Scanline-Order Image Warping using Error-Controlled Adaptive Piecewise Polynomial Approximation

Ken Turkowski, Apple Computer, Inc.
turk@apple.com
8 Jan. 2002, rev. 1 Apr. 2002

Abstract

A general class of real-time image-warping algorithms is based upon adaptively approximating the warping function on each scanline with piecewise polynomials, and rendering each piece by forward differencing, to yield a 3-10X improvement in speed over traditional techniques. The error is strictly bounded by controlling the interval length as a function of the derivatives of the warping function.

Categories and Subject Descriptors (according to ACM CCS): G.1.2 [Numerical Analysis]: Approximation — Approximation of Surfaces and Contours —Spline and Piecewise Polynomial Approximation; I.3.3 [Computer Graphics]: Picture/Image Generation —Display Algorithms; I.4.1 [Image Processing and Computer Vision]: Enhancement — Geometric Correction.

Additional Keywords: image processing, image-based rendering, morphing, numerical analysis, rendering, texture mapping.

1 Introduction

Image-based-rendering depends on the ability to interactively warp a set of images according to changes in viewing and other parameters. Examples of these are Morphing [Beier92], QuickTime VR [Chen95b], Plenoptic Rendering [McMillan95], Talisman [Torberg96], View Morphing [Seitz96], Light Fields [Levoy96], the Lumigraph [Gortler96], Tour Into the Picture [Horry97], Concentric Mosaics [Shum99], and Relief Texture Mapping [Oliveira00]. It is these types of applications that are targeted for order-of-magnitude rendering acceleration by taking advantage of improved algorithms on modern computer architectures.

The fastest rendering algorithms tend to be those based on incremental scan-conversion, because they take advantage of the coherency of the computation of a continuous function between pixels by updating the value of a complex function from the adjacent pixel by using a simple computation. The algorithm in this paper yields such a fast scan-conversion, yet does so with explicit control over the error. This has yielded a significant increase in rendering speed for interactive viewing of cylindrical, spherical, and cubic environment maps from a central viewpoint in QuickTime VR, its precision has allowed the composition of different media elements (panoramas, video sprites, 3D) maintaining the precise registration required to maintain a feeling of immersion.

2 Background: Incremental Scan-Conversion

The essence of rendering computer graphics by scan-conversion is the efficient evaluation of a vector-valued function at regularly spaced intervals along a curve in N-space.

With Gouraud-shaded, Z-buffered polygons [Foley90], the values of r, g, b, and z are evaluated at unit intervals in x. These can be efficiently evaluated by using linear forward differencing, e.g.

$$\begin{bmatrix} r & g & b & z \end{bmatrix}(x+1) = \begin{bmatrix} r & g & b & z \end{bmatrix}(x) + \begin{bmatrix} r_x & g_x & b_x & z_x \end{bmatrix}(x)$$

Note that we use parentheses above to indicate the arguments to a vector valued function, where the vector components are enclosed within brackets. Since the colors vary slowly in the Gouraud approximation to the shading equation, it suffices to use a constant delta vector without apparent rendering artifacts.

When texture mapping, however, the colors vary at a much higher rate than with Gouraud shading, so the perspective correction is important [Wolberg90]. In the incremental equation,

$$\begin{bmatrix} u & v \end{bmatrix}(x+1) = \begin{bmatrix} u & v \end{bmatrix}(x) + \begin{bmatrix} u_x & v_x \end{bmatrix}(x),$$

the first partial difference is not constant, although if planar polygons are being texture-mapped, it is constant when represented in homogeneous space:

$$\begin{bmatrix} uw & vw & w \end{bmatrix}(x+1) = \begin{bmatrix} uw & vw & w \end{bmatrix}(x) + \begin{bmatrix} (uw)_x & (vw)_x & w_x \end{bmatrix}$$

When curved surfaces are being texture-mapped (or when nonlinear, non-projective texture-mapping is used on planar polygons), then we cannot use this trick. However, we can use higher-order interpolation, such as that represented by the cubic forward differencing equation

$$\begin{aligned} &\begin{bmatrix} u & u_x & u_{xx} & u_{xxx} & v & v_x & v_{xx} & v_{xxx} \end{bmatrix}(x+1) = \\ &\begin{bmatrix} u & u_x & u_{xx} & u_{xxx} & v & v_x & v_{xx} & v_{xxx} \end{bmatrix}(x) + \\ &\begin{bmatrix} u_x & u_{xx} & u_{xxx} & 0 & v_x & v_{xx} & v_{xxx} & 0 \end{bmatrix}(x) \end{aligned} \qquad (2.1)$$

This sort of computation is particularly attractive on modern processors, which have SIMD vector instructions capable of performing the above operation in two or four instructions.

There are two things necessary to implement image warping using polynomial forward differencing as embodied in eq. (2.1):

(1) Computation of the forward differencing coefficients

$$\mathbf{U} = \mathbf{U}\left(\mathbf{f}(\cdot,\cdot), x, y, \mathcal{N}\right)$$

(2) Determination of the valid interval length, given an accuracy ε.

$$\mathcal{N} = \mathcal{N}\left(\mathbf{f}(\cdot,\cdot), x, y, \varepsilon\right)$$

We address both of these in this paper. These are both computed quickly at run-time, though a significant amount of time may be needed at design time.

