MIP-Map Level Selection for Texture Mapping

Jon P. Ewins, *Member*, *IEEE*, Marcus D. Waller,
Martin White, and Paul F. Lister, *Member*, *IEEE*

Abstract—Texture mapping is a fundamental feature of computer graphics image generation. In current PC-based acceleration hardware, MIP-mapping with bilinear and trilinear filtering is a commonly used filtering technique for reducing spatial aliasing artifacts. The effectiveness of this technique in reducing image aliasing at the expense of blurring is dependent upon the MIP-map level selection and the associated calculation of screen-space to texture-space pixel scaling. This paper describes an investigation of practical methods for per-pixel and per-primitive level of detail calculation. This investigation was carried out as part of the design work for a screen-space rasterization ASIC. The implementations of several algorithms of comparable visual quality are discussed and a comparison is provided in terms of per-primitive and per-pixel computational costs.

Index Terms—Texture mapping, filtering, MIP-map, minification, level of detail, rasterization, interpolation.

1 Introduction

Texture mapping has become one of the most important operations in computer graphics today. The use of texture mapping allows the application of a high degree of visual complexity to 3D scenes without the expense of overly complex geometric modeling. Once the texture mapping process is in place, it allows the application of arbitrary surface detail at a constant performance cost. Each point on a textured surface maps to a position in the texture map from which a value is retrieved that can be used to modify color, reflectivity, transparency, or the surface normal. Heckbert [12] defined texture filtering as the process of resampling the texture image onto the screen grid. Each screen coordinate x, y maps to a texture-space coordinate s, t. For each discrete screen element, or pixel, the filtering operation determines which discrete texture-space pixels, or texels, will contribute. In this paper, we are concerned with the operation of texture filtering within the context of rasterization hardware, as part of a projective screen based renderer. In this case, the filtering operation is between the faces of perspectively projected triangles and 2D image maps.

Fig. 1 illustrates the inverse mapping of a screen-space pixel into texture-space. It can be seen in Fig. 1 that both the shape and scale of the pixel change when mapped to texture-space. The overall operation of texture mapping, including the procedure of *shrink-wrapping* [1] a texture image onto an irregular object, may produce a curvilinear quadrilateral as shown in red [25]. However, for practicality, this is approximated to a flat-sided quadrilateral as shown, with textures mapped linearly to the polygons forming the surface [20]. Based on the representation of the screen pixel as a square, Heckbert [13] proposed that most surfaces can be assumed to be approximately planar over the area covered by a pixel and that, based on this assumption, a screen-space pixel will map to a quadrilateral in texture-space.

• *The authors are with the Centre for VLSI and Computer Graphics, University of Sussex, Falmer, Brighton, BN1 9QT, England, UK.*
E-mail: {J.P.Ewins, M.D.Waller, M.White, P.F.Lister}@sussex.ac.uk.

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 107429.

The representation of the screen-space pixel as a square is only an interpretation. The pixel is, in fact, a point sample and, in the case of the square representation, the four vertices are themselves just sample values. The use of a discrete finite area for a pixel is not a particularly accurate model of the display technology [21], but is done for convenience. While texture filtering algorithms have also been based on other shape representations, including the circle [11], [12], the square representation is convenient and it is the representation used in this paper.

It can be seen in Fig. 1 that several texels may map to one screen-space pixel. The color of each texel underneath the shaded pixel outline should contribute to the final pixel color. The ratio of this texture-space to screen-space scaling is called texture minification. Such texture minification occurs as a result of mapping to irregular objects and to the effects of perspective projection [2], [13]. The texture minification ratio may vary across the projected surface on a per-pixel basis. Therefore, the task of a texture filter is to determine which texels contribute to each pixel. Such a filter must be space-variant, i.e., the size and shape of the filter should adapt to the mapping of each pixel. The first step is to efficiently estimate the pixel inverse mapping. This can be performed in three ways:

1) *Direct addressing of the four offset pixel corners*—This gives an accurate impression of the resulting irregular quadrilateral and, as such, gives consideration to subpixel minification changes. However, it does require four separate mapping operations.
2) *Using the partial derivatives of s and t with respect to x and y*—This assumes a set of partial derivatives (s_x, t_x, s_y, t_y) are constant across the area of the pixel and results in a parallelogram approximation. In practice, these derivative values will vary continuously at a subpixel level. However, this approximation simplifies the calculation and is commonly used in filtering algorithms. Heckbert [13] suggested that, other than near vanishing points and singularities, the parallelogram is a fair approximation. The result of this mapping technique is shown in Fig. 2.

1077-2626/98/$10.00 © 1998 IEEE

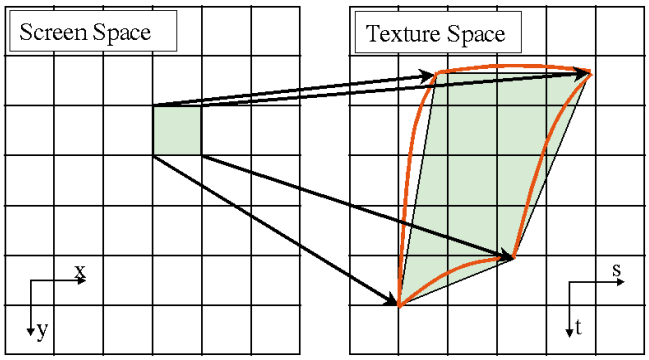


Fig. 1. Pixel inverse mapped to texture-space.

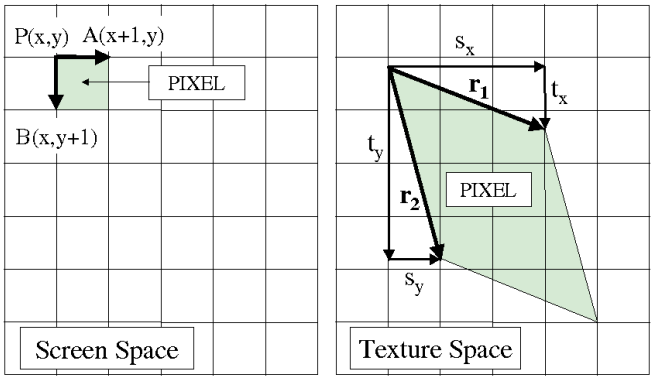


Fig. 2. Pixel inverse mapped to texture-space using constant partial derivatives.

3) *Pixel Clipping*—An alternative, but rather costly, method involves clipping the textured polygon to the screen pixel boundaries [24]. The texture coordinates for each vertex of the resulting clipped region are then calculated and used to determine an average texture sample address for the pixel. The area of the clipped region can be used to determine the texture minification ratio.

1.1 Texture Filtering

Several high-quality space-variant filtering algorithms have been developed and implemented [3], [8], [9], [10], [11], but these tend to be very computationally expensive, producing arbitrary mappings dependent upon the per-pixel scaling. Texture filtering requires accessing texture memory, a time consuming operation, and this should be kept to a minimum and, if possible, constant cost. To this end, filtering algorithms were developed that utilize prefiltered texture storage. The image pyramid or MIP-map is a commonly used means of reducing the cost of per-pixel texture accesses during filter construction. The term MIP-map was introduced by Williams [26]. The term MIP, which stands for *multum in parvo* or "much in a small place," refers to the prefiltering and storage of multiple texture maps of decreasing resolution which attempt to contain the same information in an increasingly smaller space. This is done by progressively averaging groups of four neighboring texels to form each new layer of the image pyramid. This process is continued from the initial, full detail, or base texture level, referred to as level 0, until the final single texel level is reached. The texture minification represented by each layer is based on the side dimensions, not the area. Initial texture dimensions must be a power-of-two magnitude. Fig. 3 illustrates the image pyramid and the texture storage requirement.

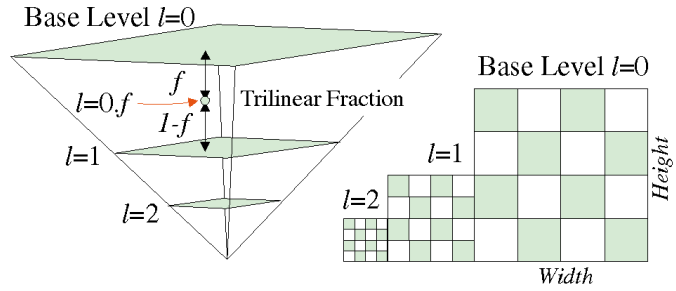


Fig. 3. MIP-map image pyramid and storage requirement.

Each MIP-map level in the pyramid is a power-of-two scaled version of the base level. This is very convenient for hardware implementation, both in terms of MIP-map generation and level calculation. Although shown as such in Fig. 3, the width and height of the base texture are not necessarily of equal magnitude.

