide lines are presented for evaluating the error bounds of various implementation schemes.

1. Error analysis on registers without guard bits

For a cubic, the points on the curve can be computed incrementally using three additions per cycle. The contents accumulated in the registers at each cycle are shown in the table. We first assume that the registers are aligned as shown in Figure 1 and that each register has n fractional bits.

cycle	Contents of Forward Difference Register			
0	a	b	c	d
1	a	a+b	b+c	c+d
2	a	2a+b	a+2b+c	b+2c+d
3	a	3a+b	3a+3b+c	a+3b+3c+d
4	a	4a+b	6a+4b+c	4a+6b+4c+d
5	a	5a+b	10a+5b+c	10a+10b+5c+d
6	a	6a+b	15a+6b+c	20a+15b+6c+d
:	:	:	:	:
k	a	ka+b	ak(k-1)/2+ kb+c	ak(k-1)(k-2)/6 + bk(k-1)/2 + kc+d

The error accumulation problem can then be analyzed by examining the data in the d register after k iterations:

$$d = a\frac{k(k-1)(k-2)}{6} + b\frac{k(k-1)}{2} + kc + d$$

The error accumulated in the d register is approximately

$$e_d = E_a\frac{k(k-1)(k-2)}{6} + E_b\frac{k(k-1)}{2} + kE_c + E_d$$

which is dominated by the initial error in the a register. The initial errors in the registers are $E_a = E_b = E_c = E_d = 2^{-n}$.

The error accumulated in the d register after k cycles is approximately

$$e_d \approx E_a \frac{k(k-1)(k-2)}{6} + O(k^2)$$

A rough estimation on the error accumulation for cubic functions with $k=2^m$ forward steps is approximately $3m-2$ bits:

$$\frac{2^m(2^m-1)(2^m-2)}{6} < \frac{2^{3m}}{4}$$

Therefore, the minimum number of fractional bits required for 2^m forward steps is $n \geq 3m-2$. Based on this analysis, using 32-bit integer layout as shown in Figure 1 one can handle curves in a 1Kx1K grid up to $k=100$ forward steps. This assumes that the 32 bits are used for one sign bit + one overflow bit + 10 integer bits + 20 fractional bits. The a register in this case contains only twenty fractional bits. Or it can handle curves on a 512x512 grid up to $k=128$ forward steps, by assuming one sign bit, one overflow bit, 9 integer bits and 21 fractional bits.

2. Error analysis on registers with guard bits

With successive guard bits in the forward differencing registers as in Figure 3, the error accumulation is similar to the previous analysis except that additional error is introduced through the truncation of the guard bits. Again we analyze the error problem by examining the forward difference process and the contents accumulated in the registers. Each register has a different number of fractional bits, i.e., the a and b registers each have $3n$ bits, the c register has $2n$ bits, and the d register has n bits. The initial errors in the registers are $E_d = 2^{-n}$, $E_c = 2^{-2n}$ and $E_a = E_b = 2^{-3n}$. The content of the d register shown in the table indicates that the error accumulated in the d register after k cycles is dominated by three terms:

$$e_d \approx E_b \frac{k(k-1)(k-2)}{6} + E_c \frac{k(k-1)}{2} + k\,E_d + O(k^2)$$

Contents of Forward Difference Register			
0	b	c	d
1	a+b	b+c+E_c	c+d +E_d
2	2a+b	a+2b+c+2E_c	b+2c+d +E_c+ 2E_d
3	3a+b	3a+3b+c+3E_c	a +3b+3c+d+3E_c +3E_d
4	4a+b	6a+4b+c+4E_c	4a +6b+4c+d+6E_c +4E_d
5	5a+b	10a+5b+c+5E_c	10a +10b+5c+d+10E_c +5E_d
6	6a+b	15a+6b+c+6E_c	20a +15b+6c+d+15E_c +6E_d
:	:	:	:
k	ka+b	ak(k-1)/2+kb +c+k E_c	ak(k-1)(k-2)/6 + bk(k-1)/2 +kc+d+E_c k(k-1)/2+kE_d

* The a register column is not shown in this table.

A rough estimate of the error accumulated with this method is approximately $3m+1$ bits for 2^m forward steps. In this case the minimum number of fractional bits required for 2^m forward steps is $3n \geq 3m+1$.

