Hardware Acceleration of Texture Mapping

Graham Dunnett, Richard Grimsdale, Paul Lister, and Martin White

ABSTRACT
We present a hardware design based around scan-line algorithms. The design can perform colour mapping, environment mapping and produce shading effects which include a specular term. We describe the algorithms which are implemented, and the approximations we have made to achieve near real-time performance.

1.1 Introduction

Texture mapping plays an important role in the generation of high-quality images. The benefits of introducing surface detail into a scene with colour mapping, and specular reflections with environment mapping are well documented [6, 1, 14]. We predict that texture mapping hardware will become an increasingly common feature of future workstations. Most manufacture already produce high-end machines with this capability [13, 7]. The design of specialised VLSI circuitry which can perform the mapping of a two-dimensional image onto an arbitrary three-dimensional object is therefore of great interest. The need for fast update rates, increasing user-expectations, and the complex nature of texture mapping lead us to conclude that approximations to the true algorithms are necessary.

1.2 The Texture Mapping Algorithm

In his recent paper, Heckbert [11] reports that to correctly map part of a 2D image onto a triangulated object which has undergone perspective projection, the following expression must be evaluated at each pixel:-

$$\left.\begin{aligned} u &= \frac{AX+BY+C}{DX+EY+F} \\ v &= \frac{GX+HY+I}{DX+EY+F} \end{aligned}\right\} \qquad (1.1)$$

where **u** and **v** are the 2d texture coordinates varying across the image being mapped, **X** and **Y** are screen coordinates, and **A** to **I** are coefficients which vary for each surface displayed and characterise the mapping. This formulation is known as *rational linear interpolation* and is an efficient solution to texture mapping in software.

The presence of the division operator in the per pixel stage of texture mapping makes real-time performance difficult to achieve if we attempt to generate the texture coordinates with this method, either in software or with specialised hardware. No commercial processor is yet capable of performing division at a rate of 40 Million per second—the rate required for a screen refresh rate of 25 Hz, and 800,000 texture-mapped pixels per frame. In this paper we propose an hardware solution to this problem using quadratic interpolation to approximate Equation 1.1.

1.3 Quadratic Approximation

1.3.1 Approximation Equation

Equation 1.1 can be approximated with a quadratic function in two variables [13]:

$$u \approx aX^2 + bY^2 + cXY + dX + eY + f \tag{1.2}$$

A similar expression (with different coefficients a to f) will be used to approximate v. The coefficients a to f will be different for each triangle processed and must be computed before texture coordinates can be generated. To solve for the coefficients requires six equations to be established and solved simultaneously. For a given triangle we assume that the u and v components are known at each vertex. It is then straightforward to compute the exact value of u and v at the midpoints of each edge using equation 1.1. (In fact two additions and one division are all that are required per texture coordinate). This provides sufficient information to establish the following matrix equation:

$$\begin{pmatrix} X_0^2 & Y_0^2 & X_0Y_0 & X_0 & Y_0 & 1 \\ X_1^2 & Y_1^2 & X_1Y_1 & X_1 & Y_1 & 1 \\ X_2^2 & Y_2^2 & X_2Y_2 & X_2 & Y_2 & 1 \\ X_3^2 & Y_3^2 & X_3Y_3 & X_3 & Y_3 & 1 \\ X_4^2 & Y_4^2 & X_4Y_4 & X_4 & Y_4 & 1 \\ X_5^2 & Y_5^2 & X_5Y_5 & X_5 & Y_5 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix} = \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{pmatrix} \tag{1.3}$$

The 6 simulataneous equations must be solved for the coefficients a to f. Solving Equation 1.3 requires Gaussian elimination, a process which takes much computational power.

1.3.2 Forward Differencing

Once the coefficients a to f have been found, texture address generation may proceed incrementally through use of forward differencing [16].

