

RECURSIVE TEXTURE MAPPING

WIM COUWENBERG

ABSTRACT. Mapping a texture image onto a flat spatial surface and subsequently projecting the image into screen space (such as a computer display) by a central projection results in a fractional linear transformation. Hence the inverse texture map is fractional linear as well. The divisions involved in this inverse can be prohibitively expensive for real time applications. This paper presents a method that computes the texture coordinates by a simple recursive formula avoiding divisions. The texture map thus obtained only introduces very little distortion, is very efficient on mainstream hardware and is easy to implement.

1. TEXTURE MAPPING BASICS

In this section we will very briefly go over some texture mapping basics to get a feel for the kind of transformations involved. As in many applications we will restrict ourselves to texture mapping of triangles. Rendering a triangle involves three different spaces: *texture space*, *object space* and *screen space*. Texture space is 2-dimensional and defines the texture image to be mapped onto our given triangle. Object space is the 3-dimensional space in which the actual triangle is positioned. Screen space is 2-dimensional and can be thought of as embedded in object space. The mapping from object space onto screen space is a central projection as seen from some *view point* in object space. We will call the plane that passes through the view point and is parallel to screen space the *base plane*. Points on the base plane can not be mapped into screen space (in fact, projectively speaking, the base plane represents the “line at infinity” in screen space). We will assume that our triangle does not pass through the base space and is situated on the same side of it as the screen space. This means that the triangle is “visible” in its entirety from the view point. For convenience we also introduce *triangle space* as the hyperplane in object space that is spanned by the triangle.

Let (T_1, T_2, T_3) be the vertices in texture space of our triangular texture pattern. Let (P_1, P_2, P_3) and (Q_1, Q_2, Q_3) be the vertices of the triangle in object space and screen space respectively. Let $(\lambda_1, \lambda_2, \lambda_3)$ be affine coordinates in texture space, non-negative on the triangular texture, such that $\lambda_i = 1$ corresponds with vertex T_i . Similarly, let (μ_1, μ_2, μ_3) and (η_1, η_2, η_3) be affine coordinates in triangle space and screen space, non-negative on the (projected) triangle, such that $\mu_i = 1$ and $\eta_i = 1$ correspond to P_i and Q_i respectively.

In these coordinates the texture map that maps T_i to P_i is now simply the *identity*

$$(1.1) \quad (x_1, x_2, x_3) \mapsto (x_1, x_2, x_3).$$

The central projection of object space onto screen space also takes a rather simple form in these affine coordinates. Let z_1, z_2, z_3 be the respective distances

of the vertices P_1, P_2, P_3 to the base plane (all positive by assumption). Then the central projection is expressed by:

$$(1.2) \quad (x_1, x_2, x_3) \mapsto \left(\frac{x_1 z_1}{d}, \frac{x_2 z_2}{d}, \frac{x_3 z_3}{d} \right), \quad d = x_1 z_1 + x_2 z_2 + x_3 z_3.$$

This can be readily verified as the denominator $d = x_1 z_1 + x_2 z_2 + x_3 z_3$ is simply the distance of the point (x_1, x_2, x_3) to the base plane and P_i must map to Q_i . The inverse mapping of screen space into triangle space is given by a very similar formula, namely:

$$(1.3) \quad (x_1, x_2, x_3) \mapsto \left(\frac{x_1/z_1}{d}, \frac{x_2/z_2}{d}, \frac{x_3/z_3}{d} \right), \quad d = x_1/z_1 + x_2/z_2 + x_3/z_3.$$

This can be verified by a straightforward computation. Note that each z_i just got replaced by its reciprocal $1/z_i$. Because mapping triangle space onto texture space is the identity in affine coordinates, formula 1.3 also describes the inverse texture map of screen space into texture space. This inverse texture map is not defined on the ‘‘horizon’’ in screen space where the denominator d vanishes. If all distances z_i are equal (i.e. if the triangle is parallel to base plane) then both formula 1.2 and 1.3 reduce to the identity. In this case all transformations are affine. In all other cases the mapping to and from screen space will be fractional.