For texture filtering, MIP-maps are used in conjunction with bilinear and trilinear interpolation [26]. For bilinear filtering, the texture-space to screen-space minification is calculated and used to select the MIP-map level that contains the prefiltered texel data for a minification ratio nearest to that calculated. This process is also referred to as the level of detail calculation. Generally, systems employing this technique will set a maximum MIP-map level. It is seldom necessary to use all of the possible levels down to the 1×1 texel image. The use of such level clamping avoids problems with the large minification ratios that may arise at vanishing points and singularities. Following the level selection, the texture coordinates s, t are scaled to the dimensions of the selected level and the memory addresses are appropriately offset. Bilinear interpolation is then performed using the four texels nearest to the sample point. As illustrated in Fig. 3, the calculated minification value will frequently fall somewhere between two prefiltered MIP-map levels. For improved results, trilinear filtering is used. Here, bilinear interpolation is performed on both levels on either side of the calculated minification value. Trilinear interpolation is then achieved by linearly interpolating between the color values resulting from these two bilinear interpolations to the intermediate interpolation fraction f as shown. Trilinear filtering is one of the most commonly used methods of filtering in real-time hardware today. The filtering operation can be summarized as follows:

1) Receive the texture address s, t for the current screen pixel x, y
2) Calculate the texture minification, j
3) Extract the level of detail or MIP-map levels to be used, e.g., for a minification ratio of 6 to 1, the levels referring to minification ratios of 4 and 8 are used, i.e., MIP-map levels 2 and 3 ($l = 2$)
4) Calculate the trilinear interpolation fraction, f

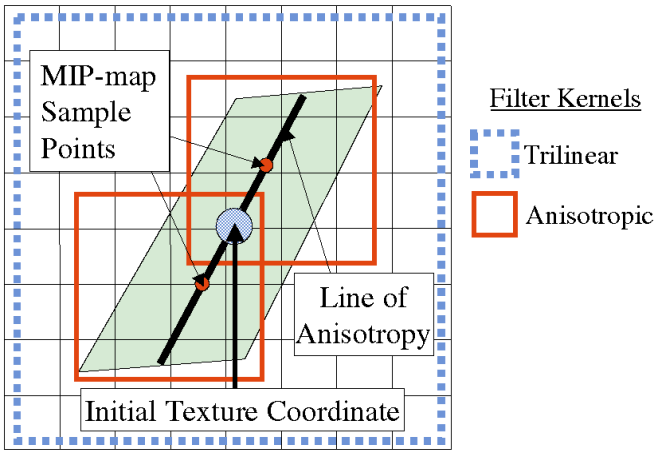


Fig. 4. 2:1 Anisotropic filtering using two trilinear filter operations.

5) Scale the texture address *s*, *t* for the levels selected
6) Perform trilinear interpolation

MIP-map prefiltering and the trilinear interpolation are based on an isotropic square filter shape. The filter is space-variant in size, but not in shape. It can be seen in Fig. 1 and Fig. 2 that the mapping of the pixel is, in fact, anisotropic. That is to say that the screen- to texture-space scaling in the *s* and *t* axes are not equal. Therefore, such isotropic filters will reduce aliasing caused in one axis at the expense of some blurring in the other. Crow [5] suggested the use of summed area tables that allow mapping to arbitrary rectangles. Improved results are returned, but the storage requirements are higher than Williams' MIP-map pyramid and, for textures skewed to 45°, the filter shape is still isotropic. As well as the 3D square-based MIP-map image pyramids, 4D image pyramids have been used that are formed from axis-aligned rectangles [14]. Five-dimensional pyramids that include rotation information to support arbitrary alignment have also been suggested [14] for elliptical filters, but, again, the storage requirements are high.

An effective and practical method proposed by Schilling et al. [18] combines space variance with MIP-mapping by performing multiple trilinear filter operations along the direction of anisotropy. The operation of this filter is illustrated in Fig. 4.

Figs. 13a and 13b illustrate, respectively, the results of a single isotropic trilinear filter and an implementation of this latter kind of multiple-isotropic anisotropic texture filtering [7]. The detailed investigation of such anisotropic filters and a discussion on their possible implementations is beyond the scope of this paper, which is primarily concerned with the investigation of a practical implementation of a single trilinear filter. However, it should be noted that the MIP-map level selection algorithms for such an anisotropic filter are similar in operation and cost to those discussed here.

In the following sections of this paper, Section 2 provides a description of various algorithms for the calculation of texture minification. All methods included were implemented in a C++ simulation environment [6] for comparison of results. This allowed the algorithms to be tested in the context of an entire graphics pipeline. A table of relative computational costs for the implementation of these algorithms is provided. Section 3 then discusses hardware methods for the extraction of the MIP-map level and trilinear interpolation fraction from the calculated texture minification value and, finally, Section 4 summarizes the conclusions drawn from the investigation.

2 Calculating Texture Minification

This section provides a detailed discussion of methods for determining the ratio of texture minification, including both per-pixel and per-primitive algorithms. This study begins with a look at the well known hypotenuse comparison algorithm recommended by Heckbert [13]. This is then followed by a far simpler comparison test, with the results and costs compared. Several different methods of calculating the partial derivatives s_x, t_x, s_y, and t_y are also investigated. First, algebraic solutions are considered, with both direct calculation and incremental interpolation discussed. This is then followed by a look at difference calculations that make use of pixel coherence.

It is important to remember when considering these different solutions that they are all approximations, based on a pixel representation that is itself an approximation, for an isotropic filter that is also an approximation. There is no right or wrong choice. This is recognized by the OpenGL specification [19] that suggests a range of possible methods rather than recommending a particular one. In the final analysis, it is the perceived quality of the images produced, particularly in animations, relative to the implementation and performance cost that matters. Fig. 15 compares the result of applying each of the main algorithms discussed in the following sections to the texturing of a perspectively projected plane with the pathological checkerboard pattern.

2.1 Hypotenuse Comparison

As mentioned earlier, in practice, texture minification does not occur equally in both the *s* and *t* directions. Therefore, for isotropic MIP-map filtering, one minification figure must be determined that provides the best approximation. If a minification value is too large, the image will be excessively blurred, too small, and the aliasing artifacts are not removed. Heckbert worked on the premise that overfiltering is more desirable than underfiltering. Although several different methods were suggested in his paper [13], he identified choosing the greater of the two edges of the projection parallelogram as the best solution. This can be found from the hypotenuse direction vectors shown in Fig. 2 with:

$$j = \max\left(|\mathbf{r_1}|, |\mathbf{r_2}|\right)$$

where

$$|\mathbf{r_1}| = \sqrt{s_x^{\,2} + t_x^{\,2}}$$

and

$$|\mathbf{r_2}| = \sqrt{s_y^{\,2} + t_y^{\,2}}$$

$$\therefore j = \sqrt{\max\left(s_x^{\,2} + t_x^{\,2}, s_y^{\,2} + t_y^{\,2}\right)}, \qquad (1)$$

where j = texel : pixel minification ratio.

The computational cost of this expression is, therefore, four multiplications, two additions, one square root, and a single comparison. Added to this are the costs of calculating the partial derivatives s_x, t_x, s_y, and t_y. These are discussed in the following subsections.

2.1.1 Algebraic Solution to s_x t_x s_y t_y

Due to the nonlinear nature of perspectively correct texture mapping, the changes in s and t with respect to x and y alter on a per-pixel basis. To employ perspective correction, texture coordinates s and t are generated by hyperbolic interpolation [2], [13]. The nonlinear texture mapping coordinates are generated as the function of two linear operations, as shown in (2):

$$s = \frac{S}{Q}, \quad t = \frac{T}{Q} \tag{2}$$

$$S = Ax + By + C,\ T = Gx + Hy + I,\ Q = Dx + Ey + F.$$

Terms A to I are the coefficients of the three separate linear expressions for S, T, and Q. These are plane equations calculated from the values at each of three triangle vertices that describe the plane. S and T are the vertex texture coordinates homogenized to the view plane and Q is a function of the reciprocal of depth Z.

Following partial differentiation of (2), the partial derivatives are:

$$s_x = \frac{\partial s}{\partial x} = \frac{(AF - CD) + (AE - BD)y}{Q^2} = \frac{K_3 + K_1 y}{Q^2}$$

$$t_x = \frac{\partial t}{\partial x} = \frac{(GF - ID) + (GE - HD)y}{Q^2} = \frac{K_4 + K_2 y}{Q^2}$$

$$s_y = \frac{\partial s}{\partial y} = \frac{(BF - CE) + (BD - AE)x}{Q^2} = \frac{K_5 - K_1 x}{Q^2}$$

$$t_y = \frac{\partial t}{\partial y} = \frac{(HF - IE) + (HD - GE)x}{Q^2} = \frac{K_6 - K_2 x}{Q^2} \tag{3}$$

The Q^2 term is common for each of these expressions and can be factored out of the comparison test. Therefore, the final per-pixel hypotenuse comparison calculation can be written as:

$$j = \frac{1}{Q^2}\sqrt{\max\begin{pmatrix}(K_3 + K_1 y)^2 + (K_4 + K_2 y)^2, \\ (K_5 - K_1 x)^2 + (K_6 - K_2 x)^2\end{pmatrix}} \tag{4}$$

As shown in (2), the reciprocal of Q is already required for the perspective correction of the texture coordinates. Therefore, the per-pixel computational cost above that which is already required for point-sampled texture mapping is six additions/subtractions, 10 multiplications, one square root, and one maximum comparison.