If $f(x, y)$ represents equation 1.2, then along a scanline y is constant. For increments in x we may write

$$\triangle f_x(x, y) = f(x + 1, y) - f(x, y) = a(2x + 1) + cy + d \tag{1.4}$$

Now we notice that $\triangle f_x(x, y)$ is a linear expression. Y is still constant, so applying forward differences again we obtain

$$\triangle (\triangle f_x(x, y)) = \triangle^2 f_x(x, y) = 2a \tag{1.5}$$

The second order increment is always constant. If at each step we update $\triangle f_x(x, y)$ with $\triangle^2 f_x(x, y)$ after updating $f(x, y)$ we need only 2 additions as we step along a scanline from one pixel to the next.

An identical argument can be constructed to show how the texture coordinates vary as y is incremented and x is held constant. In this case we find

$$\triangle f_y(x, y) = b(2y + 1) + cx + e \tag{1.6}$$

$$\triangle^2 f_y(x, y) = 2b \tag{1.7}$$

1.4 Texture Filtering

1.4.1 Texture Anti-aliasing

The colour map or reflection map used in the texturing process is a raster image with discrete values. As we use the information stored in the map we are in danger of under-sampling or over-sampling, leading to jagged effects in the texture. This is identical to the artifacts experienced with geometry except that is affects the texture. Many methods have been described in the literature for reducing aliasing of texture [15, 3, 10, 8, 2].

1.4.2 Mipmaps

To assist in the filtering of the texture we use the Mipmap method [15]. This is a prefiltering technique where multiple *levels* of the texture are stored at varying resolutions. Each level stores the texture at one quarter of the resolution of the previous level. As pixels are scan converted, texture values are read from the map with the most appropriate level of detail. Conceptually, the levels are arranged to form a pyramid, and the level of detail can be thought of as the height in this pyramid. Other filtering methods have been considered, however, none are suitable for the style of hardware we are proposing.

1.4.3 Level Select

For each textured pixel we must determine which level in the mipmap holds the detail we require, and this means we need an estimate of the area of the pixel in texture space [9]. The screen pixel will map to an irregular quadrilateral in texture space, and we can approximate its area in a number of ways. The partial derivatives of Equation 1.2 are found to be useful here. Heckbert [9] recommends that the following equations are used:

$$\begin{aligned} n &= \sqrt{u_X^2 + v_X^2} \\ m &= \sqrt{u_Y^2 + v_Y^2} \\ l &= \max(n, m) \end{aligned} \tag{1.8}$$

In Equation 1.8, l is the level in the pyramid we require. This formula assumes that the quadrilateral can be approximated by a parallelogram, and uses the maximum side length to select the level.

1.4.4 Blending

In general, selecting a texel from one level will over-, or under-sample the texture, depending how close the area is to one level or the next. Instead, it is preferable to select two levels and blend the two texels to better approximate the true texel value. The blend ratio required for this is related to how close the pixel area in texture space is to either of the two levels. If the area is known then the blending is easy to perform.

1.5 Postprocessing

1.5.1 Gouraud Shading Problems

When texture mapping is used it is not possible to use the well-loved Gouraud shading algorithm to perform the shading task. The reason for this is simply that the colour of the surface is not known until after texturing is performed (a pixel rate computation). This requires us to perform pixel-rate shading, which can introduce a potentially severe

bottleneck into a rendering system, particularly if advanced shading techniques are used to produce highlights.

1.5.2 Diffuse Interpolation

If the surface colour components are factored out of the traditional Gouraud shading illumination equation, we are left with an ambient and diffuse term which describes the incoming illumination received by a surface. This illumination can be interpolated across a triangle primitive, and combined with the (texture mapped) surface colour at each pixel. A second way of explaining this is that we pretend the surface is white and interpolate the intensity of the surface before blending in its true colour. As in Gouraud shading, we sum the diffuse contributions from all light sources in the scene and take the geometry of the surface into consideration.