This method offers a dramatic improvement in precision handling when implemented in 32-bit integers even though additional error is introduced through the truncation of guard bits. Using 32-bit integer layout as shown in Figure 7, a register contains up to forty two fractional bits. One can process curves in a $2^{16}x2^{16}$ screen space up to 2^{13} forward steps. This precision is more than adequate for the display screen sizes currently available on the market. This is, therefore, a much more useful and powerful technique than the previous method which gives only 100 forward steps in a 1Kx1K space.

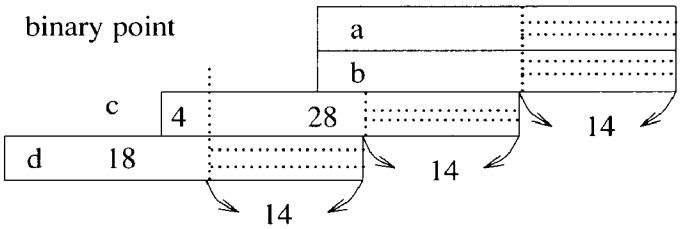

Figure 7. An integer forward difference implementation which can handle curves with up to 2^{13} forward steps.

3. Subpixel accuracy

For rendering antialiased curves, two or three additional bits of subpixel accuracy are desirable. In this case error propagation can be controlled, and higher subpixel accuracy can be achieved by subdividing to reduce the total number of forward steps in the curve. Error accumulation can be analyzed, and the maximum number of forward steps can be estimated by scaling up both the screen space and the curve, and then rendering the curve in this larger screen space. For example, an antialiased curve rendered in a 1Kx1K screen with 3 bits of subpixel address accuracy, is equivalent to a curve scaled up by a factor of eight and rendered in an 8Kx8K screen. The three least significant bits of the x,y coordinates are used as the subpixel address. The guidelines provided above can thus be used to analyze the error accumulation for a given application requirement and to calculate the limitations of a specified implementation.

4. Estimating the number of forward steps

Rockwood [15] derived a formula for estimating the number of forward difference steps required for a given Bezier curve with a specified minimum step size:

$$pmax = maximum\ |p_{i+1}-p_i| \quad \text{for } 0 \leq i \leq n-1$$

$$|f(t+\delta)-f(t)| \leq n\ \delta\ pmax$$

$$tesselation = \frac{1}{n\ pmax}$$

where n is the degree and the p_i's are the control points of the Bezier function. This formula is more intuitively clear by looking at the properties of Bezier functions:

(1) the derivative of a Bezier function $f(t)$ of degree n

$$f(t) = \sum_{i=0}^{i=n} p_i \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}$$

is a degree $n-1$ Bezier function $f'(t)$ using the vectors of the original polygon as its control points.

$$f'(t) = \sum_{i=0}^{i=m} n(p_{i+1}-p_i) \frac{m!}{i!(m-i)!} t^i (1-t)^{m-i}\ ,\ m=n-1;$$

(2) the derivative $f'(t)$ is the velocity of $f(t)$, (3) the magnitude of the derivative $|f'(t)|$ is bounded by the convex hull of its own control points, and is equal to maximum $(|n\ (p_{i+1}-p_i)|)$.

When using forward differencing to render a curve with no missing pixels, the step sizes in both the x and y directions should be no greater than one pixel step. Since the x and y coordinates of a Bezier curve are defined by two independent Bezier functions, the tesselation number can be computed in terms of the maximum convexhull of the x coordinate and the y coordinate instead of the convexhull of the length of the polygon:

$$xmax = maximum(x_{i+1} - x_i)$$
$$ymax = maximum(y_{i+1} - y_i)$$
$$pmax = maximum(xmax, ymax).$$

This tesselation number is less expensive to compute and is on the average 30% smaller than the number computed using vector length. However, it is big enough to ensure no x or y step size is more than one pixel.