It can be seen that, because Q can be factored out, the direction vector $\mathbf{r_1}$ and the associated component partial derivatives s_x and t_x are dependent only on y and the direction vector $\mathbf{r_2}$ and the partial derivatives s_y and t_y are dependent only on x. This fact can be used to simplify the implementation. Assuming the primitive traversal mechanism of the scan conversion implementation will move only in either the x direction (span traversal) or y direction (scan-line traversal) at any one time, then the s_x and t_x calculations and the s_y and t_y calculations are independent. Therefore, in hardware, the same arithmetic units can be used twice, assuming some extra logic required for the storage of the unchanging values. This reduces per-pixel cost in terms of required functional units by four multiplications, three additions, and one square root.

Another saving was proposed by Heckbert [13] who suggested per-primitive precalculation and look up table (LUT) storage of $\mathbf{r_2}$ for the x extent of the primitive and, then, one calculation of $\mathbf{r_1}$ per scan-line, to minimize the per-pixel cost. The effectiveness of this method depends on the size of the triangles used.

Assuming the coefficients of (2) already exist for the texture address generation and considering reused terms, per-primitive setup costs for the calculation of K_1, K_2, K_3, K_4, K_5, and K_6 are 12 multiplications and six subtractions.

2.1.2 Incremental Interpolation

It was shown in (4) that the Q^2 reciprocal can be factored out as below.

$$j = \frac{1}{Q^2}\sqrt{\max\left(s_x'^2 + t_x'^2, s_y'^2 + t_y'^2\right)}, \tag{5}$$

where

$$s_x' = s_x Q^2, \quad t_x' = t_x Q^2, \ s_y' = s_y Q^2$$

and

$$t_t' = t_y Q^2.$$

Each partial derivative term is now a linear expression of the form $K_b \pm K_a x$ or $K_b \pm K_a y$ and, as such, can be calculated in a graphics pipeline by incremental interpolation. Such a method exhibits a dependence on the scan conversion traversal employed, but requires only one addition per derivative. If the traversal in x and y is assumed independent, then the cost is two additions per-pixel. Initial starting-value calculations must be performed during a setup phase.

2.1.3 Texture Minification Scaling

Texture coordinates are usually applied to 3D models in the range 0-1. This ensures the independence of the model from the physical dimensions of the texture maps applied. It is therefore necessary to scale the minification value j by the dimensions of the texture base level.

If we assume a rectangular texture map of dimensions *Width* by *Height*, then, for systems supporting textures of nonsquare dimensions where *Width* does not equal *Height*, the partial derivatives must be scaled by the corresponding dimensions before the comparison test of (5) with:

$$s_x'' = s_x' \ Width \text{ and } s_y'' = s_y' \ Width$$
$$t_x'' = t_x' \ Height \text{ and } t_y'' = t_y' \ Height. \tag{6}$$

Assuming power-of-two dimensions, this requires either four left-shift operations for fixed-point representation or four small fixed-point additions for floating-point number representation. Alternatively, the coefficients K_1, K_2, K_3, K_4, K_5, and K_6 can be scaled during setup.

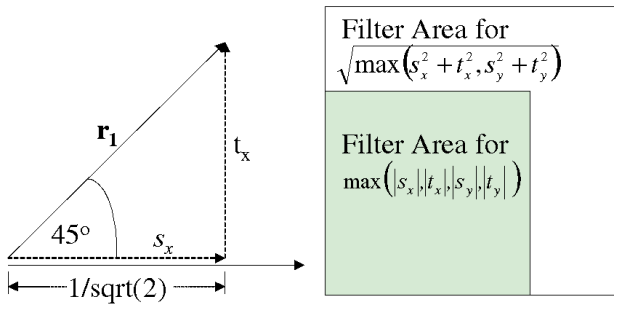


Fig. 5. Worst case for vector approximation.

Where the texture dimensions are equal, moving the texture scaling to after the comparison test can further reduce the cost to a single scaling operation. The use of post comparison scaling can also further simplify the hardware implementation, as explained in Section 3.

2.2 Partial Derivative Comparison

For this approach, we have simplified the hypotenuse comparison test of Section 2.1 by removing the expensive square root part of this test. The texture minification ratio is found as the result of:

$$j = \max\left(|s_x|, |t_x|, |s_y|, |t_y| \right)$$

$$j = \frac{1}{Q^2} \max\begin{pmatrix} |K_3 + K_1 y|, |K_4 + K_2 y|, \\ |K_5 - K_1 x|, |K_6 - K_2 x| \end{pmatrix} \quad (7)$$

This reduces per-pixel computational cost to six multiplications, four additions/subtractions, and a three-stage maximum comparison test. However, as described for the hypotenuse comparison in Section 2.1, this requirement can be further reduced if x and y independence is considered. This reduces per-pixel computational cost by two multiplications and two additions/subtractions and removes a comparison test. Using incremental interpolation, the cost is just two multiplications, two additions/subtractions, and two comparison tests.

This approach also reduces the cost of the texture scaling as described in Section 2.1.3 by half with:

$$j = \max\left(\max\left(|s_x|, |s_y| \right) Width, \max\left(|t_x|, |t_y| \right) Height \right). \quad (8)$$

In this simplified approach, the chosen partial derivative will always be a component of the required direction vector, $\mathbf{r_1}$ or $\mathbf{r_2}$. In the worst case, the two orthogonal components of a vector will be of equal dimension and $1/\sqrt{2}$ of the hypotenuse vector magnitude. This is illustrated in Fig. 5.

This means that the measured minification is less than that calculated by the hypotenuse comparison and that the amount of filtering is reduced. This is the lower bound recommended by the OpenGL specification [19]. Figs. 12a and 12b show the point-sampled MIP-map levels for the partial derivative maximum comparison and hypotenuse comparison tests, respectively. As expected, the latter selects the lower-detail levels slightly earlier, but, otherwise, the results are similar. Figs. 12c, 12d, 12e, and 12f illustrate the effect on a trilinear-filtered texture.

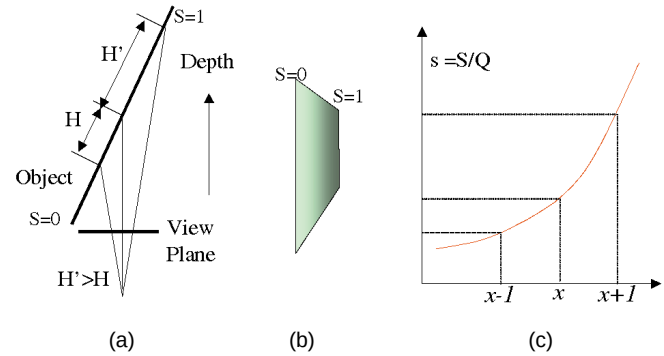


Fig. 6. Nonlinear response due to perspective foreshortening. (a) plan view. (b) Front projected view. (c) Response of s versus x.

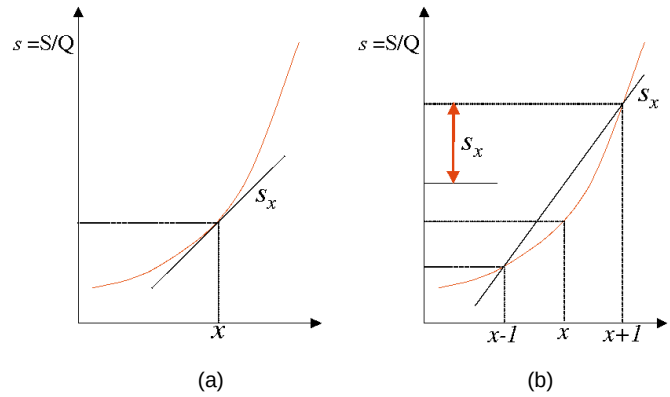


Fig. 7. The difference calculation.

2.3 Difference Comparison

In the algorithms discussed previously, the partial derivatives used in the different comparison tests were calculated algebraically by either direct per-pixel computation or incremental interpolation. In this section and the next, we look at alternative ways of calculating the partial derivatives, using difference calculations that take advantage of pixel data coherence to simplify the operation. In all cases, either of the above alternative comparison tests may still be used.

Fig. 6 defines the nonlinear response caused by perspective foreshortening. The curve shown is exaggerated for clarity.