Technique	*Interpolate*
Gouraud Shading:	$C_d I_a k_a + \sum C_d I_{d_i} k_d (\mathbf{L}_i \cdot \mathbf{N})$
Diffuse Shading:	$I_a k_a + \sum I_{d_i} (\mathbf{L}_i \cdot \mathbf{N})$

1.5.3 Specular Interpolation

High-lights

The Gouraud shading model does not use a specular term. One reason for this is that the edges of a highlight are straightened and look unnatural. The linearization is a direct consequence of linearly interpolating colour. Adaptively subdividing triangles in the vincinity of highlights can improve the image quality, although at the expense of additional triangle setup costs, and identifying candidate triangles. An alternative is to use a quadratic interpolation scheme for the specular term [4]. This permits the boundary of the highlight to be curved, giving a more natural appearance. A setup, identical to that discussed in Section 1.3 can be performed, with incoming specular intensity replacing texture coordinates.

Reflections

Environment maps can be used to give the impression of mirror reflections in a scene. Rather than using texture coordinates to access these maps, an indexing *direction* is used instead. This direction is the reflected view direction vector which is used to intersect an axis-aligned plane containing the map itself. The reflection map stores the colour of the incoming light received from that direction. Figure 1.1 shows a simple geometry illustrating the concepts of reflection mapping.

Note that reflection mapping can only be used to model perfectly reflecting surfaces. No scattering of the reflected light is possible such as from rough surfaces. Pre-computing the reflection map with this scattering included is non-trivial. Light sources could be incorporated into these maps, however, this alters the map creation step, which normally is just a straightforward rendering of the scene from the position of the shiny object.

Clearly it is not feasible to perform real-time intersection testing. Instead a setup task can compute intersections at triangle vertices and midpoints. These give map coordinates which can be interpolated across the triangle. Colour map coordinates and reflection map coordinates can then be treated in an identical manner. This is immediately attractive because the texture coordinate generator and address synthesis hardware can be duplicated.