3 Relation to Prior Work

Catmull and Smith [Catmull80] develop a 2-pass technique, where the image warping function is decomposed into two one-dimensional transformations. Each horizontal scanline from the source image is warped independently of the other scanlines and written to an intermediate buffer. Then each vertical scanline from the intermediate buffer is warped independently and written to the destination buffer. Sometimes it is better or even necessary to do the vertical pass first, depending on the nature of the transformation. The authors illustrate the 2-pass technique with image rotation, image viewed in perspective, and texture-mapped bilinear and biquadratic patches. Incremental forward-differencing techniques can be used in each of the passes, though the authors do not mention this. The authors mention two problems: (1) the bottleneck problem resulting from a down-scaling in one pass followed by up-scaling in the next, resulting in loss of resolution; (2) and foldover, when the inverse of one of the mapping functions is multi-valued.

Paeth [Paeth86] developed a three-pass technique for image rotations. Unlike the Catmull-Smith technique, this algorithm avoids scaling, but does so at the expense of requiring two intermediate buffers for three passes. Each pass (horizontal, vertical, then horizontal) implements a shear transformation, and the innermost loop is simply a subpixel translation of a one-dimensional array of pixels. This technique was independently reported by Tanaka [Tanaka86].

Chen and Miller [Chen95a] describe a two-pass algorithm to interactively view cylindrical environment maps as specified by a projective camera. In one pass, an arctangent warp is performed on each horizontal scanline with the use of a lookup table, and in the other pass, each vertical scanline is scaled.

Wolberg and Boult [Wolberg89] generalize the Catmull-Smith technique by using large tables to store the warping function, and provide heuristics to deal with the bottleneck and foldover problems, by utilizing extra memory. The bottleneck problem is dealt with by performing both an H-V, and a V-H process and then picking the best result.

Wolberg [Wolberg90] describes the use of polynomial forward differencing on each scanline to accelerate rendering, but does so without taking error tolerance into account when subdividing. As such, it seems to be a mere curiosity, without much of a practical application.

There are several reasons why the above methods are unsuitable for our intended applications.

Inefficient Use of Modern Computer and Memory Architectures

These algorithms were developed in an era when processor and memory cycle times were approximately equal, or at least of the same order of magnitude. Modern processors, with vector and superscalar architectures running with gigahertz clocks, have a much different ratio of memory to processor cycle times. Processor speeds have increased much faster than memory speeds, yielding a disparity of 1-2 orders of magnitude. In 2001, the ratio is between 40:1 and 100:1 [Moshovos01]. It is expected that this speed disparity will only get worse, and is expected to approach 1000:1 in a few years [Patt01].

Even though memory *speeds* have not kept up with processor speeds, memory *architectures* have advanced to partially compensate for this disparity. Unfortunately, the common multilevel caches only really help when accessing memory sequentially. There have been some specialized cache preload instructions introduced into modern processors, but they are difficult to use because they require pipeline programming (with *prime*, *pump* and possibly *flush* code segments), that increases code size and makes maintenance more difficult.

Regardless, the disparity in processor versus memory speeds imposes a severe penalty on algorithms that make use of incoherent memory access, requiring that we abandon the conventional wisdom that table lookups are fast (unless the tables are small and accessed frequently). It is advantageous to eliminate the use of a table if its values can be easily computed, thus freeing the cache for other things.

The Catmull-Smith, Paeth, Chen, and Wolberg89 algorithms are separable, requiring two or three passes. At least one of these passes is in the vertical direction, which is virtually guaranteed to *not* reside in the cache, thus stalling the processor for 100 cycles or so on every access. Even the horizontal pass, with coherent access, stalls when accessing a new cache line. Although this cost is amortized over all the memory in that cache line, it is still significant and will become even more so, e.g. if the 8 words in a cache line are accessed sequentially, their access take 12 cycles on the average if it takes 100 cycles to fill a cache line, and 125 cycles if it takes 1000 cycles to fill a cache line.

On modern computer architectures, then, it is preferable to perform the transformations in a single pass, thus avoiding the memory access penalty associated with multi-pass algorithms. Also, it is preferable to compute functions on the fly, rather than taking cache faults due to table lookup.

Constantly Varying Parameters Invokes a Significant Initialization Phase

The Wolberg89 algorithm makes use of large tables for computing the warping function, so is heavily penalized by memory access. Also, it requires a significant startup phase, in order to initialize the two large mapping tables. Typically, the time required for this initialization phase is at least an order of magnitude more than the time taken in the subsequent image-warping phase. If the warping parameters change in every frame, then it is a waste of time to construct the mapping table at all, and the Wolberg algorithm is less efficient than the naïve direct computation method. The Chen algorithm also has a startup phase, but its 1D tables are initialized quickly.

Loose Error Control

[Wolberg90] is the algorithm that comes closest to the techniques that we use, but it provides no quantifiable error control. In the two-pass algorithms, it is difficult to perform a numerical analysis, and it is expected that errors on the order of a pixel are not uncommon.

Our Algorithm

Our algorithm improves on the prior art in that it *requires no initialization of lookup tables*, it is *single pass*, the warping function is *computed on the fly* without the use of tables, it can achieve *any specified error tolerance* (we use 1/10 of a pixel), it has *no bottleneck problem*, and it naturally *produces subpixel accuracy* sufficient for interpolation. It only deals with surjective (i.e. onto) functions, though, so *foldover* needs to be resolved by splitting the warping function into multiple surjective domains (e.g. at silhouettes), resolving self-occlusion by the painter's algorithm [Foley90].