In computer graphics applications it is common to render a triangle per scan line. Scan lines will usually be oriented horizontally or vertically. In affine screen coordinates such a line is parameterized by

$$(1.4) \quad t \mapsto (x_1 + t y_1, x_2 + t y_2, x_3 + t y_3)$$

where (x_1, x_2, x_3) is some base point on the scan line and (y_1, y_2, y_3) is an offset vector such that $y_1 + y_2 + y_3 = 0$. Substituting this into the inverse texture map 1.3 yields a texture parameterization of the form

$$(1.5) \quad t \mapsto \left(\frac{a_1 t + b_1}{c t + d}, \frac{a_2 t + b_2}{c t + d}, \frac{a_3 t + b_3}{c t + d} \right)$$

for some parameters a_i, b_i, c, d such that $a_1 + a_2 + a_3 = c$ and $b_1 + b_2 + b_3 = d$. Observe that all denominators are the same but the numerators are generally different. In practice the parameter t will run over some interval of integers, representing a contiguous run of pixels on a scan line.

Eliminating the division from a fractional linear function running over an interval of integers is exactly the key to the implementation of fast texture mapping algorithms.

2. RECURSIVE APPROXIMATION

Based on the observations from the texture mapping formulas we will concentrate on general fractional linear functions

$$(2.1) \quad f: n \mapsto \frac{a n + b}{c n + d}$$

where n runs over some interval $\{0, 1, \dots, m\}$ of integers. We can restrict ourselves to the case where both f itself and its denominator $c n + d$ are positive over this interval, as these are basically an affine texture coordinate and a denominator from 1.3 respectively. Moreover, we will assume that the rendering of the scan line starts at the side of the triangle that is *closest* to the base plane. So reverse the rendering

orientation on the scan line if necessary. In terms of parameters of the function f this translates into $c \leq 0$.

We are looking for a way to compute, or rather approximate, the sequence $f(0), f(1), \dots, f(m)$ by using computationally cheap addition and multiplication operations only¹. The coordinates in texture space all share the same denominator. Therefore the following scheme may be advantageous: compute the numerators for each coordinate by a simple progressive scheme (at the cost of one addition per coordinate) and then multiply by the reciprocal of the common denominator. This motivates to narrow the form of f down still further and only consider functions

$$(2.2) \quad f: n \mapsto \frac{1}{cn + d}.$$

The following sections will assume f to be of this form and, still, $c \leq 0$ and $cn + d > 0$ for $n \in \{0, \dots, m\}$. The idea is to find a polynomial in two variables $p(x, y)$ of small degree such that

$$(2.3) \quad f(n + 1) \approx p(n, f(n))$$

is a “good” approximation in some sense. Of course p will depend on the particular values of c, d and possibly of m as well. Then we proceed as follows. Compute the sequence $f_0, f_1, f_2, \dots, f_m$ by the recursion

$$(2.4) \quad f_0 = f(0), \quad f_{n+1} = p(n, f_n).$$

that only involves the exact computation of the first value $f(0)$. Hopefully the sequence thus obtained is a “good” approximation for the sequence of exact values throughout the entire interval. One could even hope for an *exact* recursion formula for some polynomial p , but such a recursion can not exist by the following lemma.

Lemma 2.1. *If the function f is not constant (i.e. $c \neq 0$) then there does not exist any polynomial $p(x, y)$ such that $f(n + 1) = p(n, f(n))$ for all $n \in \mathbb{Z}$.*

If such a recursion would exist it would hold for all $n \in \mathbb{R}$ because it is an equality between rational functions. However, $f(x + 1)$ has a pole only at $x = -d/c - 1$ whereas $p(x, f(x))$ can have a pole at most at $x = -d/c$. \square

Note that there *is* an exact recursion if n is only taken from a finite interval. Simple Lagrange interpolation of the values of f will do. However, the degree of p would be much too high and its coefficients too complex to be of any use. Before considering some candidates for recursive approximation let us introduce the inverse F of f , explicitly given by:

$$(2.5) \quad F: n \mapsto \frac{1 - dn}{cn}.$$

So $F(f(n)) = f(F(n)) = n$ for all n where these expressions are well-defined.

3. EULER PREDICTION

The first and maybe most obvious candidate for recursive approximation is Euler prediction. Set $y = f(x)$ for some real $x \in [0, m - 1]$ and take h such that $|h| \leq 1$. Then $|cy| < 1$ and

$$(3.1) \quad f(x + h) = \frac{y}{1 + hcy} = y - hcy^2 + h^2 c^2 y^3 - \dots$$

¹Fast multiplication may need support from specific hardware features. Addition is fast on all hardware.