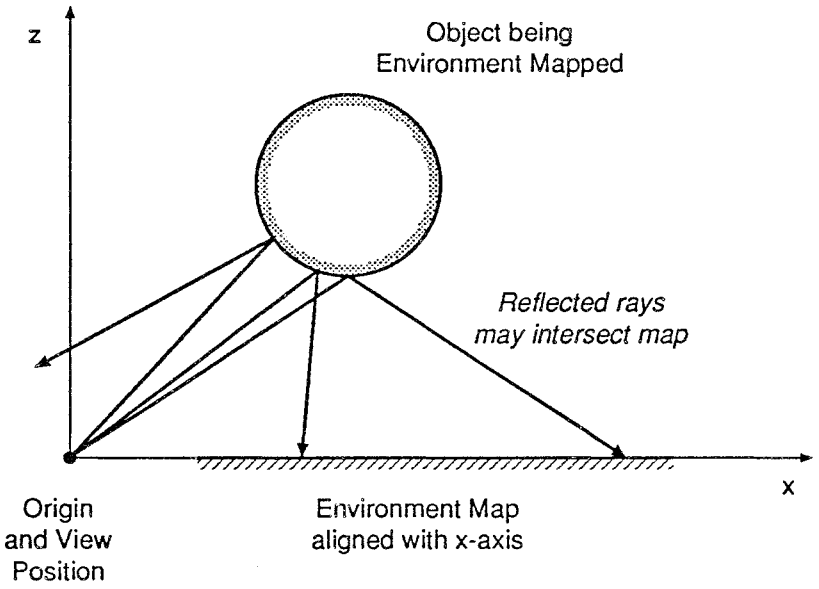

FIGURE 1.1. Reflected Rays are Intersected with the Reflection Map

1.6 Texture Mapping Hardware

1.6.1 Texture Coordinate Generator

Figure 1.2 shows the interpolation stage we have designed to produce texture coordinates for each pixel. Two interpolation units operating in parallel will be required to generate u and v. The interpolation unit design is straightforward, being very similar to linear units we have designed in the past [5]. Fixed point arithmetic is used throughout. The figure shows that in parallel to the increment for the texture coordinate u, the first order X and Y derivatives are updated by second order terms. The design shown here can increment/decrement in X (along scanlines) or Y (from scanline to scanline). Our current scan-conversion controller does not need to decrement in Y, and so we may remove this capability from the design. The data widths are being analysed for optimisation. A complication exists in that we may want to use the quadratic interpolation unit for other purposes. See Section 1.5.3. This may require us to redesign the interpolation units to use floating point arithmetic. We permit replications of the texture up to 16 times in each of the u and v directions. This allows tiling of a single texture, and reduces the storage requirements for a texture which repeats. Texture maps may be stored at resolutions up to 512×512, and so the f output must have a width of $4 + 9 = 13$ bits. Hardware can be used to clamp the output to the range (0..1] if wrapping is not wanted. (The test-multiplex stage for this is not shown in the figure.)

1.6.2 Control Signals

A small state machine is used to control the scan conversion of each triangle primitive. Signals provided by the logic include whether an increment or decrement has been made along a scanline, or whether an increment or decrement has been made to a new scanline. These signals are used to control whether an x or y increment is made to the texture coordinate value, and whether the first order differences need updating. The table below shows how the quadratic interpolator signals are generated from the output of the scan-conversion state machine.

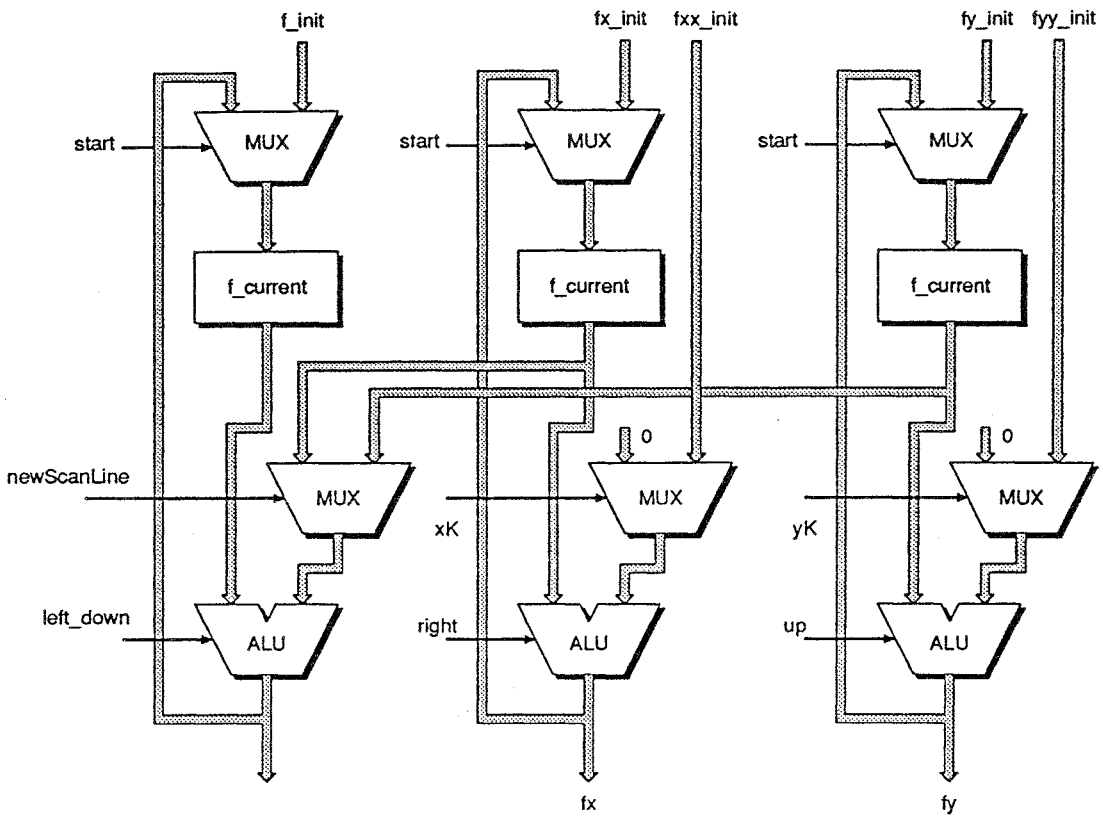

FIGURE 1.2. A Quadratic Interpolation Unit

Signal	*Creation*
`newScanLine`	`up OR down`
`xK`	`up OR down`
`yK`	`left OR right`
`left_down`	`left OR down`