4 Problem Definition

The generalized problem investigated by this paper is as follows:

We would like to efficiently evaluate a nonlinear multivariate function repetitively at regularly spaced intervals along a curve in the domain. The function is assumed to be relatively complex, involving divisions and transcendental functions. Rather than

invoking a complex evaluation at each point, we would like to do complex functional evaluations at only a few points, and use efficient computations such as polynomial evaluations in between. In particular, we would like to use polynomial forward differencing.

If the function were univariate, it would be possible to run a nonlinear optimization offline to produce a set of Chebyshev (L_∞ optimum) polynomials [Ralston78] and subintervals that span the domain, approximating the function within a given error tolerance. This is the technique used in scientific and mathematical libraries.

However, since the function is multivariate, and not a tensor product function in general, then we cannot compute Chebyshev polynomials ahead of time. However, we *can* reduce the multivariate function to a univariate parametric one by restricting the domain to a parametric curve, e.g.

$$g(t;\mathbf{p}) = f\left(x_\mathbf{p}(t), y_\mathbf{p}(t)\right),$$

where $f(\cdot,\cdot)$ is a bivariate function, and $\left(x_\mathbf{p}(t), y_\mathbf{p}(t)\right)$ is a vector function of the free variable *t*, and fixed parameter vector p. The function $g(t;\mathbf{p})$ is univariate in *t*, with parameters p. An example of such a function is $\sin(\omega t)$, which can be considered a bivariate function in ω and *t*, or a univariate function of *t* with parameter ω.

The primary focus of this paper is the scan-conversion of a nonlinear deformation (warp) of a 2-dimensional image, as embodied by the bivariate vector function

$$u = U(x,y) \qquad v = V(x,y).$$

At a fixed value of *y*, we wish to evaluate both functions at regularly spaced points on an interval in *x*. The *x*-interval will be different, in general, from one scanline (i.e. fixed value of *y*) to the next.

The structure of a computer graphics scan-conversion algorithm takes the form of a very efficient inner loop surrounded by one or more outer loops. The canonical scan conversion loop in *C* looks like:

```
for (y = top; y <= bottom; y++) {   /* Y loop */
   left  = CalculateLeft( y);
   right = CalculateRight(y);
   dx = right - left + 1;
   for (x = left; dx--; x++) {      /* X loop */
      u = CalculateU(x, y);
      v = CalculateV(x, y);
      dstPixel[y][x] = GetSrcPixel(u, v);
   }
}
```

In this paper, we introduce another loop between the *x* and *y* loops, to loop between subintervals.

```
for (y = top; y <= bottom; y++) { /* Y loop */
   left  = CalculateLeft( y);
   right = CalculateRight(y);
   width = right - left + 1;
   for (x = left; width != 0; ) {  /* Interval loop */
      dx = CalculateInterval(x, y); /* error control*/
      SetupInterval(x, y, dx);     /* poly coeffs */
      if (dx > width) dx = width;
      width -= dx;
      for ( ; dx--; x++) {         /* X loop */
         u = CalculateUOnInterval(x, y); /* fwd diff */
         v = CalculateVOnInterval(x, y); /* fwd diff */
         dstPixel[y][x] = GetSrcPixel(u, v);
      }
   }
}
```

where `CalculateUOnInterval()`, `CalculateVOnInterval()` are very simple evaluation functions evaluated at every pixel, and `CalculateInterval()` and `SetupInterval()` have the same order of complexity as `CalculateU()` and `CalculateV()`, but are computed only once an interval.

Polynomials can be very efficient to evaluate at regular intervals by using forward differencing, as indicated by the following code for a cubic polynomial:

```
while(i--) { Do(u); u += du; du += ddu; ddu += dddu; }
```

There are, however, numerical accuracy problems with the higher order differences, so it is necessary to do a thorough numerical analysis to determine the accuracy needed in the computations. The required accuracy is a function of the maximum interval length.

Here is the problem, then: determine the set of subintervals and polynomial coefficients to approximate an arbitrary function $f(\cdot)$ over an interval *I*, with tolerance ε_{max}.

5 Functional Approximation with Polynomials

We take a computationally efficient approach to functional approximation, though it does not produce an optimum approximant under any measure: evaluate the function at regularly spaced subintervals between the endpoints of the interval, and produce the polynomial that interpolates those points. This has the form of :

$$A_n(x) = \begin{bmatrix} 1 & \dots & \left(\frac{x-L}{R-L}\right)^{n-1} \end{bmatrix} \mathbf{T}_n \begin{bmatrix} f(L) \\ \vdots \\ f(R) \end{bmatrix} \tag{5.1}$$

with the appropriate matrix $\mathbf{T}_n$ produced with Mathematica [Wolfram88] and given in Table 1.

The endpoints are always interpolated. For linear approximation, only the endpoints (0, 1) need to be evaluated. For quadratic interpolation, it is necessary to also evaluate the function at the midpoint (0, 1/2, 1). For cubic interpolation, the 1/3 and 2/3 points are needed, while for quartic interpolation, the 1/4/, 1/2, and 3/4 points are needed, as well as the endpoints.

6 Forward Differencing

Conversion from the power basis (given above) to the forward difference basis is accomplished with the appropriate matrix $\mathbf{D}_n$ from Table 1 as indicated in eq. 6.1.

$$\begin{bmatrix} F_o \\ \vdots \\ F_{n-1} \end{bmatrix} = \mathbf{D}_n \mathbf{T}_n \begin{bmatrix} f(L) \\ \vdots \\ f(R) \end{bmatrix} \tag{6.1}$$

7 Maximum Error of Approximation

The maximum approximation error, ε_n, takes the general form

$$\varepsilon_n(\Delta) = \left| \frac{f^{(n+1)}\left(x_n^*\right)}{K_n} \right| \Delta^{n+1} \tag{7.1}$$

where *n* is the degree of the interpolation, K_n is an degree-specific constant, $\Delta = R - L$ is the interval length, and $x_n^* \in [L, R]$ is the value of *x* at which the derivative attains its maximum value in the interval. The constants K_n are given in Table 1 for four degrees of approximation.