5. Error bound on AFD without guard bits

In a previous paper the authors observed that the error accumulated using OFD gives an upper bound on the error accumulated using AFD. OFD uses a fixed parametric increment sufficiently small so as to prevent missing pixels in the highest velocity region of the curve. AFD, on the other hand, increases its parametric increment whenever the step size is too small. The parametric step size used in AFD is always greater than or equal to that used in OFD. Thus, the total number of forward steps in AFD is considerably smaller than that in OFD if the velocity of the curve is not constant. In the case of OFD, $2k$ forward steps can accumulate an error of approximately

$$e_d\ (OFD) \approx \frac{2k(2k-1)(2k-2)}{6}$$

With an adjust-up operation in AFD $2k$ forward steps is reduced to k forward steps which consequently introduces an error of approximately

$$e_d\ (AFD) \approx 2^3\ \frac{k(k-1)(k-2)}{6}$$

The 2^3 factor is due to the left shift 3-bits in an adjust-up operation. The result shows that the error accumulated in AFD is always less than or equal to the error accumulated in OFD:

$$\frac{2k(2k-1)(2k-2)}{6} > 2^3\ \frac{k(k-1)(k-2)}{6}$$

6. Error bound on AFD with guard bits

When using registers with successive guard bits, the error accumulated in the d register, as discussed previously, is dominated by three terms:

$$E_b \frac{k(k-1)(k-2)}{6} + E_c \frac{k(k-1)}{2} + kE_d$$

The initial errors in the registers with n guard bits are $E_d = 2^{-16}$, $E_c = 2^{-n}E_d$ and $E_b = 2^{-2n}E_d$. The error accumulated in the d register after 2^n forward steps is

$$e_d\ (OFD) \approx \left(\frac{2^n(2^n-1)(2^n-2)}{6}2^{-2n} + \frac{2^n(2^n-1)}{2}2^{-n} + 2^n\right) E_d$$

In the case of AFD, there are only $n-1$ successive guard bits in the registers after an adjust-up operation is performed. Ordinarily the initialization errors in registers with $n-1$ guard bits would be $E_d = 2^{-16}$, $E_c = 2^{-n+1}E_d$ and $E_b = 2^{-2n+2}E_d$. However, the initial errors in the a, b and c registers are magnified by two due to the adjust-up operation $[U_C]$, which results in

$$E_c \approx 2^{-n+2}E_d \quad \text{and} \quad E_b \approx 2^{-2n+3}E_d$$

The error accumulated after one adjust-up operation followed by 2^{n-1} forward steps is $e_d\ (AFD) \approx$

$$\left(\frac{k(k-1)(k-2)2^{-2n+3}}{6} + \frac{k(k-1)2^{-n+2}}{2} + 2^{n-1}\right) E_d$$

where $k = 2^{n-1}$. The result shows that $e_d(AFD) \le e_d(OFD)$, i.e. the error accumulated in AFD through an adjust up operation and 2^{n-1} forward steps is no greater than the error accumulated in OFD through 2^n forward steps.

7. Overflow control

The issues of precision control and estimation of the number of forward steps have been discussed above. Overflow is another issue of great concern when using integer AFD. The following section will analyze this issue using cubic Bezier curves and the convex hull property of Bezier functions. A cubic Bezier curve is defined by four control points, p_a, p_b, p_c, and p_d which reside in a screen size of $2^{13}x2^{13}$:

$$|p_a| < 2^{13},\ |p_b| < 2^{13},\ |p_c| < 2^{13},\ |p_d| < 2^{13}$$

Three times the convexhull of the curve is bounded by 2^n

$$3|p_a - p_b| < 2^n,\ 3|p_b - p_c| < 2^n,\ 3|p_c - p_d| < 2^n$$

Here the notation "$3|p_a - p_b| < 2^n$" implies the x coordinate $3|p_a(x) - p_b(x)| < 2^n$ and the y coordinate $3|p_a(y) - p_b(y)| < 2^n$. The Bezier control points can be transformed into polynomial basis by

$$\begin{cases} D = p_d \\ C = 3(p_c - p_d) \\ B = 3(p_b - p_c) - 3(p_c - p_d) \\ A = (p_a - p_b) - 2(p_b - p_c) + (p_c - p_d) \end{cases}$$

The coefficients in polynomial basis are then converted into forward difference basis. Under the assumptions above, the polynomial coefficients A, B, C, and D are bounded by

$$|D| < 2^n,\ |C| < 2^n,\ |2B| < 2^{n+2},\ |6A| < 2^{n+3}.$$

Using the register layout shown in Figure 7, the d register contains 18 integer bits and 14 fractional bits. The maximum value the registers can hold at initialization time is less than 2^{17}. If the Bezier control points satisfy the following constraints the maximum number of forward steps required is 2^{13} and there will be no overflow problem at the initialization stage.

$$3|p_a - p_b| < 2^{13},\ 3|p_b - p_c| < 2^{13},\ 3|p_c - p_d| < 2^{13}$$

Next we look at the possibility of overflowing during the forward differencing process. We first extend a given curve $f(t) = t^3 p_a + 3t^2(1-t)p_b + 3t(1-t)^2 p_c + (1-t)^3 p_d$ by drawing an extension to the curve for the parameter range t=1.0 to t=2.0, as shown in Figure 8.