Breaking off this series at the j -th power of h and setting $h = 1$ results in the Euler predictor of order j . We will only consider the first order and second order predictors

$$(3.2) \quad f(x+1) \approx y - cy^2$$

$$(3.3) \quad f(x+1) \approx y - cy^2 + c^2y^3.$$

These predictors suggest to use the recursion polynomials

$$(3.4) \quad p(x, y) = p_1(y) = y - cy^2, \quad \text{and}$$

$$(3.5) \quad p(x, y) = p_2(y) = y - cy^2 + c^2y^3$$

respectively that do not even depend on x and d at all! Let us consider the first order Euler predictor and hence choose $p_1(y)$ as our recursion polynomial. The rendering error for our recursive scheme can easily be estimated.

Theorem 3.1. *Compute the sequence f_0, \dots, f_m as in 2.4 using the first order Euler predictor $p_1(y)$. Then for each $n \in \{0, \dots, m\}$ the progressive error measured in screen space pixels, $n - F(f_n)$, is bounded by:*

$$(3.6) \quad 0 \leq n - F(f_n) \leq -\log\left(1 + \frac{cn}{d-c}\right).$$

At $n = 0$ we have $n - F(f_n) = -F(f(0)) = 0$ so the lower bound holds in this case. For $x \in [0, m-1]$ and $y = f(x)$ we compute the error contribution for a single step:

$$(3.7) \quad x+1 - F(p_1(y)) = \frac{-c}{c(x-1)+d} \geq 0.$$

This shows that our prediction will fall a bit short in each step. Now by induction and formula 3.7 we get

$$(3.8) \quad n+1 - F(f_{n+1}) = n+1 - F(p_1(f_n)) \geq F(f_n) + 1 - F(p_1(f_n)) \geq 0$$

which proves the lower bound for the progressive error. The error can be bounded from above by a summation of error estimates as in

$$(3.9) \quad \begin{aligned} n - F(f_n) &= n + \sum_{j=0}^{n-1} (F(f_j) - F(f_{j+1})) \\ &= n + \sum_{j=0}^{n-1} (F(f_j) - F(p_1(f_j))) \\ &= \sum_{j=0}^{n-1} \frac{-c}{cF(f_j) + d - c} \leq \sum_{j=0}^{n-1} \frac{-c}{cj + d - c}. \end{aligned}$$

The terms of the latter sum are increasing in j so this sum can be bounded from above by integration:

$$(3.10) \quad n - F(f_n) \leq \int_{x=0}^n \frac{-c}{cx + d - c} = -\log\left(1 + \frac{cn}{d-c}\right).$$

This proves the upper bound for the progressive error. \square

Note that for $c = 0$ the error bound becomes 0 which is consistent with the fact that f is constant in this case. As an example, consider the functions

$$(3.11) \quad f(n) = \frac{1}{-n + 120}, \quad F(n) = 120 - \frac{1}{n}$$

for $n \in \{0, \dots, 100\}$. The recursion polynomial in this case becomes $p_1(y) = y + y^2$. Computing the series f_0, \dots, f_{100} gives $f_{100} \approx 0.0461$, which is about 8 percent off the exact value of $f(100) = 0.05$. However, the maximum error in screen space equals $100 - F(f_{100}) \approx 1.692$ which means that the approximation for f lags less than two screen pixels behind over the entire stretch of the scan line. In particular the approximation procedure will *never* overshoot the texture but rather “stretch” it a few pixels. Note that theorem 3.1 reports an error upper bound of $\log(121/21) \approx 1.751$ in this case, which seems accurate enough.

An analogue of theorem 3.1 can be formulated for the second order Euler predictor $p_2(y)$ as defined by equation 3.5 as well. In fact, the second order predictor turns out to function remarkably well.