These expressions were not derived from the Taylor series, but by subtracting the approximation from the function and evaluating

n	Coeff. K_n	Forward Difference Matrix $\mathbf{D}_n$	Interpolation Matrix $\mathbf{T}_n$
1	8	$\begin{bmatrix} 1 & 0 \\ 0 & \alpha \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$
2	128	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \alpha & \alpha^2 \\ 0 & 0 & 2\alpha^2 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ -3 & 4 & -1 \\ 2 & -2 & 1 \end{bmatrix}$
3	$\frac{10368}{5}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha & \alpha^2 & \alpha^3 \\ 0 & 0 & 2\alpha^2 & 6\alpha^3 \\ 0 & 0 & 0 & 6\alpha^3 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix}$
4	$\frac{262144}{7}$	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \alpha & \alpha^2 & \alpha^3 & \alpha^4 \\ 0 & 0 & 2\alpha^2 & 6\alpha^3 & 14\alpha^4 \\ 0 & 0 & 0 & 6\alpha^3 & 36\alpha^4 \\ 0 & 0 & 0 & 0 & 24\alpha^4 \end{bmatrix}$	$\frac{1}{3}\begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ -25 & 48 & -36 & 16 & -3 \\ 70 & -208 & 228 & -112 & 22 \\ -80 & 288 & -384 & 224 & -48 \\ 32 & -128 & 192 & -128 & 32 \end{bmatrix}$

Table 1. Coefficients, Forward Difference and Interpolation Matrices for eqs. (5.1), (6.1), (7.1) and (8.1), where $\alpha = 1/N$.

it midway between the interpolating points of the approximation. The expressions derived thusly take the same form as that from the Taylor series, but the constants are different lar ger in general, so they provide a more conservative bound, though ill-behaved functions can still exceed this bound (because they achieve their maximum at another point), so we might call this the *near-maximum* error.

In general, we do not know where the point x^* is, unless we know something about the nature of the derivative. We do know, however, that the derivative has wild fluctuations, so we will typically regularize the derivative, remove singularities, and approximate it with a smooth, computationally efficient bounding function. If the domain is then split into monotonic partitions, then the maximum value of the derivative bounding function is attained at either end of the interval, or at the boundary between domain partitions.

In the warping functions we have investigated for rendering environment maps, two classes of functions have emerged:

(1) the bounding function is monotonic everywhere, and

(2) the bounding function attains a maximum at x=0, and is monotonically decreasing from there.

Both of these have been easy to deal with. More complex functions, especially those that are functions of more than two variables, may be difficult to characterize.

In a partition where the bounding function is monotonically decreasing, the maximum value is attained at the left end of the interval. When it is monotonically increasing, though, the maximum value is attained at the right end. But we do not know the right end of the interval, because we will choose it to meet a given error tolerance.

8 Determining the Maximum Interval Length

The maximum number of forward difference evaluations, N, is related to the maximum interval length, Δ, by the spacing, δ, between the evaluations:

$$N = \left\lfloor \frac{\Delta}{\delta} \right\rfloor,$$

where $\lfloor \cdot \rfloor$ is the *floor* function. The right end of the interval is related to the left by

$$R = L + N\delta .$$

Solving the error expressions above for Δ and then using that to find N, we arrive at

$$N_n = \left\lfloor \frac{1}{\delta} \sqrt{\frac{K_n \varepsilon_{max}}{\max\limits_{x \in [L,R]} \left| f^{(n+1)}(x) \right|}} \right\rfloor \tag{8.1}$$

where the constants K_n are given in Table 1. Here, we have explicitly written $\max\limits_{x \in [L,R]} \left| f^{(n)}(x) \right|$ to indicate that the maximum value of the derivative over the interval is to be used.

9 Evaluating the Derivative over an Interval

The general expression for the length of the interval to meet a given error tolerance is given by:

$$N = \left\lfloor \frac{1}{\delta} \left(K_n \frac{\varepsilon_{max}}{\max\limits_{x \in [L, L+N\delta]} \left| f^{(n)}(x) \right|} \right)^{\frac{1}{n}} \right\rfloor$$

where we have used $R = L + N\delta$ in the interval specification.

We see that N is *implicitly* defined by this equation, and therefore cannot be computed in closed form, in the general case. However, as we have seen in the last section, if the derivative (or preferably, its bound) is monotonically *decreasing* in the interval, then its maximum value is attained at the left end of the interval, and can be computed in closed form. When it is monotonically *increasing*, some form of iteration is necessary. Three algorithms come to mind:

(1) fixed-point iteration,

(2) secant iteration, and

(3) higher derivative extrapolation

Fixed Point Iteration

Fixed point iteration is straightforward. Starting with the interval length determined from the left derivative, a new interval length estimate is made from the previous one.

$$N_{i+1} = \left\lfloor \frac{1}{\delta} \left(K_n \frac{\varepsilon_{max}}{\left| f^{(n)}(L + N_i\delta) \right|} \right)^{\frac{1}{n}} \right\rfloor$$

Care needs to be taken to determine convergence, because the function is not smooth due to its quantization to integers. As a result, a *limit cycle* can occur, whereby the function oscillates back and forth between two values. The iteration should terminate when the value no longer decreases.