Fig. 7 shows the difference between the results of this approach (Fig. 7b) and those of the direct computation methods (Fig. 7a) with regards to the approximation of the texture minification at a point in the image. The partial derivative is found as the average over two neighboring pixels to compensate for the nonlinear response of s against x.

Using (2), where A, B, G, H, D, and E are the coefficient terms for S, T, and Q before perspective correction, we can approximate the partial derivatives as:

$$s_x = \frac{\partial s}{\partial x} \approx 0.5\left(\frac{S+A}{Q+D} - \frac{S-A}{Q-D} \right) \; s_y = \frac{\partial s}{\partial y} \approx 0.5\left(\frac{S+B}{Q+E} - \frac{S-B}{Q-E} \right)$$

$$t_x = \frac{\partial t}{\partial x} \approx 0.5\left(\frac{T+G}{Q+D} - \frac{T-G}{Q-D} \right) \; t_y = \frac{\partial t}{\partial y} \approx 0.5\left(\frac{T+H}{Q+E} - \frac{T-H}{Q-E} \right) \quad (9)$$

Therefore, the per-pixel computational cost of this method considering reused denominator components is

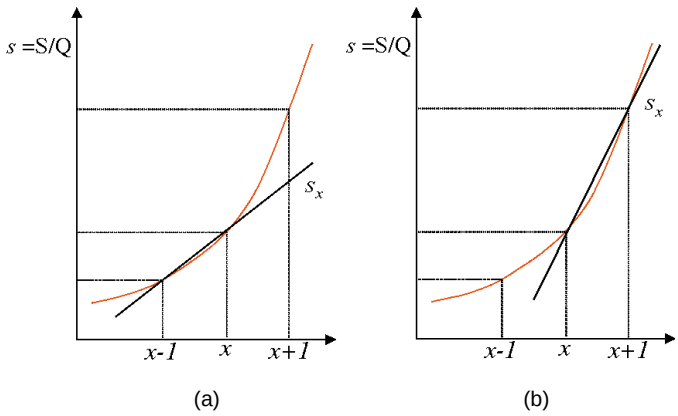


Fig. 8. Simplified difference calculation.

16 additions/subtractions and eight divides. The eight divides can be replaced with four reciprocals and eight multiplications. The inexpensive divide by two can be performed after the comparison operation or combined into the texture dimension scaling operations.

Significantly, with this method, there is no requirement for setup calculations beyond those required for point sampled texturing. Using the hypotenuse comparison, the per-pixel computational cost is 18 additions/subtractions, 12 multiplications, four reciprocals, one square root, and one scaling operation. Using the simplified partial derivative comparison test of Section 2.2, the costs are reduced by four multiplications and two additions/subtractions, and the requirement for a square root is removed.

An important point to note here is that because the Q terms cannot be factored out of the expressions, with this technique, the x and y traversals are not independent.

2.4 Simplified Difference Approximation

The difference approximation described in Section 2.3 can be simplified by extending the partial derivatives in only one direction, as shown in Fig. 8.

We now calculate the partial derivatives with:

$$s_x = \frac{\partial s}{\partial x} \approx \left| s - \frac{S \pm A}{Q \pm D} \right| \text{ and } t_x = \frac{\partial t}{\partial x} \approx \left| t - \frac{T \pm G}{Q \pm D} \right|$$

$$s_y = \frac{\partial s}{\partial y} \approx \left| s - \frac{S \pm B}{Q \pm E} \right| \text{ and } t_y = \frac{\partial t}{\partial y} \approx \left| t - \frac{T \pm H}{Q \pm E} \right|, \quad (10)$$

where $s = \frac{S}{Q}$ and $t = \frac{T}{Q}$ have already been calculated for texture addressing. The partial derivatives are thus calculated at a cost of 10 additions/subtractions, four multiplications, and two reciprocals.

The ± operators in (10) highlight the fact that the sign chosen will affect the texture minification returned. This is the reason why the difference algorithm of the previous section uses the average over two pixels, but how necessary is this? The answer depends on the severity of the curve. Fig. 13a shows an example of an oblique tilted plane with a steep depth range at close proximity to the view plane. Also shown is a graph of the response of s and t against x for a particular scan-line.

In this example, the increment in s is seen to increase noticeably, although not excessively. By experimentation, we found that as long as the direction of gradient choice is consistent, the images generated are acceptable.

However, to maintain the hypothesis that it is better to overfilter rather than underfilter, it would seem prudent to ensure that the larger of the two possible gradients offered in Figs. 8a and 8b is used. Fig. 9 and Fig. 10 show how the larger of the two gradients can be determined from the signs of the coefficients of Q from (2).

$$\text{Where } D = Q_x = \frac{\partial Q}{\partial x} \text{ and } E = Q_y = \frac{\partial Q}{\partial y}$$

A positive value for D implies that the depth of the polygon (distance from viewer) decreases with respect to x (negative z_x). Therefore, the magnitudes of s_x and t_x are falling and the gradient to $(x - 1)$ should be used. For a negative D, the $(x + 1)$ gradient is used. Coefficient E can be used in the same way for s_y and t_y. Therefore, the ± operators in (10) can be determined using the sign of D and E at no extra cost.

Figs. 13d and 13e show the results of trilinear filtered color coded MIP-map levels for the simplified difference method and two-pixel averaged difference method, respectively.

2.5 Data Storage Schemes

The simplified difference approximation of the previous section has advantages in hardware implementation. By storing the previous values of s and t found during span traversal, the s_x and t_x values can be found by simple subtraction of neighboring pixels. This reduces per-pixel cost by five addition/subtractions, one reciprocal, and two multiplications.

It should be noted however that, with this method, the gradient value chosen, as illustrated in Figs. 8a and 8b, may not be consistent, depending instead upon the direction of span traversal. There are scan conversion algorithms, such as those based on linear edge functions, that traverse left and right on alternate scan-lines [17], [23]. Such algorithms will result in level artifacts as shown in Fig. 13c. The effects are, of course, more significant the greater the minification variation in the projected geometry.

Pixel data caching schemes can also reduce the cost of calculating s_y and t_y when applied to the pixels on subsequent scan-lines. The effect of such an approach in terms of balancing reduced computational cost against storage requirement will depend on the size and shape of the triangle.

2.5.1 Deferred Texturing Implementations

In an architecture supporting deferred shading [16], the per-pixel scan conversion and occlusion operations are performed independently of the shading operation. For texturing, this means that the texture coordinates are generated for each pixel in advance of any texture memory access or filtering operation. The coordinates for neighboring pixels, (s_0, t_0), (s_1, t_1), (s_2, t_2), and (s_3, t_3), as shown in Fig. 11, are therefore available simultaneously.

These pixel samples can be considered as the texture coordinates for the four offset corners of the screen-space pixel marked in blue. Using just (s_0, t_0), (s_1, t_1), and (s_2, t_2),

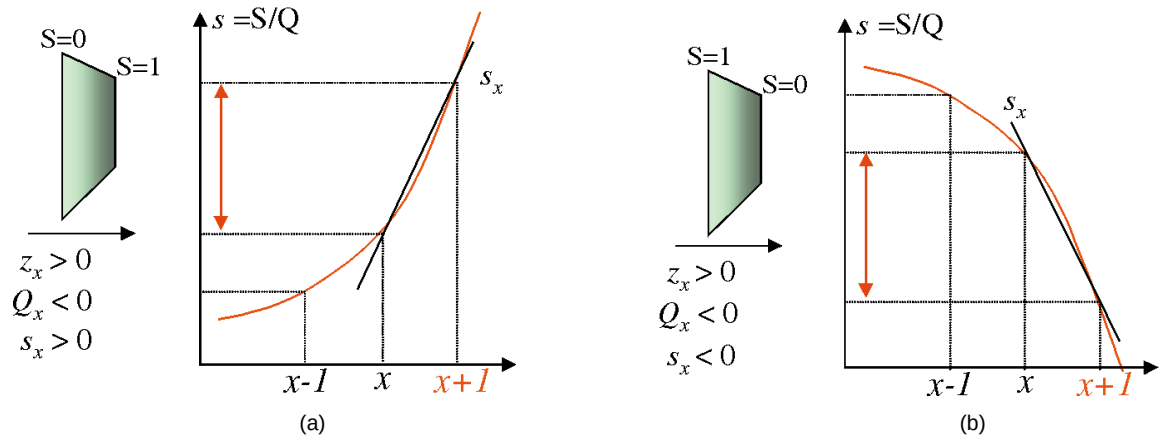


Fig. 9. Selecting maximum gradient for negative Q_x.