Theorem 3.2. *Compute the sequence f_0, \dots, f_m as in 2.4 using the second order Euler predictor $p_2(y)$. Then for each $n \in \{0, \dots, m\}$ the progressive error measured in screen space pixels, $n - F(f_n)$, is bounded by:*

$$(3.12) \quad 0 \leq n - F(f_n) \leq \frac{4c^2 n}{(2d - c + 2cn)(2d - c)} < 2.$$

The proof follows the same lines as the proof of theorem 3.1. The single step error at $y = f(x)$ for $x \in [0, m - 1]$ now becomes

$$(3.13) \quad x + 1 - F(p_2(y)) = \frac{4c^2}{(2d - c + 2cx)^2 + 3c^2/4} \geq 0$$

which leads to the lower bound $0 \leq n - F(f_n)$ by the same inductive argument. The upper bound of the progressive error can now be estimated by the following integration:

$$(3.14) \quad n - F(f_n) \leq \int_{x=0}^n \frac{4c^2}{(2d - c + 2cx)^2 + 3c^2/4} \leq \int_{x=0}^n \frac{4c^2}{(2d - c + 2cx)^2} = \frac{4c^2 n}{(2d - c + 2cn)(2d - c)}.$$

Now $n \leq m < -d/c$ and substituting $n = -d/c$ in the upper bound obtained in 3.14 leads to the universal upper bound of 2 that does not even depend on *any* parameter. \square

Applying second order Euler prediction to the example function from 3.11 leads to the recursion polynomial $p_2(y) = y + y^2 + y^3$. Using this polynomial in recursive approximation for f gives $f_{100} \approx 0.0499$ which is about a fifth percent off the real value. The error in screen space equals $100 - F(f_{100}) \approx 0.0393$, or about a 25th part of a pixel. Theorem 3.2 reports an error bound of approximately 0.0405 which is again quite accurate.

4. NEWTON-RAPHSON ITERATION

If $y = f(n + 1)$ then $F(y) = n + 1$. A solution to the latter equation can be approximated by Newton-Raphson iteration. Start at some initial value y and

perform a single iteration to obtain the estimate

$$(4.1) \quad f(n+1) \approx y - \frac{F(y) - n - 1}{F'(y)} = 2y - (cn + c + d)y^2.$$

This suggests to use the recursion polynomial

$$(4.2) \quad p(x, y) = 2y - (cx + c + d)y^2$$

in our approximation procedure. Note that if $y = f(x)$ then the Newton predictor $p(x, y)$ coincides with the first order Euler predictor $p_1(y)$. Although the polynomial $p(x, y)$ is of degree three, the coefficient of the y^2 term in 4.2 is in fact an arithmetic progression as x ranges over $\{0, \dots, m-1\}$. This makes the Newton predictor even simpler and more efficient to implement. More importantly, the approximations obtained by the Newton predictor are excellent, as the following theorem shows.

Theorem 4.1. *Compute the sequence f_0, \dots, f_m as in 2.4 using the Newton predictor $p(x, y)$. Then for each $n \in \{0, \dots, m\}$ the progressive error measured in screen space pixels, $n - F(f_n)$, is bounded by:*

$$(4.3) \quad 0 \leq n - F(f_n) \leq \min\left(1, \frac{-c}{cn + d}\right).$$

If $c = 0$, i.e. f is constant, then the progressive error will be zero. So assume $c < 0$. Again the proof is based on the expression for the single step error in screen space. Let $x \in [0, m-1]$, $h \in [0, 1]$ and set $y = f(x-h)$. Then

$$(4.4) \quad x + 1 - F(p(x, y)) = \frac{-c(h+1)^2}{cx + d - 2hc - c} > 0.$$

The lower bound from equation 4.3 follows by a similar inductive argument as in the proof of theorems 3.1 and 3.2. On the other hand, because $x \leq m-1$ we also have $cx + d > -c$ and hence the bounds

$$(4.5) \quad 0 < x + 1 - F(p(x, y)) < \frac{h+1}{2} \leq 1.$$

If $f_j = f(j-h)$ for some $h \in [0, 1]$ — meaning that f_j lags behind h pixels in screen space — and h' is defined as $j+1 - F(f_{j+1})$ then equation 4.5 shows that $0 < h' < 1$. Now $f_{j+1} = f(j+1-h')$ so f_{j+1} also lags behind at most one screen pixel. As $f_0 = f(0)$ we conclude by induction that $n - F(f_n) \leq 1$ for all $n \in \{0, \dots, m\}$. Remains to show that this bound can be sharpened as stated.