Secant Iteration

Given two estimates of the interval length (initially computed from $x_0 = L$ and $x_1 = L + M(x_0)\delta^\circ$), linearly interpolate to find a better right interval endpoint.

$$\overline{N}_{i+1} = \frac{x_i M(x_{i-1}) - x_{i-1} M(x_i)}{M(x_i) - M(x_{i-1})}$$

$$x_{i+1} = L + \overline{N}_{i+1}\delta$$

This should also be checked for limit cycles in the convergence.

Higher Derivative Extrapolation

This makes use of the Taylor expansion around *L* to approximate the derivative at *R*, yielding the equation

$$N = \frac{1}{\delta f^{(n+1)}(x)}\left(\frac{K_n \varepsilon}{\delta^n N^m} - f^{(n)}(x)\right),$$

which is most easily solved by iteration, as suggested by this equation. The main problem with this method is potential overshoot if there are inflection points in the n^{th} derivative. The other two methods are much better behaved.

10 Regularizing the Interval Length Function

Higher derivatives have inherently wild fluctuations, and frequently pass through zero. The reciprocal of the n^{th} root of the derivative is used in the interval length computation, resulting in frequent infinite singularities. For this reason, the derivative should be replaced by a better-behaved function. Typically, we do this in several steps:

(1) regularization
(2) removal of singularities
(3) composition of *u* and *v* bounds
(4) graphical analysis
(5) bounding approximation
(6) bounds check

Regularization

Many derivatives have polynomial components, with both positive and negative coefficients. Making all of the coefficients positive eliminates most of the singularities and ill behavior.

Removal of Singularities

Of the singularities that remain, most appear as *x* or *y* factors in the numerator of the derivative. At zero, their reciprocal blows up to infinity. Replacing these by expressions of the form $\sqrt{a^2 + x^2}$ will remove those singularities. However, it is necessary to graphically interact with this function in order to determine appropriate values of *a* that preserve the shape.

Composition of *u* and *v* Bounds

There are actually two interval length surfaces: one for *u* and one for *v*. The minimum for the two surfaces then becomes the composite surface used in the real-time rendering.

Graphical Analysis

Interacting with a 3D graph of the interval length function is necessary in all steps of the regularization process. However, it appears as a separate step because here is where most of the time is spent. In this phase, the function is inspected at various scales and viewpoints, and 2D cross-sectional plots are produced in order to gain a better understanding of the function. It is important to answer the questions:

(1) What are the salient features of the function?
(2) What are the minima?
(3) What is the behavior at the origin and the coordinate axes?
(4) What is the asymptotic behavior?
(5) What is the principal domain? What region of parameter space do you expect to be used most frequently? Are there any forbidden zones ?
(6) Do you care if there is some inefficiency outside of the principal domain?
(7) Do you care if there is some distortion outside of the principal domain?

Once you gain an intimate knowledge of the function, you can begin to think about ways to approximate it with a simpler, computationally efficient lower-bounding function.

Bounding Approximation

Since the interval length function is not going to be evaluated in the innermost loop, it can be moderately complex, including

(1) division, multiplication, subtraction, addition,
(2) square roots,
(3) rational polynomials,
(4) B-spline surfaces

If needed, some trigonometry, exponentials, or powers may be used, but these are somewhat expensive, and can possibly be approximated using argument reduction and rational polynomial approximation.

The asymptotic behavior tends to be fractional powers (due to the n^{th} root), but it isn t necessary to model this behavior, because it is outside of the principal domain. A linear, square root, or constant function can suffice as long as it is a lower bound. Keep in mind that the probability of hitting the asymptotic region is small, and you can afford to evaluate it less efficiently. Usually, the intervals in the asymptotic region are much larger than that in the principal domain, so would be more efficiently computed than in the principal domain even with sloppy bounds.

Bounds Check

The final step is to check that the bounding surface really acts as a bound for the original reciprocal n^{th} derivative.

Example

A texture-mapped sphere will be used as an example. The fundamental equations for rendering a central view of a sphere texture-mapped with an equirectangular projection image (latitude-longitude) can be reduced (after dividing *x* and *y* by *z*) to:

$$u(x, y) = \arctan(x), \qquad v(x, y) = \arctan\left(\frac{y}{\sqrt{1 + x^2}}\right)$$

In order to render this with cubic forward differencing, we need fourth derivatives. The fourth derivative of *u*, and a smoother upper bound are given by

$$u_{xxxx}(x) = \frac{24x(1 - x^2)}{(1 + x^2)^4}, \qquad \hat{u}_{xxxx}(x) = \frac{6}{(1 + x^2)^2}$$

The reciprocal fourth root of this is used to determine interval length. These are plotted in Figure 1 to get a good feel for the quality of the bound. Note that although the derivative is asymptotically superlinear ($O\left(|x|^{\frac{5}{4}}\right)$), the bound is linear, and

remains a bound outside of the principal domain of [—5,5]. This principal domain was determined by two means: (1) varying the viewing parameters through typical and extreme values to find the range of these intermediate parameters, and also (2) looking at the plots to identify interesting regions as opposed to asymptotic regions.