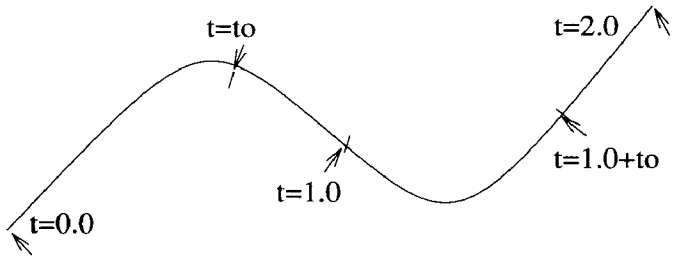

Figure 8. A segment of curve with the parameter ranging from $t = t_o$ to $t = 1.0 + t_o$.

At any instant during the process when the parameter is at $t = t_o$, the contents in the four registers, a, b, c, and d, are the forward difference coefficients of the portion of the extended curve with parameter ranging from $t = t_o$ to $t = 1.0 + t_o$ and scaled by the current tesselation δ :

$$\begin{cases} d = f(t_o) \\ c = f'(t_o) + f''(t_o)\delta / 2 + f'''(t_o)\delta^2 / 6 \\ b = f''(t_o) + f'''(t_o)\delta \\ a = f'''(t_o)\delta \end{cases}$$

The derivatives $f'(t), f''(t)$ and $f'''(t)$ are lower order Bezier functions.

$$f'(t) = 3[t^2(p_a - p_b) + 2t(1-t)(p_b - p_c) + (1-t)^2(p_c - p_d)]$$

$$f''(t) = 6[t(p_a - 2p_b + p_c)] + (1-t)(p_b - 2p_c + p_d)]$$

$$f'''(t) = 6[(p_a - p_b) - 2(p_b - p_c) + (p_c - p_d)]$$

The bound on the convex hull of functions $|f'(t)|$ is 2^{13}, on functions $|f''(t)|$ is 2^{15}, and on functions $|f'''(t)|$ is 2^{16}. Therefore, the magnitude of the derivatives satisfy the constraints at any instant $0.0 \le t_o \le 1.0$

$$|f(t)|,\ |f'(t)|,\ |f''(t)|,\ |f'''(t)| < 2^{16}$$

With the integer AFD implementation we proposed in Figure 5, the d register has 16 integer bits and 16 fractional bits. The maximum number of forward steps this scheme can handle is 2^{12}. In this case, the constraints on the convexhull of the control polygon is

$$3|p_a - p_b| < 2^{12}, 3|p_b - p_c| < 2^{12}, 3|p_c - p_d| < 2^{12}$$

Integer AFD for Surfaces

AFD can be used to render shaded, textured, and trimmed surfaces. Texture and imagery can be mapped onto a surface as a function of the parameters s and t. Shading and image mapping on a bicubic surface is performed by drawing many isoparametric curves very close together so that no pixel gaps exist in between. Each isoparametric curve is a cubic function of parameter t defined by a constant $s = s_i$. The spacing δs from one curve to the next is measured to decide whether the next curve should be drawn closer to or farther from the current curve.