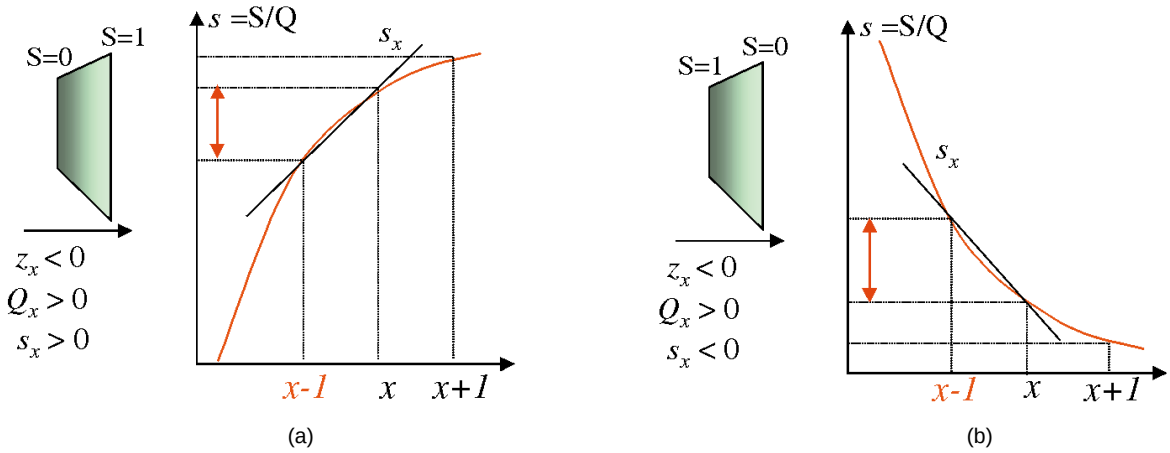


Fig. 10. Selecting maximum gradient for positive Q_x.

the simplified difference derivatives of Section 2.3 can be calculated directly at a cost of just four subtractions.

Another point to note here is that if all four values are used, then the filter kernel can be based on the actual irregular quadrilateral rather than the approximated parallelogram.

2.6 Cross-Product Solution

The difference between the cross-product method described in this section and the methods discussed previously is that, here, the comparison test is dropped altogether. Here, we present a method for calculating the texture minification j directly from a single expression. The screen-space to texture-space pixel mapping is again considered to be a parallelogram, but this time, the minification ratio for the MIP-map square filter is determined as a function of both of the two edges of this parallelogram. This is done by calculating the area of the parallelogram and by then finding the edge length of a square of equal area.

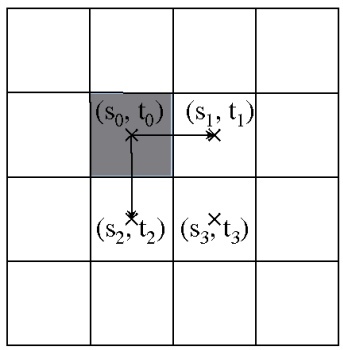


Fig. 11. Deferred shading precalculated coordinates.

The area of a parallelogram can be found with the cross-product of the two defining vectors.

$$j = \sqrt{\left| s_x t_y - s_y t_x \right|}. \tag{11}$$

By substitution of (3), it can be shown that:

$$j = \sqrt{\left| \frac{k}{Q^3} \right|}, \tag{12}$$

where $k = AHF + EGC + BDI - BGF - DHC - AIE$.

This has a per-primitive cost of 12 multiplications and five additions/subtractions. However, in a pipeline using incremental interpolation, this cost can be reduced further. Reorganization of the result of substituting (3) into (11) produces:

$$j = \sqrt{\left| \frac{Q_{start} k_1 + S_{start} k_2 + T_{start} k_3}{Q^3} \right|}, \tag{13}$$

where

$k_1 = (AH - BG)$, $k_2 = (EG - DH)$, and $k_3 = (BD - AE)$,

and S_{start}, T_{start}, and Q_{start} are starting values for incremental interpolation.

The constant k can now be calculated in setup by substitution of the known interpolation starting values S_{start}, T_{start}, and Q_{start} at a cost of nine multiplications and five additions/subtractions. Per-pixel cost is three multiplications and one square root. Again, the reciprocal of Q is assumed already available from the perspective divide.

2.7 Per-Vertex Minification Interpolation

Here, the texture minification value is incrementally interpolated from precalculated vertex values. A single texture minification value is calculated during setup for each vertex of the triangle. The texture minification for each pixel is then found by solution of a linear expression or, for more accuracy, by hyperbolic or rational linear expression, requiring a perspective divide. This can be calculated either directly or by incremental interpolation, in the same way as the other vertex parameter data; color, normals, depth, and texture coordinates. The coefficients for the minification expression are found with:

$$\frac{\partial j}{\partial x} = \frac{(j_1 - j_0)(y_2 - y_0) - (j_2 - j_0)(y_1 - y_0)}{(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)}$$

$$\frac{\partial j}{\partial y} = \frac{(j_1 - j_0)(x_2 - x_0) - (j_2 - j_0)(x_1 - x_0)}{(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)}$$

$$jstart = j_0 + \left(OffsetX \frac{\partial j}{\partial x}\right) + \left(OffsetY \frac{\partial j}{\partial y}\right) \quad (14)$$

Considering the terms common to the other parameters, the additional setup cost is, therefore, six multiplications and six additions/subtractions. For perspective correction, an extra multiplication is required to homogenize [2] to the view plane. Using incremental interpolation, the per-pixel cost is then just a single increment/decrement addition plus a divide for perspective correction.

The problem, however, is how to obtain vertex texture minification data in the first place. As explained in the Introduction, the amount of minification is a function of the perspective projection of the scene geometry. Therefore, j cannot be calculated in advance in the model generation phase; it must be performed in setup for each projected, clipped, visible triangle. Any of the traversal independent methods described above could be used to calculate the minification at each vertex. However, these methods use coefficients and expressions that are functions of the plane of the triangle, not the vertex. Therefore, shared vertices may be recalculated several times.

Using the partial derivative maximum comparison method, the calculation of texture minification for three vertices requires a further 30 multiplications, 18 additions/subtractions, and nine comparisons. This is a very large setup requirement and, so, is only viable for large triangle sizes.

2.8 Per-Primitive Average Minification

If texture minification varies continuously across each primitive and bilinear and trilinear filtering operations are performed per-pixel, then it seems sensible to calculate the level of detail at each pixel. However, any improvements in the resulting visual quality must be weighed against the cost of this operation. Texture minification can be estimated cheaply on a per-primitive basis and several current PC graphics accelerator cards adopt that approach for this reason. Therefore, we include a discussion of such a method.

Using the x and y and the s and t coordinates supplied for each vertex of a triangle, the areas of the triangles in both screen-space and texture-space can be calculated. The texture minification is then found as the square root of the ratio of these two areas.

The area of a triangle can be found as half of the magnitude of the cross-product of two of its edges. Therefore, for screen and texture-space,

$$PixelArea = \frac{1}{2}\left((x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)\right)$$

$$TexelArea = \frac{WidthHeight}{2}\left((s_1 - s_0)(t_2 - t_0) - (s_2 - s_0)(t_1 - t_0)\right). \quad (15)$$

The texture-space area scaling by *WidthHeight*, shown in (15), assumes the texture coordinates are supplied in the range 0-1. The divide by two is common to both area calculations and is, therefore, cancelled out by the ratio divide.

We assume that the cross-product for x and y is already present in the setup code for shading attribute parameter preparation. See the denominators in (14). Including the area-ratio divide and edge-length calculation, the setup cost is, therefore, five subtractions, one divide, one square root, and a power-of-two scaling operation.

The resulting visual quality of this approach is clearly highly dependent upon the size of triangles used in the database model. Fig. 14 illustrates the results for varying amounts of tessellation. The method is shown compared with the cross-product per-pixel algorithm. The images show clearly how the results of these two area-based methods converge as triangle size decreases. Higher tessellation is required where the minification range is greatest. Conveniently, as can be seen in Fig. 14c, following object-space tessellation, perspective projection causes the triangles to be smaller toward the vanishing points of the model. Further postprojection adaptive tessellation might be performed based on proximity to the view plane and obliqueness of the projection angle [15].

2.9 An Algorithm Comparison

Table 1 provides a summary of the relative computational costs for the algorithms described in the preceding sections. These cost figures are given in terms of per-primitive setup costs and per-pixel run-time costs and are the costs above that required for point sampled texture mapping. The costs of texture minification scaling and those of other operations common to all of the described algorithms, such as level extraction, have been omitted from the table for clarity. Section 3 provides information on the hardware implementation of extracting the level of detail data from the minification value.