1.6.3 Mipmapping Hardware

Figure 1.2 shows that the first-order differences f_x and f_y are output along with the texture coordinate f. In conjunction with the second interpolation unit, this provides all the information we need to perform mipmapping. A LUT, using the partial differences as input, generates the mipmap levels and blending factor needed for correct filtering. The LUT stores entries implementing Equation 1.8. The optimal size of the LUT is still under investigation. Once the mipmap level has been determined two addresses are computed using a simple address generation unit. This takes the texture coordinates and the mipmap level as input. The level identifier is decoded into two physical start addresses in memory, corresponding to the two mipmap levels. An offset is computed using the texture coordinates and two addresses synthesised. The two texels can then be accessed in parallel over two 32-bit buses. We arrange the texture memory to ensure that all even levels of the mipmap are stored in one memory bank, and odd levels in another. Accesses can then be made in parallel to both banks. The texture information stored in the memories is red, green, blue and alpha, and each is stored in 8 bits. Once texels have been returned, four 8-bit add-multiply stages perform the blending on the four channels using the blending factor produced from the LUT. The results are then passed forward to the

general blending units for shading and other image synthesis tasks.

1.7 Blending Hardware

1.7.1 General Blending Unit

As identified in Section 1.5 there are many uses for blending operations at the rear end of a rendering pipeline. Our hardware design recognises this and we have a flexible network of blending stages. Each stage is capable of performing:

$$C_i = ((A_i - B_i)\alpha_i + B_i) \tag{1.9}$$

on three parallel channels, where A, B and α are inputs, and C is an output, and i varies from 0 to 2. Figure 1.3 shows the architecture of our blend unit. This uses basic library components and is simple in design.

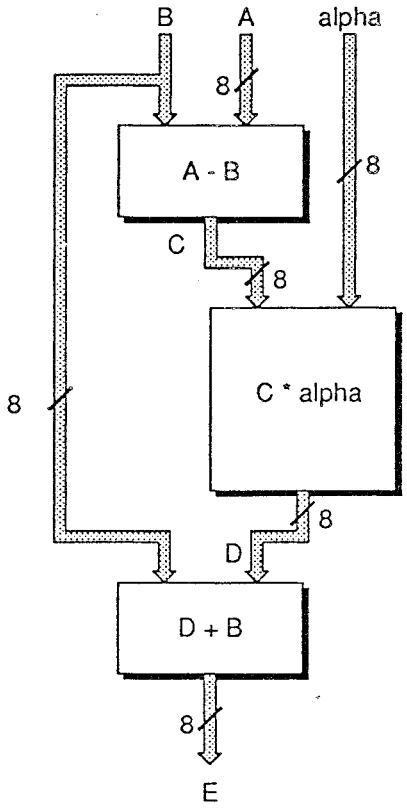

FIGURE 1.3. A Blending Unit

The output D shown in the figure is the 8 high order bits of the 16 bit result of the multiplication. This arrangement considers the α and C channels to be 8 bit, fixed point numbers between 0 and 1.

1.7.2 Blend Pipeline

Colour Mixing

Texture-maps may be used to perturb (or modulate) surface colour rather than provide surface colour. This action can mix the colour of the surface with the colour derived from

the texture access. A blending unit is required to perform this task. In our design we allow the blending ratio to be constant across an object, and be provided by the user, or vary, and come from the alpha channel of the texture map.