A polynomial bicubic function can be converted into forward difference basis by the transformation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \delta & \delta^2 & \delta^3 \\ 0 & 0 & 2\delta^2 & 6\delta^3 \\ 0 & 0 & 0 & 6\delta^3 \end{bmatrix} \begin{bmatrix} f_{00} & f_{01} & f_{02} & f_{03} \\ f_{10} & f_{11} & f_{12} & f_{13} \\ f_{20} & f_{21} & f_{22} & f_{23} \\ f_{30} & f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & \beta^2 & 2\beta^2 & 0 \\ 0 & \beta^3 & 6\beta^3 & 6\beta^3 \end{bmatrix}$$

where the $f_{i,j}$'s are the control points in polynomial basis, and δ and β are the parametric increments in the s and t directions, respectively. A surface can be tesselated using ordinary forward differencing into $m*n$ four-sided polygons or m and n isoparametric curves by choosing the scaling factors to be $\delta = 1/m$ and $\beta = 1/n$. After basis conversion, a surface is described by three coordinate functions, $f_x(s,t)$, $f_y(s,t)$, and $f_z(s,t)$, each being a bicubic function of s and t. An isoparametric curve at a constant $s = s_i$ is a cubic function

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

where the four coefficients a,b,c,d can be computed from the surface matrix $[A]$ using two dimensional AFD. These coefficients are stored in the first row of matrix $[A]$, i.e. a_{00}, a_{01}, a_{02}, and a_{03}. In the curve-to-curve outer loop, a forward-step $[F]$ operation is applied to the surface matrix $[A]$ to produce the coefficients of the next isoparametric curve :

$$\begin{bmatrix} a_{00}+a_{10} & a_{01}+a_{11} & a_{02}+a_{12} & a_{03}+a_{13} \\ a_{10}+a_{20} & a_{11}+a_{21} & a_{12}+a_{22} & a_{13}+a_{23} \\ a_{20}+a_{30} & a_{21}+a_{31} & a_{22}+a_{32} & a_{23}+a_{33} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

After a forward-step operation is applied, the first row of $[A]$, $a_{00}+a_{10}$, $a_{01}+a_{11}$, $a_{02}+a_{12}$, and $a_{03}+a_{13}$ are the coefficients of the next isoparametric curve. These coefficients are then used in the pixel-to-pixel inner loop for computing the address of the pixels on the isoparametric curve. If the spacing between the current curve and the next curve is too small, an adjust-up operation is performed to increase the spacing before a forward step takes place. If the spacing is too large, it is reduced with the $[D]$ operator before $[F]$ is applied:

$$\begin{bmatrix} a'_{0j} \\ a'_{1j} \\ a'_{2j} \\ a'_{3j} \end{bmatrix} = [U] \begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} a'_{0j} \\ a'_{1j} \\ a'_{2j} \\ a'_{3j} \end{bmatrix} = [D] \begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix}$$

The two initial parametric increments of AFD can be estimated based on the initial velocity of a surface, i.e. $\delta \approx \partial f/\partial s(0,0)$, $\beta \approx \partial f/\partial t(0,0)$. By choosing these factors to be powers of two, $\delta = 2^{-m}$ and $\beta = 2^{-n}$, the initialization can be greatly simplified.

1. Hybrid integer AFD for surfaces

AFD surface rendering technique can also be implemented in integer arithmetic as long as the precision limitation is carefully considered. When using forward differencing for surface rendering, the error accumulates during both the curve-to-curve and pixel-to-pixel procedures. The curve-to-curve outer loop generates the coefficients of the next isoparametric curve and the pixel-to-pixel inner loop computing the pixel addresses. If 2^m forward steps are spent in the curve loop and 2^n steps in the pixel loop, the error in the a register is magnified by a factor of $2^m(2^m-1)(2^m-2)/6$ in the curve outer loop and a factor of $2^n(2^n-1)(2^n-2)/6$ in the pixel inner loop. The error can accumulate as much as 2^{m+n} forward steps have accumulated. As shown in the analysis section, the 32-bit integer AFD is adequate for rendering curves in a 4Kx4K space up to 2^{12} forward steps. Using 32-bit integer AFD for surface rendering, the maximum number of forward steps of the scheme is limited to $m+n \leq 12$, i.e. approximately 64 curves each no longer than 64 pixels. Surfaces requiring more than 64x64 forward steps must be subdivided.

Alternatively, one can implement the curve outer loop in 64-bit AFD to compute the coefficients of the isoparametric curves and pixel inner loop in 32-bit AFD to compute the pixel addresses. This hybrid scheme provides more precision and uses proper operators in each loop, for example, $[F]$, $[U]$, and $[D]$ in the outer loop and $[F_C]$, $[U_C]$, and $[D_C]$ in the inner loop. Matrix $[F_C]$ is the same as $[F_i]$ except that the d register is in 16.16 fixed point format. A forward step in the curve outer loop can be performed with twelve 64-bit-integer additions (a 64-bit-integer is stored in two 32-bit integers). A forward step in the pixel inner loop can be done with three additions and two register shifts.