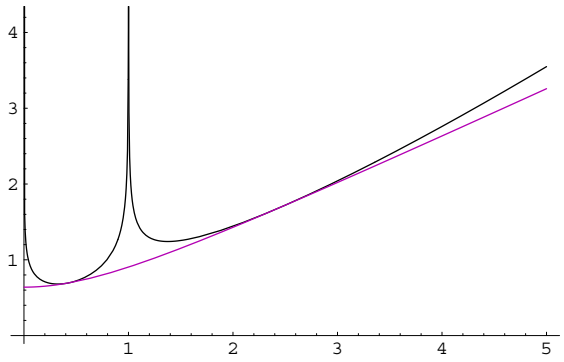

Figure 1. Reciprocal nth root of derivative, and regularized bound $\left(6^{-1/4}\right)\sqrt{1+x^2}$.

Finding bounds for such derivatives is somewhat of an art. Changing all minus signs to plus is a technique that frequently works, although more work was needed in the above example.

Finding the minimum of this function (or maximum of the regularized derivative) over an interval is not hard because of the monotonicity of this function: just evaluate it at the interval endpoint closest to the origin.

In this case, the *u* function is univariate, so it is easy to verify bounds graphically.

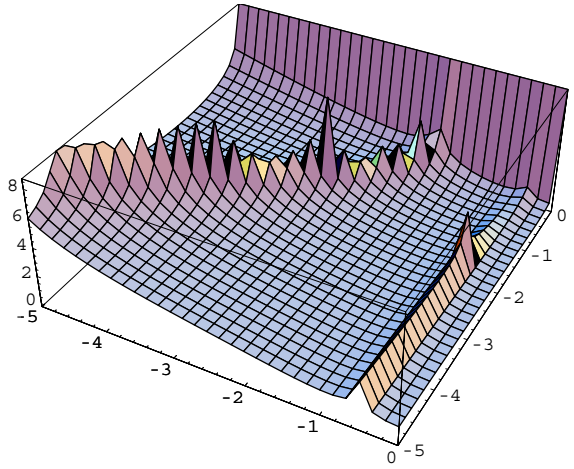

Figure 2. Reciprocal root of derivative: $\left[f^{(4)}(x)\right]^{-\frac{1}{4}}$.

However, the *v* function is bivariate. Its fourth partial derivative with respect to *x* is

$$-3y\frac{\left(\begin{array}{l}-8x^{10}+8x^8y^2-\left(1+y^2\right)^2\left(3+y^2\right)+x^6\left(45+52y^2\right)+\\ x^4\left(55+73y^2+16y^4\right)+x^2\left(15+22y^2+11y^4+4y^6\right)\end{array}\right)}{\left(1+x^2\right)^{\frac{7}{2}}\left(1+x^2+y^2\right)^4}$$

and its reciprocal fourth root is shown in Figure 2.

We regularize this by making all signs positive and removing the singularity at *y*=0 by replacing *y* with $\sqrt{1/4+y^2}$, yielding

$$3\cdot\sqrt{\tfrac{1}{4}+y^2}\,\frac{\left(\begin{array}{l}8x^{10}+8x^8y^2+\left(1+y^2\right)^2\left(3+y^2\right)+x^6\left(45+52y^2\right)+\\ x^4\left(55+73y^2+16y^4\right)+x^2\left(15+22y^2+11y^4+4y^6\right)\end{array}\right)}{\left(1+x^2\right)^{\frac{7}{2}}\left(1+x^2+y^2\right)^4}$$

and plot its reciprocal fourth root in Figure 3.

Next, we compose the *u* and *v* bounds together by taking the minimum of the two, and plot it in Figure 4. In this case, the *u* bounds dominate, yielding the composite interval bounding function

$$f(x)=\left(6^{-1/4}\right)\sqrt{1+x^2}\ .$$

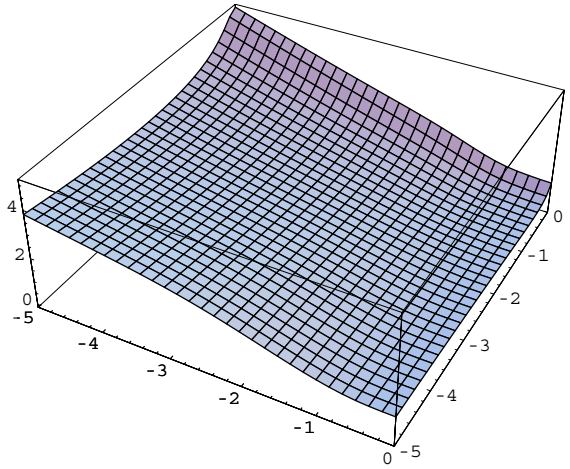

Figure 3. Regularized bound for $\left[f^{(4)}(x)\right]^{-\frac{1}{4}}$.

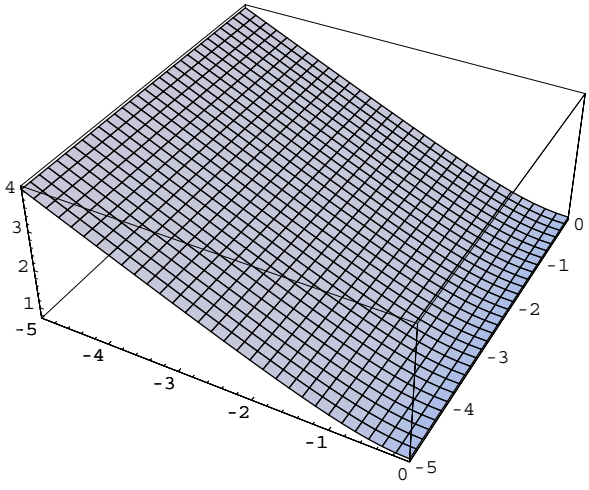

Figure 4. Composite bound, minimum of *u* and *v* bounds Min(Figure 1, Figure 3).