Fix some $n \leq m$ and define the function λ on the real interval $[0, 1]$ by

$$(4.6) \quad \lambda: h \mapsto \frac{-c(h+1)^2}{cn + d - 2c(h+1)}.$$

Then $0 < \lambda(h) < 1$ and λ is strictly increasing on the interval. By an inductive argument and equation 4.4 it follows that for $j \leq n$ the inequality

$$(4.7) \quad j - F(f_j) \leq \lambda^j(0)$$

holds, where λ^j stands for the j -fold iteration of λ . Now the sequence of iterations $\lambda^j(0)$ for $j = 0, 1, 2, \dots$ increases to the fixed point h_{fix} of λ on the interval $[0, 1]$. In particular $n - F(f_n) < h_{\text{fix}}$. A straightforward computations shows

$$(4.8) \quad h_{\text{fix}} = \frac{cn + d - \sqrt{(cn + d)^2 + 4c^2}}{2c} \leq \min\left(1, \frac{-c}{cn + d}\right).$$

This proves the upper bound for the progressive error. \square

Applying Newton prediction to the example function from 3.11 leads to the recursion polynomial $p(x, y) = 2y + (x - 119)y^2$. Using this polynomial in recursive approximation for f gives $f_{100} \approx 0.0499$ and f_{100} is about a quarter percent off the real value. The error in screen space equals $100 - F(f_{100}) \approx 0.0496$, or about a 20th part of a pixel. Theorem 4.1 reports an error bound of 0.05 that is again quite accurate.

5. GENERAL FRACTIONAL LINEAR FUNCTIONS

To exploit the fact that the coordinates in the inverse texture map share a common denominator we restricted ourselves to fractional functions of the form

$$(5.1) \quad f: n \mapsto \frac{1}{cn + d}.$$

However, the recursive approximation methods that we discussed for this situation can be extended to the general fractional linear function of the form

$$(5.2) \quad f: n \mapsto \frac{an + b}{cn + d}$$

as long as we still assume that $c \leq 0$ and $cn + d > 0$ for $n \in \{0, 1, \dots, m\}$. The respective recursion polynomials to use in the recursive approximation scheme become

$$(5.3) \quad p_1(y) = y + \frac{(cy - a)^2}{ad - bc} \quad \text{for first order Euler,}$$

$$(5.4) \quad p_2(y) = y + \frac{(cy - a)^2}{ad - bc} + \frac{c(cy - a)^3}{(ad - bc)^2} \quad \text{for second order Euler and}$$

$$(5.5) \quad p(x, y) = y + \frac{(cy - a)((x + 1)(cy - a) + dy - b)}{ad - bc} \quad \text{for Newton.}$$

It is obvious that the setup costs to compute the parameters for the approximation algorithm, such as the coefficients in the recursion polynomial, will be higher compared to the corresponding formulas 3.4, 3.5 and 4.2. Moreover, the parameters a and b from the numerator occur in the recursion polynomials and so these polynomials can not be shared among the coordinates of the inverse texture map.

On the other hand the error bounds for the various recursive approximation schemes as reported by theorems 3.1, 3.2 and 4.1 still hold in the exact same form for general linear fractional functions. In particular the parameters a and b are not relevant in these bounds. As an example consider the functions

$$(5.6) \quad f: n \mapsto \frac{50t}{-t + 120}, \quad F: n \mapsto \frac{120t}{t + 50}.$$

Then $f(n)$ ranges from 0 to 250 for $n \in \{0, \dots, 100\}$. The recursion polynomials for first order Euler, second Euler and Newton-Raphson prediction of f respectively are

$$(5.7) \quad p_1(y) = \frac{5}{12} + \frac{61}{60}y + \frac{1}{6000}y^2,$$

$$(5.8) \quad p_2(y) = \frac{121}{288} + \frac{1627}{1600}y + \frac{41}{240000}y^2 + \frac{1}{36000000}y^3,$$

$$(5.9) \quad p(x, y) = \frac{5x + 5}{12} + \frac{x + 1}{60}y + \frac{x - 119}{6000}y^2.$$

The expansion in 5.9 is chosen in such a way that we can make use of the fact that the coefficients for powers of y are arithmetic progressions in x . The progressive errors in screen space $E = 100 - F(f_{100})$ for these three recursion polynomials are exactly the same as the errors we found for the example 3.11, namely $E \approx 1.692$, $E \approx 0.0393$ and $E \approx 0.0496$.