1.7.3 Diffuse Illumination

Linearly interpolated diffuse illumination is is used to modify the pixel colour. A second blending unit is used for this purpose. The illumination is just a weighting of the incoming colour by the diffuse illumination, and so the B channel of the blend unit is connected to a zero input. This achieves the weighting correctly, and the output is the illuminated surface colour.

1.7.4 Specular Accumulation

Specular reflection from a surface may come from two sources, the environment or lights in the scene. A quadratic interpolation of the specular lights will be performed, and combined with values read from an environment map stored in the framebuffer. The environment map will have been produced by an earlier scanconversion pass, and so will already exist in the framebuffer. A separate pair of quadratic interpolators will be necessary to generate the environment map addresses, permiting both colour mapping and reflection mapping to be performed on each surface. The highlight and incoming specularly reflected light contributions should be summed together to give the total specularly reflected component. In our system, however, we provide more flexibility by performing a linear blend between these two values. This allows finer control over how bright the specular highlights are, compared to the reflected objects. The user will supply the blend ratio Ω for this operation as part of the scene database. A blending unit is used for this.

1.7.5 Specular Illumination

The final stage in the blending pipeline is to mix the diffusely illuminated surface with the specularly reflected light. Once again we use a blending unit to perform this task. Allowing a mixing between the diffuse colour and specular colour is an approximation, but will give the user the ability to produce non-physical effects. In addition this method helps prevent colour component overflow which is often a problem in scenes with multiple light sources. The blend ratio provided by the user for this stage we call Π.

1.7.6 Overall Network

Figure 1.4 illustrates the overall blending network we have designed to perform colour mapping, environment mapping and illumination with a specular component.

1.8 Results and Conclusion

A software model of the interpolation and blending hardware has been produced. This has been written in 'C'. Results obtained from this indicate that the approach we have taken is valid. In particular the quadratic approximation to equation 1.1 is accurate to within 2 pixels in 200, or 1 percent. Figures 1.5, and 1.6 show rational linear interpolation and quadratic interpolation of texture coordinates, respectively. The differences in these two images are shown in Figure 1.7. The chequerboard is a *worst-case* texture, with the eye easily picking out errors in the mapping. For less well-defined textures we find that the approximation performs well.