The cost of register realignment before entering a pixel inner loop can be eliminated by properly scaling the coefficients of matrix $[A]$ in the t direction at the initialization time so that the first 32 bits of the coefficients a_{00}, a_{01}, a_{02}, and a_{03} can be input directly to the 32-bit integer AFD pixel inner loop:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \delta & \delta^2 & \delta^3 \\ 0 & 0 & 2\delta^2 & 6\delta^3 \\ 0 & 0 & 0 & 6\delta^3 \end{bmatrix} \begin{bmatrix} f_{00} & f_{01} & f_{02} & f_{03} \\ f_{10} & f_{11} & f_{12} & f_{13} \\ f_{20} & f_{21} & f_{22} & f_{23} \\ f_{30} & f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 2^{-n} & 2 & 0 \\ 0 & 2^{-2n} & 6\,2^{-n} & 6\,2^{-n} \end{bmatrix}$$

While rendering a surface, the decision to adjust up or down, or to forward step is made based on the distance between two adjacent isoparametric curves, $d(t) = |f(s+\delta s, t) - f(s,t)|$. The distance is a cubic function of t. The maximum distance between the two curves can be estimated using the Bezier convex hull property if the distance function $d(t)$ is converted into Bezier basis [6]. Converting $d(t)$ into Bezier basis at every curve is quite expensive. Instead of converting $d(t)$ at every curve, one can maintain a dual matrix $[B]$ to minimize this overhead. Matrix B is transformed into Bezier basis in the t direction and forward difference basis in the s direction. Matrix B and the surface matrix A are operated on synchronously in the curve-to-curve outer loop. The first row of A stores the coefficients of the current isoparametric curve. The second row of B stores the distance to the next isoparametric curve. The Bezier convex hull property can be applied directly to the second row of B to find the maximum distance between two consecutive isoparametric curves.

2. Dicing with integer AFD

An anti-aliasing technique [7] was proposed based on *dicing* to produce micropolygons followed by skittered subsampling and averaging for fast high-quality rendering of complex images. Micropolygons are the basic geometric unit of the technique. They are flat shaded quadrilaterals approximately 1/2 pixel on a side in screen space. Screen space derivatives of a surface are used to estimate how finely to dice, and subdivision and ordinary forward differencing are used to do the actual dicing.

The algorithm presented by Cook for dicing, shading, clipping, and rendering micropolygons is as follows:

```
Dice the primitive into a grid of micropolygons;
Compute normals and tanget vectors for the
    micropolygons in the grid;
Shade the micropolygons in the grid;
Break the grid into micropolygons;
For each micropolygon
    Bound the micropolygon in eye space;
    If the micropolygon is outside the hither-yon range,
        cull it;
    Convert the micropolygon to screen space;
    Bound the micropolygon in screen space;
    For each sample point inside the screen space bound
        If the sample point is inside the micropolygon
            Calculate the z of the micropolygon at the
                sample point by interpolation;
            If the z at the sample point is less than the z
                in the buffer
                Replace the sample in the buffer
                    with this sample;
```

Integer AFD can be a much more efficient method for dicing in screen space since the screen space step size is the threshold which controls adjust up, adjust down, or forward step.

A similar algorithm is proposed here using integer AFD to dice a surface.

```
AFD a primitive into strips of isoparametric curves;
AFD an isoparametric curve into a sequence of points;
Form a chain of micropolygons in between
    two consecutive isoparametric curves
For each micropolygon
    Bound the micropolygon in eye space;
    Cull the micropolygon if outside the hither-yon range;
    Bound the micropolygon in screen space;
    Cull the micropolygon if outside the viewport range;
    For each sample point inside the screen space bound
        If the sample point is inside the the micropolygon
            Interpolate the z at the sample point;
            If the z at the sample point is less than the z
                in the buffer
                Shade the micropolygon if not yet done;
                Replace the sample in the buffer
                    with this sample;
```

A surface is first sliced into many strips of isoparametric curves using 64-bit integer AFD. Two consecutive isoparametric curves are no more than 1/2 pixel apart. Each isoparametric curve is then tesselated into a sequence of points using 32-bit integer AFD with two consecutive points lying no more than 1/2 pixel apart. When the tesselation is done, two consecutive isoparametric curves form a chain of micropolygons. Each micropolygon is a triangle with two vertices on one isoparametric curve and one vertex on the other isoparametric curve. Micropolygons are approximately 1/2 pixel on a side in screen space, and flat shaded.