11 Discussion

Summary of the Algorithm

First, a class of rendering problem is identified.

Computational Accuracy

When evaluating the forward difference coefficients, keep in mind that the powers of α get very small. Resist the temptation to concatenate the α matrix with the power basis matrix for the higher order polynomials. For optimum accuracy, convert the function samples to the power basis first, then multiply by the α matrix to yield the forward difference coefficients, performing the dot products in doubled precision, and accumulating the summands in increasing magnitude.

As long as the ratio between adjacent order differences is not too large, the forward differences can be performed in single precision floating point, which has 24-bit precision. If we split the precision between two adjacent order differences (yielding a 12-bit overlap), then $2048=2^{11}$ steps (almost $4096=2^{12}$) can be taken before $^1/_2$ LSB computational error propagates between the difference orders (rationalized as 1 bit initialization error + 11 bits accumulation error + 11 bits clean value + 1 spare/sign bit). If the initialization error is higher due to sloppy computations, then the maximum interval size will need to be reduced, roughly 2 bits of interval size for every 1 bit of initialization error, due to the split of the bits between adjacent orders of differences. For example, suppose that the initialization of the differences is accurate to 20 bits (i.e. the 21st bit is in error), then the maximum interval length is about $1024=2^{10}$.

The forward differences can be carried out in fixed point; however, it is usually necessary to shift between orders of differences. For example,

```
u    += du   >> n0;
du   += ddu  >> n1;
ddu  += dddu >> n2;
dddu += ddddu;
```

The shift values depend upon the interval length as $n_i \approx \log_2(\mathcal{N})$, but also to some extent on the function itself. Adequate numerical analysis for a particular range of function can usually determine a maximum allowable interval length and constant values for the shifts, if desired, though shift-tailoring for each interval can produce longer accurate intervals, potentially even longer than that available in single-precision floating-point, since native fixed-point is widely accurate to 32 bits, and even 64 bits on some machines, whereas single precision IEEE floating point only has 24 bits of mantissa.

Order of Approximation

Generally, a higher order polynomial yields larger intervals resulting in higher performance than a smaller one, at the expense of more computation to generate the forward difference coefficients. Also, a high order inner loop increment is more involved and can take more time, unless all variables can fit into registers and/or vector instructions are used. High order polynomials are more sensitive to noise and computational errors as well, so interval length may be limited due to computational accuracy.

Applicability to Graphics Hardware

Forward differencing machines are particularly easy to implement in hardware, especially in fixed-point. Evaluating the interval length and forward difference coefficients can be done in a more general-purpose co-processor, in a pipeline fashion.

Results

This technique was used to render cubic environment maps [Greene86b] under interactive camera control, using fixed-point quadratic forward differencing in standard C without the use of specialized SIMD vector instructions. The tolerance ε_{max} was set to 1/10 of a pixel in eq. (8.1), and interval lengths were frequently greater than 100. It was not necessary to have a guard band around the image to prevent memory faults with this tolerance. We were able to render a 1000x700 window of 16 bit pixels point-sampled at unit zoom 30 frames per second on a 500 MHz G3 and 50 frames per second on a 1.3 GHz Pentium 4. Bilinear interpolation reduces this to 10 and 15 frames per second, respectively. A 320x240 window yielded greater than 300 frames per second. This yielded a 3X performance increase over a highly tuned scan converter using forward differencing and two perspective divisions per pixel.

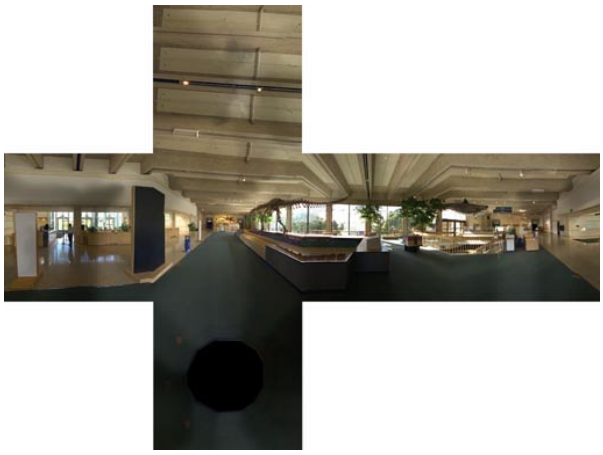

Figure 5. Cubic environment map.

Figure 6. Novel view rendered with this algorithm from the cubic environment map.

12 Future Work

Anti-Aliasing

It should be straightforward to enhance the algorithm to accommodate anti-aliasing. In our implementation, we only perform interpolation, which improves the quality when zooming in, but yields jaggies when zooming out greater than 2:1. Anti-aliasing is different than interpolation in that the kernel size for interpolation is fixed, whereas that for anti-aliasing increases in size in inverse proportion to the scale factor, when that scale factor is less than 1. The scale factor is embodied by the Jacobian (partial derivatives) of the coordinate transformation [Heckbert86], which

can be used to implement elliptically weighted average filtering [Greene86a]:

$$\mathbf{J} = \begin{bmatrix} u_x & v_x \\ u_y & v_y \end{bmatrix}$$

The x component of the Jacobian (the first row) can be approximated by the first partial difference in the forward differencing. The y component needs to have its own approximation. Alternatively, the norm of the Jacobian can be approximated, interpolated across the screen, and used as an index to a MipMap [Williams83].