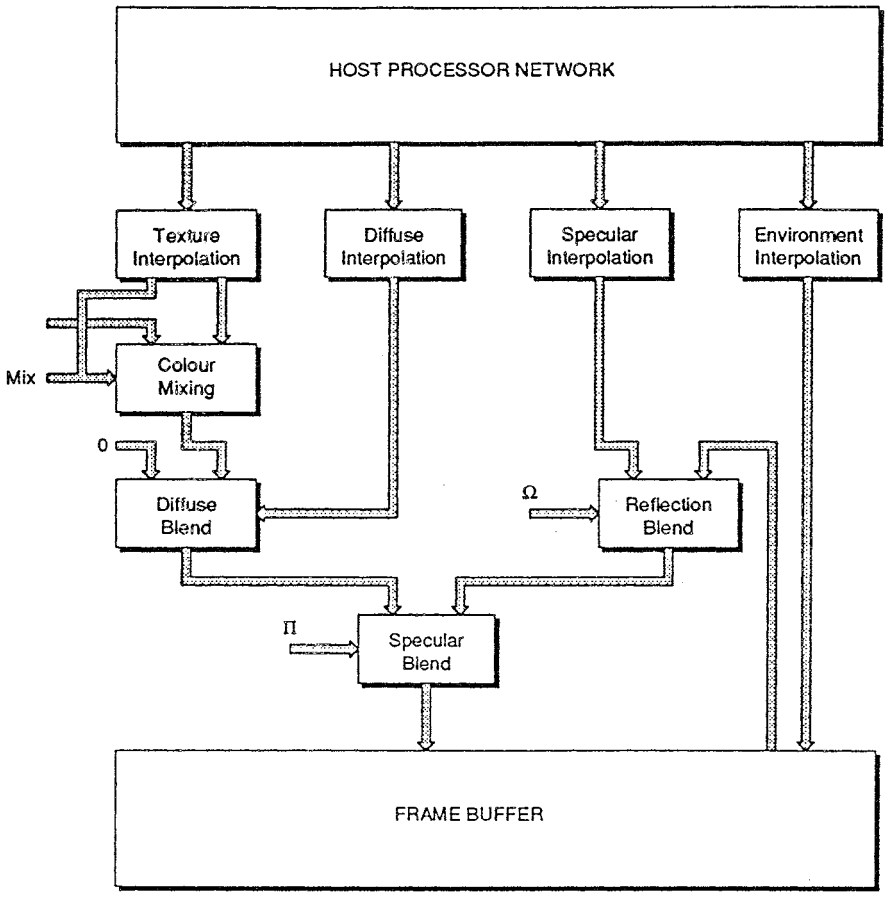

FIGURE 1.4. The Blending Network

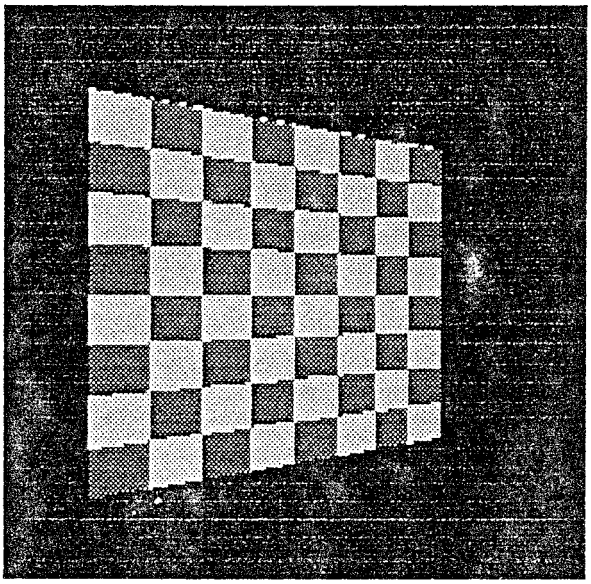

FIGURE 1.5. Rational Linear Interpolation

1.8.1 Further Work

Setup

We consider that the setup cost for quadratic interpolation is rather high. We are looking at alternatives to the Gaussian elimination. One such area under investigation is to expand