TABLE 1
SUMMARY OF ALGORITHM COMPUTATIONAL COSTS

THE ALGORITHMS		Per Primitive Setup Cost	Per Pixel Run Time Cost
1. Direct Algebraic Calculation of Partial Derivatives s_x,t_x,s_y,t_y			
1.1	Hypotenuse Comparison Test	12 m, 6 a/s	10 m, 6a/s, 1 sqrt, 1 MC
1.2	As 1.1 with split *x,y* calculation	12 m, 6 a/s	6 m, 3 a/s, 1 sqrt, 1MC
1.3	Partial Derivative Comparison Test	12 m, 6 a/s	6 m, 4 a/s, 3 MC
1.4	As 1.3 with split *x,y* calculation	12 m, 6 a/s	4m, 2 a/s, 2 MC
2. Incremental Interpolation of Partial Derivatives s_x,t_x,s_y,t_y			
2.1	Partial Derivative Comparison Test with incremental interpolation and split *x,y* calculation	16 m, 10 a/s	2m, 2 a/s, 2 MC
3. Calculation of Partial Derivatives s_x,t_x,s_y,t_y from Difference Algorithms			
3.1	Difference Algorithm with Hypotenuse Comparison	0	12 m, 18 a/s, 4 r, 1 MC, 1 sqrt.
3.2	As 3.1 with Partial Derivative Comparison	0	8 m, 16 a/s, 4 r, 3 MC
3.3	Simplified Difference Algorithm with Hypotenuse Comparison	0	8 m, 12 a/s, 2 r, 1 MC, 1 sqrt
3.4	As 3.3 with Partial Derivative Comparison	0	4 m, 10 a/s, 2 r, 3 MC
3.5	As 3.4 with Scan Line Data Storage	0	2m, 5 a/s, 1 r, 3MC
3.6	As 3.4 with Deferred Texturing	0	4 a/s, 3MC
4. Direct Calculation of Texture Minification			
4.1	Cross Product Algorithm – $j(Q^3)$	9 m, 5 a/s	3 m, 1 sqrt
5. Per Vertex Interpolated Texture Minification (with perspective correction)			
5.1	Excluding Vertex Minification Calculation	6(7)m, 6 a/s	1 a/s, (1d)
5.2	Including Vertex Minification Calculation	36(37)m, 24 a/s, 9 MC	1 a/s, (1d)
6. Per Primitive Texture Minification			
6.2	Triangle Area Ratio	5 a/s, 1 d, 1 sqrt	0

m = Multiplication, a/s = Addition/Subtractions, r = Reciprocal, MC = Maximum Compare, d = Divide, sqrt = Square Root

3 MIP-MAP LEVEL SELECTION

The value *j*, when correctly scaled to the physical dimensions of the texture map in use, is an estimation of the degree of texture minification at a particular point in the image. The MIP-map level and trilinear interpolation factor must be extracted from this minification figure. The required level is determined with:

$$l = \lfloor \log_2 j \rfloor. \tag{16}$$

The trilinear interpolation fraction is then found with:

$$f = \frac{j - 2^l}{2^l}. \tag{17}$$

In hardware, because the MIP-map levels refer to power-of-two increments, these operations are straightforward to implement. For fixed-point number representation, the level *l* can be found directly from the position of the most significant active bit in the binary word. The removal of this bit, followed by a right shift by *l* places, returns the value of *f*.

For floating-point representation, the calculation of *l* and *f* is also straightforward. The exponent value is extracted from the floating-point word to provide *l*, and the fractional component is extracted to provide the trilinear fractional value *f*. This is proved below, where *j* is a floating-point number with exponent part 2^N and significand 1.F such that:

$$j = (-1)^S \times 2^N \times 1.F. \tag{18}$$

For texture minification, the sign S can be assumed always positive and, so, by substitution of (18) into (17):

$$l = N$$

and

$$f = \frac{\left(2^N \times 1.F\right) - 2^N}{2^N} = \frac{2^N(1.F - 1)}{2^N} = 0.F. \tag{19}$$

If the hardware implementation need only support bilinear, rather than trilinear, filtering, then a number of other simplifications can be made. First, there is no need to generate the trilinear interpolation fraction. Second, the relationship given in (20) can be used to remove the requirement for the square root from algorithms 1.1, 1.2, 3.1, 3.3, 4.1, and 6.2.

$$\log_2 \sqrt{2^m} = \log_2 2^{\frac{m}{2}} = \frac{m}{2}. \tag{20}$$

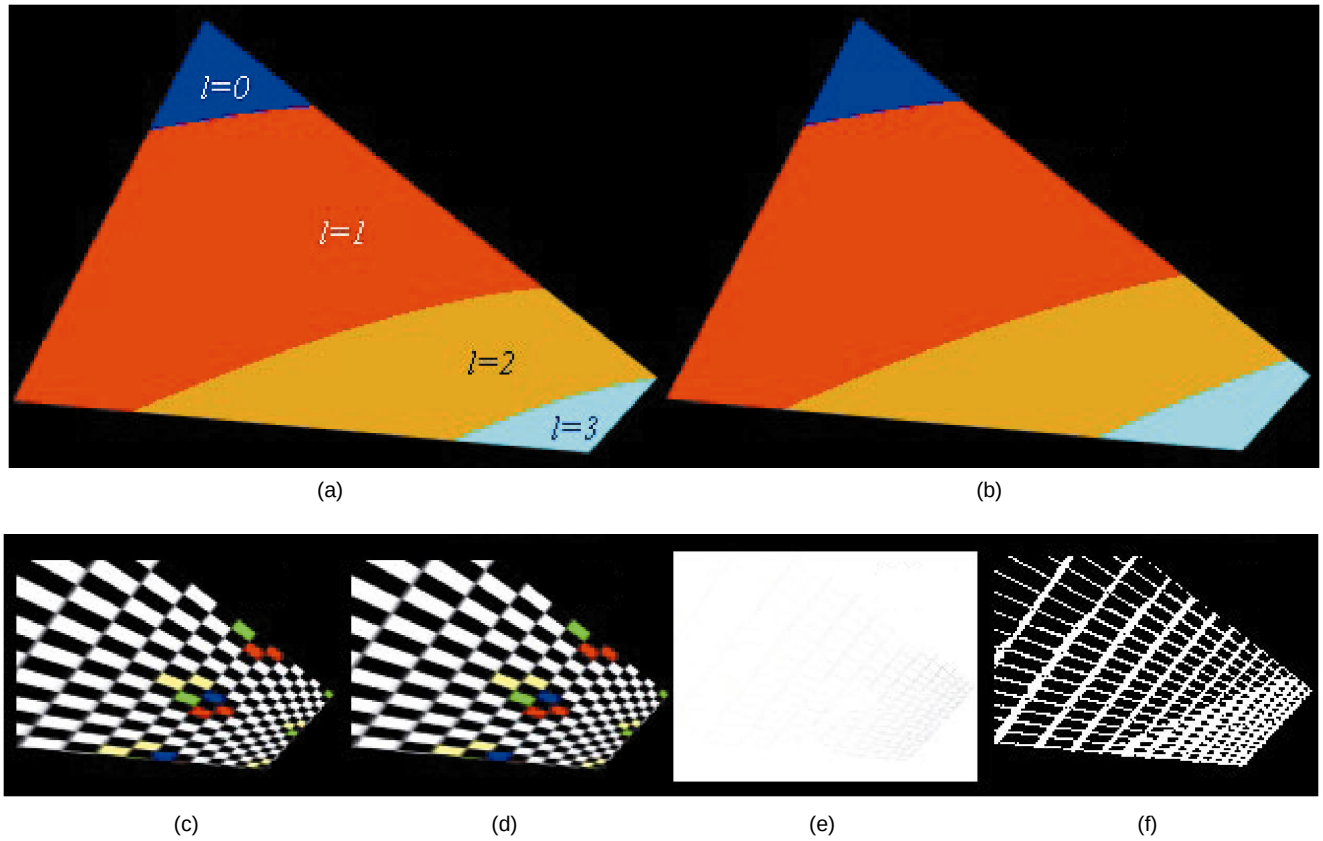


Fig. 12. Comparison of hypotenuse and partial derivative comparison algorithms. Tilted plane displaying point sampled MIP-map level selection by (a) partial derivative comparison and (b) hypotenuse comparison. (c, d) Trilinear filtered checker board pattern using (c) partial derivative comparison and [d] hypotenuse comparison. (e) Actual color difference between (c) and (d) (Note that the differences are very small and dependent upon the gamma correction of the display device). (f) Absolute difference between (c) and (d).