Using synchronized AFD technique, different components of a surface can be computed in difference spaces. For example, the forward difference coefficients of the coordinate functions can be computed in the screen space while the shading functions are calculated in the eye space.

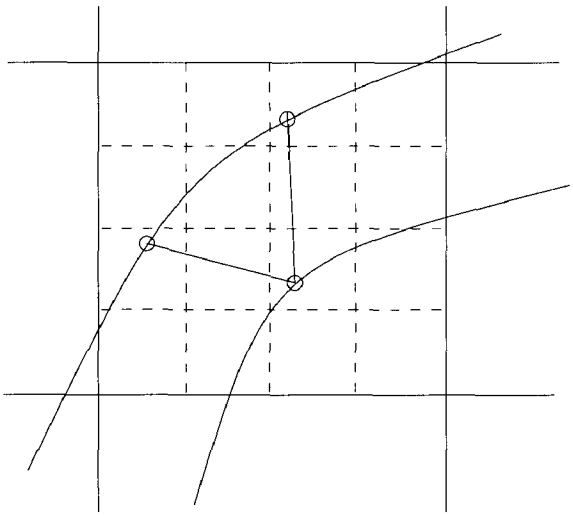

Figure 9. Dicing a bicubic surface into microtriangles with integer AFD in subpixel grid.

The techniques proposed in the shading paper [16] can be used for calculating the shading function N_x, N_y, N_z, $N.L$ and $N.H$ of a bicubic surface. Functions N_x, N_y, and N_z are the components of the normalized or un-normalized normal vector functions, and $N.L$ and $N.H$ are the inner products of the normal vector function with the light source vector L and the high light vector H

$$N.L(s,t) = L_x N_x(s,t) + L_y N_y(s,t) + L_z N_z(s,t)$$

$$N.H(s,t) = H_x N_x(s,t) + H_y N_y(s,t) + H_z N_z(s,t)$$

The coordinate functions and the shading functions are bicubic function of parameters s and t. These functions can be stepped along synchronously using integer AFD. The adjustment decision is made based on the screen step size information. The coordinate functions are diced directly in screen space, thus, the overhead of dicing a primitive in eye space and transforming the micropolygons into screen space is eliminated. The shading functions are computed and then diced in the eye space. The normal vector of each micropolygon is generated by AFD instead of computed at every micropolygon. The overhead of computing the inner products $N.L$ and $N.H$ per micropolygon is also eliminated in this method at the cost of AFDing two bicubic functions. The shading value of a pixel is computed in the inner most loop after the z buffer depth comparison, thus, the no cost is spent on computing the shading of hidden pixels.

For point light source, the un-normalized normal vector functions are used and two inner products and one normalization per pixel are required to calculate the sahding values $N.L$ and $N.H$.

Discussion

The threshold for adjustment in AFD may be set to 0.5 and 1.0, i.e. adjusting up when x and y steps are less than .5 pixel and adjusting down when x or y step is greater than 1 pixel. This ensures no missing pixels and reduces the pixel overpainting rate. Alternatively, one can use 1.0 and 2.0 as the thresholds, filling in a missing pixel whenever x or y takes a two-pixel step. The difference between the two is that in the former case redundant pixels are eliminated and in the latter, missing pixels are filled. The advantage of using a higher threshold, such as 10 to 20 pixels, is to reduce the number of forward steps and to improve the performance by rendering a curve with a polyline.

An adjust-down operation performed on a step size of greater than 1.0 can sometimes result in a step size of less than 0.5. To avoid getting into an infinite loop of adjust-ups and adjust-downs, it is helpful to enforce a forward step before an adjust-up in the implementation.