6. IMPLEMENTATION CONSIDERATIONS

Among the recursion polynomials considered the Newton-Raphson based ones seem the most likely candidates for implementation. Their approximation schemes are very accurate while being relatively easy to implement. The actual implementation will depend on the hardware under consideration. Different processors can have very different specifications and timings, certainly where multiplication operations are concerned.

The parameter x runs over an interval of integer values. Therefore its contribution to the recursion polynomial can be computed by an increment in subsequent steps. This replaces a multiply by a simple addition. If available on the processor a combined “multiply-add” instruction should be used to evaluate the recursion polynomial in the parameter y . It is generally more efficient and more accurate than a separate multiply and addition. Most mainstream hardware supports some form of “vector processing” for parallel (arithmetic) operations. This is an excellent way to perform the computations for both texture coordinates at once.

For example the Newton-Raphson scheme for general fractional linear functions with a recursion polynomial defined by 5.5 can be executed completely in parallel on such vector processors to compute both texture coordinates simultaneously. This circumvents the need for “common computations” between the two coordinates to support an efficient implementation.

Below are algorithms that implement both Newton-Raphson schemes, presented in pseudo-Pascal. The first uses the common “reciprocal denominator” computation. The second uses equation 5.5 on a general fractional linear function. The first version seems to be a bit simpler than the second. However it uses two multiplies and a multiply-add in each step that can not be computed in parallel, where the second algorithm needs only two multiply-adds. Note that the more elaborate preparation step of algorithm two can be implemented highly parallel on vector architectures.

The following algorithms compute the reverse texture map coordinates (y_1, y_2) defined by

$$(6.1) \quad y_1 = \frac{a_1 x + b_1}{c x + d}, \quad y_2 = \frac{a_2 x + b_2}{c x + d}$$

for $x \in \{0, \dots, m\}$.

Listing 1: Reciprocal denominator algorithm.

```
{ prepare numerators }
n1 := b1;
n2 := b2;
```

```

{ prepare recursion coefficient
  and initial reciprocal }
q := -d;
y := 1/d;

{ step over the scan line }
for x := 0 to m do
begin
  { compute texture coordinates }
  y1 := n1*y;
  y2 := n2*y;

  { use texture pair (y1,y2) here... }

  if x = m then break;

  { adjust numerators }
  n1 := n1 + a1;
  n2 := n2 + a2;

  { adjust coefficient }
  q := q - c;

  { compute next reciprocal value }
  y := (q*y + 2)*y;
end;

```

Listing 2: General fractional linear algorithm.

```

{ prepare initial texture pair }
y1 := b1/d;
y2 := b2/d;

{ prepare recursion coefficients
  and increments }
rdet1 := 1/(a1*d - b1*c);
rdet2 := 1/(a2*d - b2*c);

q10 := a1*b1*rdet1; q20 := a2*b2*rdet2;
q11 := -2*b1* c*rdet1; q21 := -2*b2* c*rdet2;
q12 := c* d*rdet1; q22 := c* d*rdet2;

s10 := a1*a1*rdet1; s20 := a2*a2*rdet2;
s11 := -2*a1* c*rdet1; s21 := -2*a2* c*rdet2;

```

```
s12 :=      c * c*rdet1; s22 :=      c * c*rdet2;
```

```
{ step over scan line }
```

```
for x := 0 to m do
```

```
begin
```

```
  { use texture pair (y1,y2) here... }
```

```
  if x = m then break;
```

```
  { adjust coefficients }
```

```
  q10 := q10 + s10; q20 := q20 + s20;
```

```
  q11 := q11 + s11; q21 := q21 + s21;
```

```
  q12 := q12 + s12; q22 := q22 + s22;
```

```
  { adjust texture pair }
```

```
  y1 := (q12*y1 + q11)*y1 + q10;
```

```
  y2 := (q22*y2 + q21)*y2 + q20;
```

```
end;
```

```
E-mail address: debosberg@yahoo.com
```