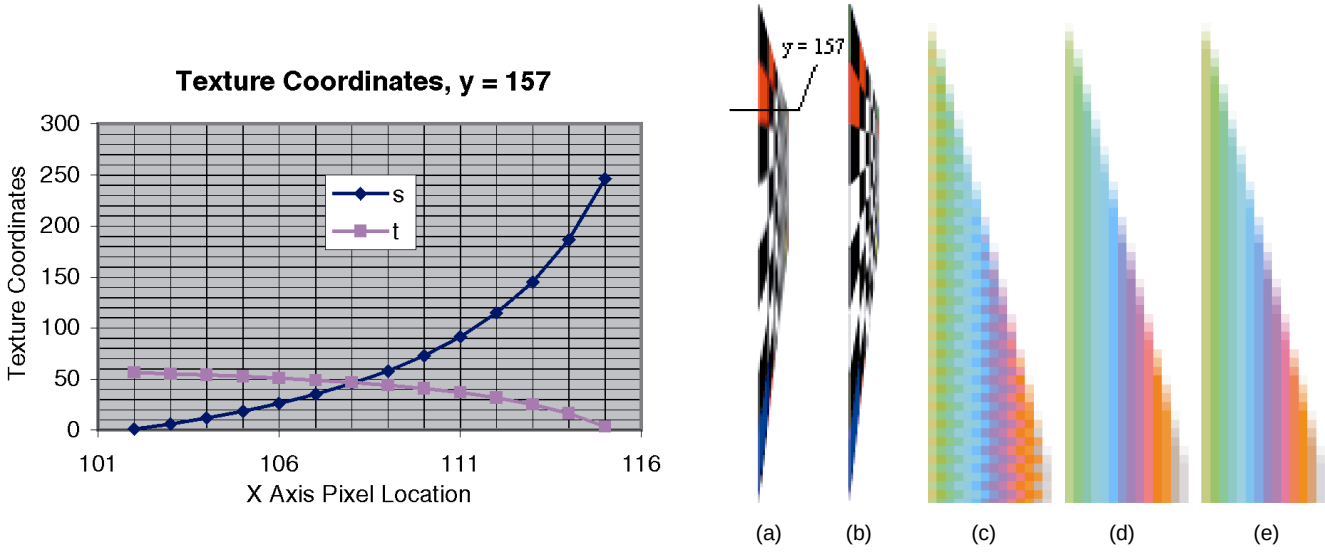
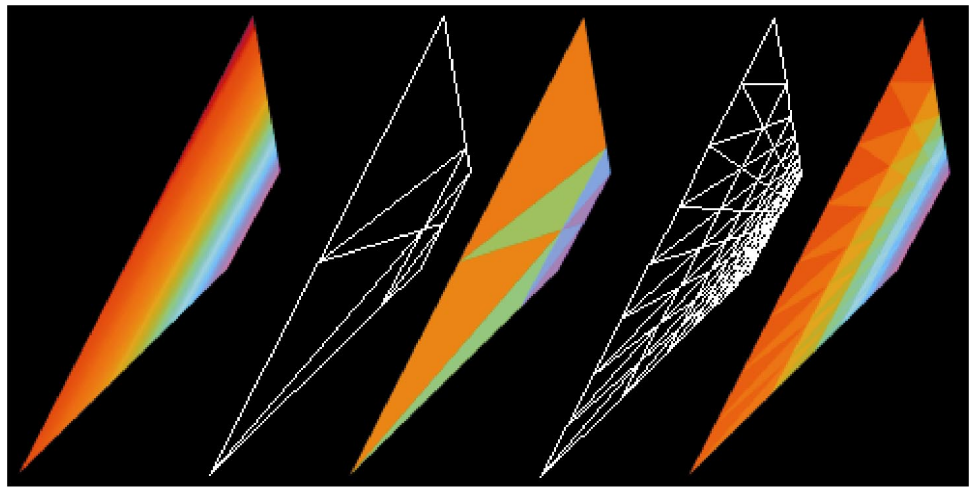


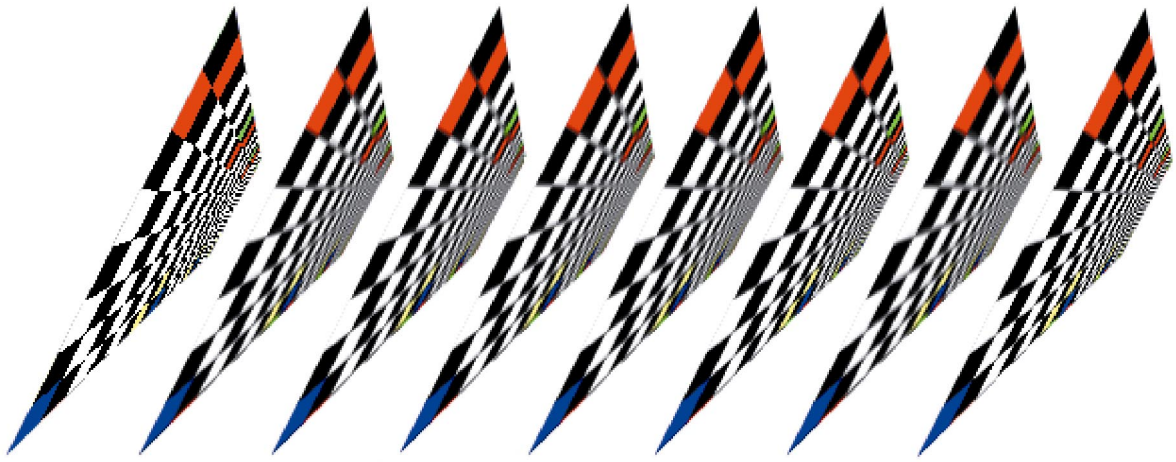
Fig. 13. Difference based minification calculation. Graph showing s and t coordinates for scan-line y = 157 of a tilted plane image shown (a) with trilinear filtering and (b) with anisotropic texture filtering. (c, d, e) Trilinear filtered colored MIP-map levels with: (c) Simplified difference calculation with data storage for primitive traversal with alternating direction. (d) Simplified difference calculation using maximum gradient. (e) Two pixel average difference.

For the algorithms using algebraic calculation for the partial derivatives, as explained in Sections 2.1.1 and 2.1.2, the coefficients calculated in setup can be very small and, therefore, require a high degree of precision in the number representation used. We therefore found it sensible to prescale the algorithm coefficients K_1, K_2, K_3, K_4, K_5, and K_6 to the texture base size during the setup operations, as suggested in Section 2.1.3. As well as reducing the precision requirement, this also removes the requirement for per-pixel scaling. The significance of doing this is dependent upon the triangle size.



(a) (b) (c) (d) (e)

Fig. 14. Per-primitive MIP-map level selection. Tilted planes displaying trilinear filtered, color-coded MIP-map levels selected by (a) cross-product area based per-pixel algorithm (unchanged by tessellation) and per-primitive area based algorithm using (b) low tessellation and (c) high tessellation.



(a) (b) (c) (d) (e) (f) (g) (h)

Fig. 15. Comparison of methods using the tessellated plane model from Fig. 14c. (a) Point sampling. (b) Hypotenuse comparison test and (c) partial derivative comparison test, both with derivatives calculated by direct computation. (d, e) Partial derivatives calculated by (d) two pixel difference subtraction and (e) single pixel simplified difference subtraction, both using partial derivative comparison test. (f, g, h) MIP-map level selection using (f) cross-product algorithm, (g) per vertex interpolation (linear) and (h) per-primitive calculation.

If postcomparison scaling is used, then the cost of this operation can be removed if a custom floating-point unit is being implemented [22]. Extracting the exponent value N from a floating-point number requires the subtraction of the bias. Assuming power-of-two texture dimensions, the exponent addition required for scaling can therefore be incorporated into the bias subtraction. For IEEE754 single precision floating point [4], using a texture of side dimensions 1,024 or 2^{10}:

$$N = e - 127 + 10 = e - 117 . \tag{21}$$

4 Conclusion

In this paper, we have presented a number of different algorithms for texture minification calculation and level selection for MIP-map trilinear filtering. We have discussed the operation of these algorithms and the relative costs of implementation. Methods of cost reduction have been suggested as well as suggestions for their practical implementation.

As stated in the introduction to the algorithm discussion, there is no right or wrong choice. The selection of which algorithm to use for a particular implementation will depend upon a number of key factors. These factors can be summarized as below:

- image quality,
- computational cost,
- system integration.

Image quality is an obvious consideration. The quality of the images produced by bilinear and trilinear filtering is largely dependent upon the MIP-map level selection and the calculated trilinear interpolation fraction. All of the algorithms described were implemented within a C++ graphics pipeline simulation [6] and tested with both static

and animated scenes using a variety of textures. Fig. 15 shows results for each of the main methods when applied to a checkerboard texture image. All of the per-pixel algorithms performed well, producing similar results, with the exception of the cross-product algorithm. As expected, this algorithm tended to cause less blurring, but at the expense of exhibiting greater aliasing. The per-primitive and per vertex techniques, as expected, require high triangle tessellation where the minification variation is greatest. Important results, which cannot be shown here, are the temporal artifacts seen during animations. The filtered images, as shown in Fig. 15, may seem excessively blurred in still shots, but this may prove necessary during animation. Underfiltering leads to pixel scintillation and, in the case of per-primitive algorithms, level *popping*.

When considering the computational costs involved with each algorithm, the number representation to be used must be considered. The relative costs of additions/subtractions and multiplications will vary depending on the use of either floating-point or fixed-point data representation. For our investigations, a custom floating-point representation was used, and it was sensible to minimize the number of expensive additions.