Better Approximation Polynomials

Chebyshev approximation generally results in half the error of our approximation. It is desirable to efficiently find Chebyshev-like approximations to increase (double) the interval length.

Automatic Bounds

A lot of effort would be saved with an automatic method to produce derivative bounding functions.

Real-Time Video Special Effects

Since this algorithm is essentially limited in speed by memory access to the source image, it seems natural to use for video special effects, such as page curls, without the use of special-purpose hardware.

Trivariate and Higher Dimensional Warping Functions

Our experience has been limited to bivariate warping functions, although the algorithm lends itself to higher-dimensional domains, albeit with more time for analysis and design.

13 Acknowledgements

Specialized texture-mapping algorithms for simple primitives have been developed at Apple Computer since 1992, when Eric Chen and Gavin Miller pioneered specialized rendering algorithms for cylinders, spheres and cubes to allow interactive adjustment of viewing parameters. Gavin Miller accelerated rendering of cubic environment maps by linearly interpolating between every 8 perspective divisions. Ali Sazegari developed the polynomial coefficients used for rendering cylindrical environment maps in QuickTime¤ VR. Mark Wheeler developed error expressions for rendering cubic environment maps. Apple Computer incorporated the error-controlled piecewise polynomial rendering into the cubic environment renderer in QuickTime¤ VR.

References

Beier92 Beier, Thaddeus and Shawn Neely, Feature-Based Image Metamorphosis, *Proc. SIGGRAPH 92*, vol. 26, no. 2, pp. 35-42, 1992.

Catmull80 Catmull, Edwin and Alvy Ray Smith, 3-D Transformations of Images in Scanline Order, *Proc. SIGGRAPH 80*, pp. 279-285, July 1980.

Chen95a Chen, Shenchang Eric and Gavin Miller, Cylindrical to Planar Image Mapping using Scanline Coherence, U.S. Patent no. 5,396,583, March 7, 1995.

Chen95b Chen, Shenchang Eric, QuickTime¤ VR An Image-Based Approach to Virtual Environment Navigation, *Proc. SIGGRAPH 95*, pp. 29-38, 1995.

Debevec96 Debevec, Paul, C. Taylor and Jitendra Malik, Modeling and Rendering Architecture from Photographs: A Hybrid Geometry and Image Based Approach, *Proc. SIGGRAPH 96*, pp. 11-20, 1996.

Foley90 Foley, James, Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.

Greene86a Greene, Ned and Paul Heckbert, Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter, *IEEE Computer Graphics and Applications*, vol. 6, no. 6, pp. 21-27, June 1986.

Greene86b Greene, Ned, Environment Mapping and Other Applications of World Projections, *IEEE Computer Graphics and Applications*, vol. 6, no. 11, pp. 21-29, November 1986.

Heckbert86 Heckbert, Paul, Survey of Texture Mapping, *IEEE Computer Graphics and Applications*, vol. 6, no. 11, pp. 56-67, November 1986.

Horry97 Horry, Youichi, Ken-ichi Anjyo, Kiyoshi Arai, Tour Into the Picture: Using a Spidery Mesh Interface to Make Animation from a Single Image, *Proc. SIGGRAPH 97*, pp. 225-232, 1997.

McMillan95 McMillan, Leonard and Gary Bishop, Plenoptic Modeling: An Image Based Rendering System, *Proc. SIGGRAPH 95*, pp. 39-46, 1995.

Moshovos01 Moshovos, Andreas and Gurindar Sohi, Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling, *Proc. IEEE*, vol. 89, no. 11, pp. 1560-1575, November 2001.

Oliveira00 Oliveria, Manual M., Gary Bishop, and David McAllister, Relief Texture Mapping, *proc. SIGGRAPH 2000*, pp. 359-368.

Paeth86 Paeth, Alan, A Fast Algorithm for General Raster Rotation, *Proc. Graphics Interface 86*, pp. 77-81, 1986.

Patt01 Patt, Yale, Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution, *Proc. IEEE*, vol. 89, no. 11, pp. 1553-1559, November 2001.

Ralston78 Ralston, Anthony and Philip Rabinowicz, *A First Course in Numerical Analysis*, McGraw-Hill, 1978.

Seitz96 Seitz, Steven and Charles Dyer, View Morphing, *Proc. SIGGRAPH 96*, pp. 21-30, 1996.

Shum99 Shum, Heung-Yeung and Li-Wei He, Rendering with Concentric Mosaics, *Proc. SIGGRAPH 99*, pp. 299-306, 1999.

Smith87 Smith, Alvy Ray, Planar 2-Pass Texture Mapping and Warping, *Proc. SIGGRAPH 87*, pp. 263-272.

Torberg96 Torberg, J. and J. Kajiya, Talisman: Commodity Realtime 3D Graphics for the PC, *Proc. SIGGRAPH 96*, pp. 353-363, 1996.

Williams83 Williams, Lance, Pyramidal Parametrics, *Proc. SIGGRAPH 83*, vol. 17, no. 3, pp. 1-11, 1983.

Wolberg89 Wolberg, George and Terrance Boult, Separable Image Warping with Spatial Lookup Tables, *Proc. SIGGRAPH 89*, pp. 369-378, 1989.

Wolberg90 Wolberg, George, *Digital Image Warping*, IEEE Computer Society Press, 1990.

Wolfram88 Wolfram, Stephen, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, 1988. <http://www.wolfram.com>