The computational complexity of the two surface rendering algorithms described in the previous section may be compared by assuming that a bicubic surface is diced into n^2 vertices constituting n^2 quadrilaterals or $2n^2$ triangles. It takes 16 multiplies and 12 additions to do a 4x4 point transformation. Computing the normal vectors of a quadrilateral takes 6 multiplies and 9 additions. The inner product computation for shading values $N.L$ and $N.H$ costs 6 multiplies and 4 additions per quadrilateral. The overhead of dicing a primitive in eye space, computing the shading values on every micropolygon, and transforming the micropolygons into screen space totals to 28 multiplies and 25 additions per quadrilateral. For a total of n^2 micropolygons, the overhead is $28n^2$ multiplies and $25n^2$ additions per surface.

Using the synchronized AFD technique, a forward-step in the curve outer loop costs 12 additions (or 48 additions when using 64-bit integer AFD) and a forward-step in the pixel inner loop costs 3 additions. Dicing a function into n isoparametric curves each with n vertices takes a total of $48n$ additions in the curve loop and $3n^2$ additions in the pixel loop. Computing five shading functions N_x, N_y, N_z, $N.L$ and and $N.H$ costs a total of $15n^2 + 240n$ additions per surface. The AFD method costs a total of $18\ n^2 + 288n$ additions. AFD has the overhead of computing the coefficients of the $N.L(s,t)$ and $N.H(s,t)$ shading functions. AFD has the advantages that (1) primitives are diced in screen space with better control of the micropolygon size; (2) the shading values of hidden pixels are not computed, which may account for a large savings in computation when the depth order of the scene is not optimized; (3) the computational complexity is significantly lower.

Acknowledgements

The authors are very grateful to Lewis Knapp for many helpful discussions and inputs to this project and for reviewing early drafts of this paper. Special thanks to John Recker for helping with the implementation and profiling. We thank Victor Klassen for several enlightening email conversations about his technique.

References

1. Salim Abi-Ezzi, "The Graphical Processing of NURB Surfaces," *Industrial Associate Review Summary*, November 1988. Rensselaer Design Research Center, Rensselaer Polytechnic Institute
2. 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.
3. Richard Bartels, John Beatty, and Brian Barsky, *An Introduction to Splines for use in Computer Graphics & Geometric Modeling*, pp. 400-406, Morgan Kaufmann Publishers, 1987.
4. Edwin Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Thesis in Computer Science, University of Utah, UTEC-CSc-74-133, 1974.
5. George M. Chaikin, "An Algorithm for High Speed Curve Generation," *Computer Graphics and Image Processing*, vol. 3, pp. 346-349, 1974.
6. Robert Cook, *Patch Work*, Tech. Memo 118, Computer Div., Lucasfilm Ltd., June 1985.
7. Robert Cook, Loren Carpenter, and Edwin Catmull, "The Reyes Image Rendering Architecture," *Proceedings of SIGGRAPH '87, Computer Graphics*, vol. 21, 1987.
8. James Foley and Andries Van Dam, "Fundamentals of Interactive Computer Graphics," *Addison-Wesley Publishers*, p. 533, 1982.
9. Victor Klassen, "Drawing Antialiased Cubic Spline Curves Using Adaptive Forward Differencing," *ACM Transactions on Graphics, under revision*, 1989.
10. Victor Klassen, "Integer Forward Differencing of Cubic Polynomials: Analysis and Algorithms," *ACM Transactions on Graphics, under revision*, 1989.
11. 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, pp. 35-46, 1980.
12. Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt, "Adaptive Forward Differencing for Rendering Curves and Surfaces," *Proceedings of SIGGRAPH '87, Computer Graphics*, vol. 21, 1987.
13. 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.
14. Vaughan Pratt, "Techniques for Conic Splines," *Proceedings of SIGGRAPH '87, Computer Graphics*, vol. 19, 1985.
15. Alyn Rockwood, *A Generalized Scanning Technique for Display of Parametrically Defined Surfaces*, 7, IEEE Computer Graphics and Applications, August 1987.
16. Michael Shantz and Sheue-Ling Lien, "Shading Bicubic Patches," *Proceedings of SIGGRAPH '87, Computer Graphics*, vol. 21, 1987.
17. Michael Shantz and Sheue-Ling Chang, "Rendering Trimmed NURBS with Adaptive Forward Differencing," *Proceedings of SIGGRAPH '88, Computer Graphics*, vol. 22, 1988.