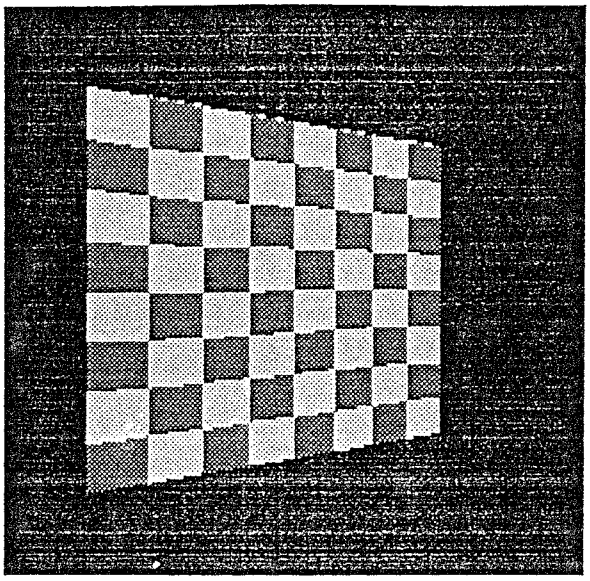

FIGURE 1.6. Quadratic Interpolation

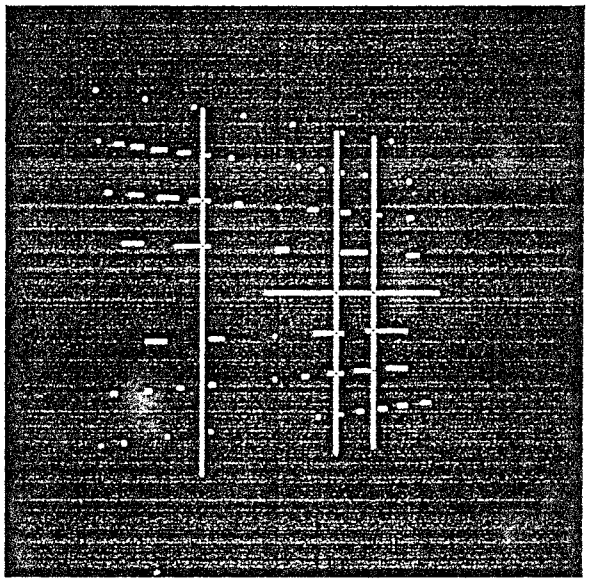

FIGURE 1.7. Differences Between Techniques

Equation 1.1 as a power series in X and Y. This approach will avoid the matrix processing.

Bit Blitting.

We can consider the scan-conversion hardware and quadratic interpolators as simple address generators, accessing different memories (frame-buffer, texture memory and environment map buffers). Operations are performed between the values returned. This is comparable to hardware which can perform Bit-blitting such as the TMS34010 graphics processor [12], although the range of operations implemented in our design is limited. We plan to investigate how suited our blending pipeline is to perform boolean operations between channels. This will involve a re-design of our blending unit.

1.9 Acknowledgements

This project is part of the Esprit program supported by the European Commission.

The authors would like to thank Mike McNeill, Ian McGroarty, Simon Pearce and others in the VLSI and Computer Graphics Research Group for their useful comments and suggestions through-out the course of this work.

1.10 References

[1] James F. Blinn and Martin E. Newell. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19, October 1976.

[2] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. *Computer Graphics*, 21(4), July 1987.

[3] Franklin C. Crow. Summed-Area Tables for Texture Mapping. *Computer Graphics*, Vol. 18, July 1984.

[4] Vincent C. J. Disselkoen. Real-time Quadratic Shading. Internal Report CS-R9123, Centre for Mathematics and Computer Sciences, The Netherlands, Kruislaan 413, 1098 SJ Amsterdam, 1991.

[5] Graham Dunnett, Martin White, Paul Lister, Richard Grimsdale, and France Glemot. The IMAGE Chip for High Performance 3D Rendering. *Computer Graphics and Applications*, 1992. Submitted for Inclusion in the November Special Issue on Graphics Hardware.

[6] James D. Foley, Andreas Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice.* Addison Wesley, 2nd edition, 1990.

[7] Silicon Graphics. IRIS Crimson Technical Report: Pre-introduction. Technical report, Silicon Graphics, 1991.

[8] Ned Greene and Paul S. Heckbert. Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter. *IEEE Computer Graphics and Applications*, 6(6), June 1986.

[9] Paul S. Heckbert. Texture Mapping Polygons in Perspective. Technical Report 13, Computer Graphics Lab, New York Institute of Technology, April 1983.

[10] Paul S. Heckbert. Survey of Texure Mapping. *Computer Graphics and Applications*, 6(11), November 1986.

[11] Paul S. Heckbert and Henry P. Moreton. Interpolation for Polygon Texture Mapping and Shading. In David Rogers and Rae Earnshaw, editors, *State of the Art in Computer Graphics. Visualization and Modeling.* Springer Verlag, 1991.

[12] Carrel R. Killebrew Jr. The TMS34010 Graphics System Processor. *Byte*, pages 193–204, December 1986.

[13] David Kirk and Douglas Voorhies. The Rendering Architecture of the DN10000VS. *Computer Graphics*, 24(4), August 1990.

[14] Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison Wesley, 1st edition, 1990.

[15] Lance Williams. Pyramidal Parametrics. *Computer Graphics*, Vol. 17, July 1983.

[16] George Wolberg. *Digital Image Warping.* IEEE Computer Society Press, 1990.