The effect of the computational cost comparison on algorithm selection will also depend upon the balance between hardware acceleration and host-based setup operations. The direct calculation Algorithms 1.3-1.4 (Table 1) and, in particular, Algorithm 2.1, which uses incremental interpolation, require relatively low per-pixel costs at the expense of additional per-primitive setup. In contrast, for the difference Algorithms 3.1-3.6, the situation is reversed. The significance of the balance of per-primitive and per-pixel cost depends upon the hardware constraints imposed and the size of the database primitives used.

Many of the latest generation accelerators now support on-board setup. This eases the host to accelerator interface and does provide acceleration of the setup process. However, setup calculations often require a higher level of precision compared to the traditional interpolation-based rasterization pipeline. These pipelines are designed to generate a fully shaded pixel during each clock cycle. Therefore, as triangle size decreases, per-primitive costs become more important for two reasons. First, the cost is obviously spread over fewer pixels. Second, as triangles possess fewer pixels, the cost of hardware to perform the setup operation without causing a bottleneck increases.

The integration of the algorithms within the context of an entire graphics architecture must also be considered. The applicability of the algorithms will depend on the primitive types supported and traversal pattern used. In our investigations, we used triangle based scene data. The direct computation algorithm sets 1 and 4 (Table 1) are, with the exception of Algorithms 1.2 and 1.4, which utilize the independence of x and y traversal, independent of the pattern of primitive traversal and pixel generation. In contrast, Algorithm sets 2 and 3 are clearly highly dependent upon the primitive traversal algorithm used. For these algorithms, the hardware must support the storage and retrieval of pixel data. It must allow for the initial latency in acquiring this stored data for the first pixel of a primitive on each scan-line. In a graphics pipeline, this requires some form of feed forward of data. This last point applies also to Algorithms 1.2 and 1.4.

ACKNOWLEDGMENTS

We would like to acknowledge the European Commission for providing part of the funding for this work under an Esprit OMI initiative called the GraphMem Project (EP20488). We would also like to thank the many reviewers for their valuable time and expert opinions.

REFERENCES

[1] E.A. Bier and K.R. Sloan, "Two-Part Texture Mapping for Ray Tracing," *IEEE Computer Graphics and Applications*, vol. 6, no. 5, pp. 40-53, Sept. 1986.

[2] J.F. Blinn, "Hyperbolic Interpolation," *IEEE Computer Graphics and Applications*, vol. 12, no. 4, pp. 89-94, July 1992.

[3] J.F. Blinn and M.E. Newell, "Texture and Reflection in Computer Generated Images," *Comm. ACM*, vol. 19, no. 10, pp. 40-53, Oct. 1976.

[4] J.J.F. Cavanagh, *Digital Computer Arithmetic: Design and Implementation*. McGraw-Hill, 1985.

[5] F.C. Crow, "Summed Area Tables for Texture Mapping," *Computer Graphics*, vol. 18, no. 3, pp. 207-212, July 1984.

[6] J.P. Ewins, P.L. Watten, M. White, M.D.J. McNeill, and P.F. Lister, "Codesign of Graphics Hardware Accelerators," *Proc. 1997 SIGGRAPH/Eurographics Workshop Graphics Hardware*, pp. 103-110, Los Angeles, 3-4 Aug. 1997.

[7] J.P. Ewins, M.D. Waller, M. White, and P.F. Lister, "An Implementation of an Anisotropic Texture Filter," Technical Report IWD_172, Centre for VLSI and Computer Graphics, Univ. of Sussex, 1998.

[8] E.A. Feibush, M. Levoy, and R.L. Cook, "Synthetic Texturing Using Digital Filters," *Computer Graphics*, vol. 14, no. 3, pp. 294-301, July 1980.

[9] A. Fournier and E. Fiume, "Constant-Time Filtering with Space-Variant Kernels," *Computer Graphics* (*SIGGRAPH '88 Proc.*), vol. 22, no. 4, pp. 229-238, Aug. 1988.

[10] M. Gangnet, D. Perny, and P. Coueignoux, "Perspective Mapping of Planar Textures," *Proc. Eurographics '82*, vol. 16, no. 1, pp. 57-71, 1982.

[11] N. Greene and P.S. Heckbert, "Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter," *IEEE Computer Graphics and Applications*, vol. 6, no. 3, pp. 21-27, June 1986.

[12] P.S. Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, vol. 6, no. 6, pp. 56-67, Nov. 1986.

[13] P.S. Heckbert, "Texture Mapping Polygons in Perspective," Computer Graphics Lab, New York Inst. of Technology, Technical Memo no. 13, 1983.

[14] P.S. Heckbert, "Fundamentals of Texture Mapping and Image Warping," Masters thesis, Univ. of California, 7 June 1989.

[15] H. Hoppe, "View-Dependent Refinement of Progressive Meshes," *Computer Graphics Proc., SIGGRAPH '97*, pp. 189-198, Los Angeles, 3-8 Aug. 1997.

[16] S. Molnar, J. Eyles, and J. Poulton, "Pixelflow: High-Speed Rendering Using Image Composition," *Computer Graphics*, vol. 26, no. 2, pp. 231-240, July 1992.

[17] J. Pineda, "A Parallel Algorithm for Polygon Rasterisation," *Computer Graphics*, vol. 22, no. 4, pp. 17-20, Aug. 1988.

[18] A. Schilling, G. Knittel, and W. Strasser, "Texram: Smart Memory for Texturing," *IEEE Computer Graphics and Applications*, vol. 16, no. 3, pp. 32-41, May 1996.

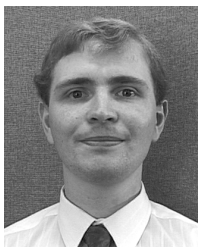
[19] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification (Version 1.1)," C. Frazier, ed., p. 95, 21 Dec. 1995.

[20] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping," *Computer Graphics Proc., SIGGRAPH '92*, vol. 26, no. 2, pp. 249-252, July 1992.

[21] A.R. Smith, "A Pixel Is Not a Little Square," *Microsoft Technical Memo*, no. 6, July 1995.

[22] I. Stamoulis, M. White, and P.F. Lister, "Floating Point Arithmetic Optimised for FPGA Resource Restrictions for Graphics Applications," Technical Report IWD_174, Centre for VLSI and Computer Graphics, Univ. of Sussex, 1998.
[23] M. Waller, "3D Rasterisation Hardware Techniques," DPhil thesis, School of Eng., Univ. of Sussex, England, 1998.
[24] A.H. Watt and M. Watt, *Advanced Animation And Rendering Techniques, Theory and Practice*, pp. 140-145. Addison-Wesley, 1992.
[25] A.H. Watt and F. Policarpo, *The Computer Image*, Chapter 4, pp. 78-85 and 91-94. ACM Press, SIGGRAPH Series, Addison-Wesley Longman Ltd, 1998.
[26] L. Williams, "Pyramidal Parametrics," *Computer Graphics*, vol. 17, no. 3, pp. 1-11, July 1983.

Jon P. Ewins received his MEng in electronic engineering from the University of Warwick in 1996. He is currently a research officer and doctoral candidate in the Centre for VLSI and Computer Graphics at the University of Sussex. His main research interests include computer graphics rendering algorithms and hardware, in particular for texture mapping and antialiasing, global illumination algorithms, and parallel architectures. He is a member of the IEEE.



Marcus Waller received his BEng degree in computer systems engineering from the University of Sussex in 1994 and his PhD in 3D rasterization hardware techniques this year. He is a research fellow in the Centre for VLSI and Computer Graphics. His main research interests are in the field of computer graphics hardware design, including hardware implementations of high-quality and real-time rasterization pipelines, texture mapping, and setup architectures.

Martin White received his BSc in computer systems engineering from the University of Sussex and his PhD in computer graphics, also from the University of Sussex. He has been a senior research staff member of the Centre for VLSI and Computer Graphics since 1993, where he has worked on and project-managed several European-funded graphics projects. He has authored or coauthored more than 45 publications. He is instrumental in a project awarded an Onyx2 IR Visualization Supercomputer for exploring new parallel architectures for global illumination, scientific visualization, and virtual reality. His main research interests are focused on geometry-based rendering algorithms for hardware implementation, e.g., scan conversion, texture mapping, antialiasing, and blending operations.

Paul F. Lister received his MSc and PhD at Aston University in Birmingham. He is a professor of electronics and director of the Centre for VLSI and Computer Graphics at the University of Sussex. He has authored or coauthored more than 130 publications. He has successfully managed several European graphics research projects funded by the European Union involving, for example, the design of a graphics workstation, and several rasterization ASICs. His main research interests include computer graphics; VLSI, ASIC, and FPGA architectures and applications; architectures for computer graphics, multimedia, and virtual reality; photorealistic rendering; graphics software; and distributed computer systems. He is a member of the ACM and the IEEE.